Бесплатная электронная книга

УЧУСЬ

Swift Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#swift

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: swift-language

It is an unofficial and free Swift Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Swift Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Swift Language

замечания

Swift - это язык приложений и системного программирования, разработанный Apple и распространяемый как открытый . Swift взаимодействует с API-интерфейсами Objective-C и Cocoa / Cocoa для Apple MacOS, iOS, tvOS и watchOS. В настоящее время Swift поддерживает macOS и Linux. В настоящее время предпринимаются усилия для поддержки Android, Windows и других платформ.

Быстрое развитие происходит на GitHub ; взносы обычно отправляются по запросу .

Ошибки и другие проблемы отслеживаются на сайте bugs.swift.org .

Обсуждения о развитии, эволюции и использовании Swift хранятся в списках рассылки Swift .

Другие источники

- Swift (язык программирования) (Википедия)
- Быстрый язык программирования (онлайн)
- Справочная справочная библиотека Swift (онлайн)
- Руководства по разработке API (онлайн)
- Быстрое программирование (iBooks)
- ... и многое другое на developer.apple.com .

Версии

Быстрая версия	Версия Xcode	Дата выхода
началось развитие (первая фиксация)	-	2010-07-17
1,0	Xcode 6	2014-06-02
1,1	Xcode 6.1	2014-10-16
1.2	Xcode 6.3	2015-02-09
2,0	Xcode 7	2015-06-08
2,1	Xcode 7.1	2015-09-23

Быстрая версия	Версия Xcode	Дата выхода
дебют с открытым исходным кодом	-	2015-12-03
2,2	Xcode 7.3	2016-03-21
2,3	Xcode 8	2016-09-13
3.0	Xcode 8	2016-09-13
3,1	Xcode 8.3	2017-03-27

Examples

Ваша первая программа Swift

Напишите свой код в файле `hello.swift` :

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

- Чтобы скомпилировать и запустить скрипт за один шаг, используйте `swift` из терминала (в каталоге, где находится этот файл):

  Чтобы запустить терминал, нажмите `CTRL` + `ALT` + `T` на *Linux* или найдите его в Launchpad на *macOS* . Чтобы изменить каталог, введите `cd` **`directory_name`** (или `cd ..` для возврата)

```
$ swift hello.swift
Hello, world!
```

  Компилятор представляет собой компьютерную программу (или набор программ), которая преобразует исходный код, написанный на языке программирования (исходный язык) на другой язык компьютера (целевой язык), причем последний часто имеет двоичную форму, известную как объектный. (Википедия)

- Чтобы скомпилировать и запустить отдельно, используйте `swiftc` :

```
$ swiftc hello.swift
```

Это скомпилирует ваш код в файл `hello` . Чтобы запустить его, введите `./` , а затем имя файла.

```
$ ./hello
Hello, world!
```

- Или используйте быстрый REPL (Read-Eval-Print-Loop), набрав `swift` из командной строки, а затем введите свой код в интерпретаторе:

Код:

```
func greet(name: String, surname: String) {
    print("Greetings \(name) \(surname)")
}

let myName = "Homer"
let mySurname = "Simpson"

greet(name: myName, surname: mySurname)
```

Давайте разложим этот большой код на части:

- `func greet(name: String, surname: String) { // function body }` - создать *функцию,* которая принимает `name` и `surname` .
- `print("Greetings \(name) \(surname)")` - это выдает на консоль «Приветствия», затем `name` , затем `surname` . В основном `\( variable_name )` выводит значение этой переменной.
- `let myName = "Homer"` и `let mySurname = "Simpson"` - создайте *константы* (переменные, значения которых вы не можете изменить), используя `let` с именами: `myName` , `mySurname` и значения: `"Homer"` , `"Simpson"` соответственно.
- `greet(name: myName, surname: mySurname)` - вызывает *функцию,* которую мы создали ранее, предоставляя значения *констант* `myName` , `mySurname` .

Запуск с использованием REPL:

```
$  swift
Welcome to Apple Swift. Type :help for assistance.
  1> func greet(name: String, surname: String) {
  2.     print("Greetings \(name) \(surname)")
  3. }
  4>
  5> let myName = "Homer"
myName: String = "Homer"
  6> let mySurname = "Simpson"
mySurname: String = "Simpson"
  7> greet(name: myName, surname: mySurname)
Greetings Homer Simpson
  8> ^D
```

Нажмите `CTRL` + `D`, чтобы выйти из REPL.

Установка Swift

Сначала загрузите компилятор и компоненты.

Затем добавьте Swift к вашему пути. В macOS расположение по умолчанию для загружаемой инструментальной цепочки - это / Library / Developer / Toolchains. Выполните следующую команду в терминале:

```
export PATH=/Library/Developer/Toolchains/swift-latest.xctoolchain/usr/bin:"${PATH}"
```

В Linux вам нужно будет установить clang:

```
$ sudo apt-get install clang
```

Если вы установили Swift toolchain в каталог, отличный от системного root, вам нужно будет выполнить следующую команду, используя фактический путь вашей установки Swift:

```
$ export PATH=/path/to/Swift/usr/bin:"${PATH}"
```

Вы можете проверить, есть ли у вас текущая версия Swift, выполнив эту команду:

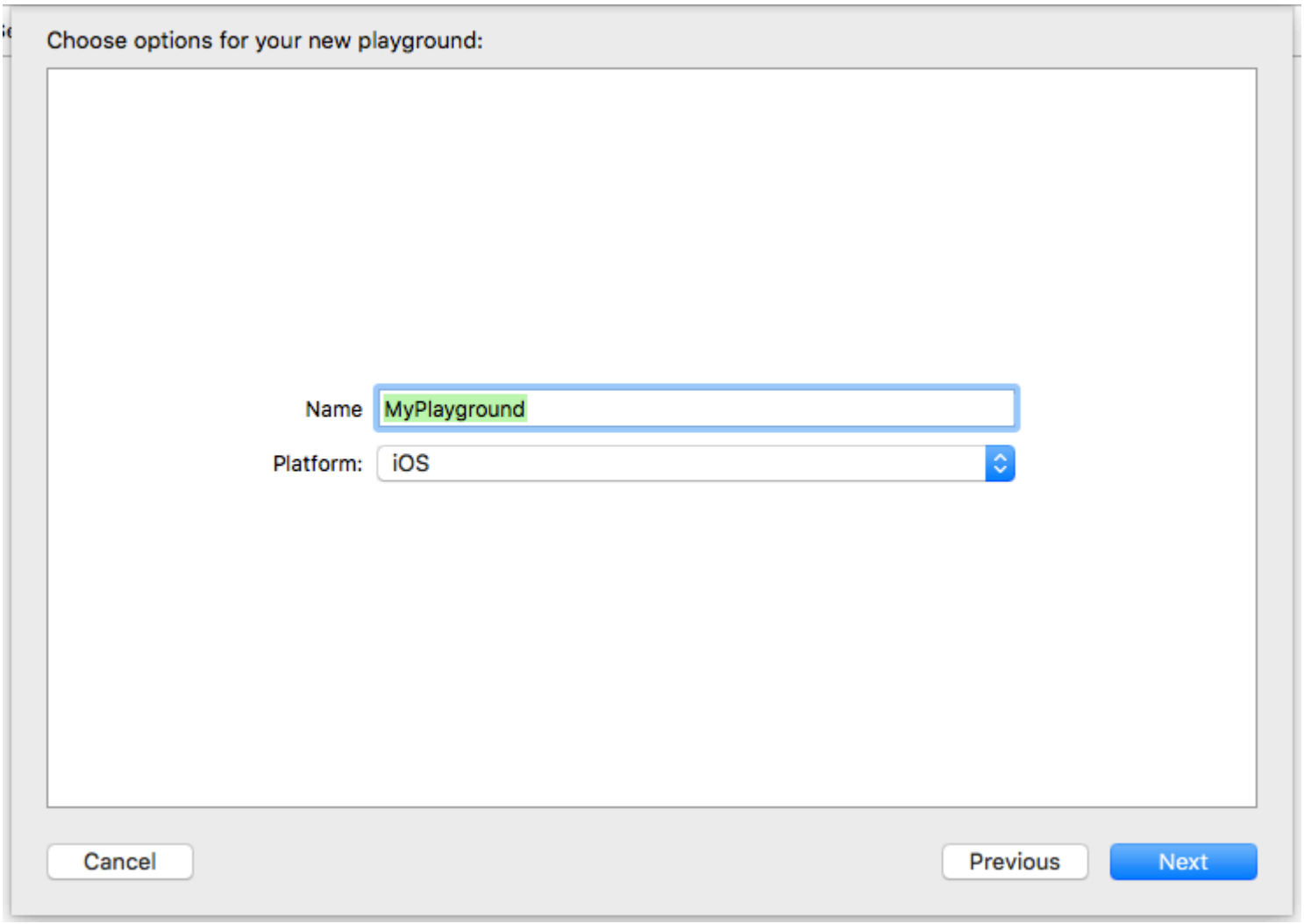
```
$ swift --version
```

Ваша первая программа в Swift на Mac (с использованием игровой площадки)

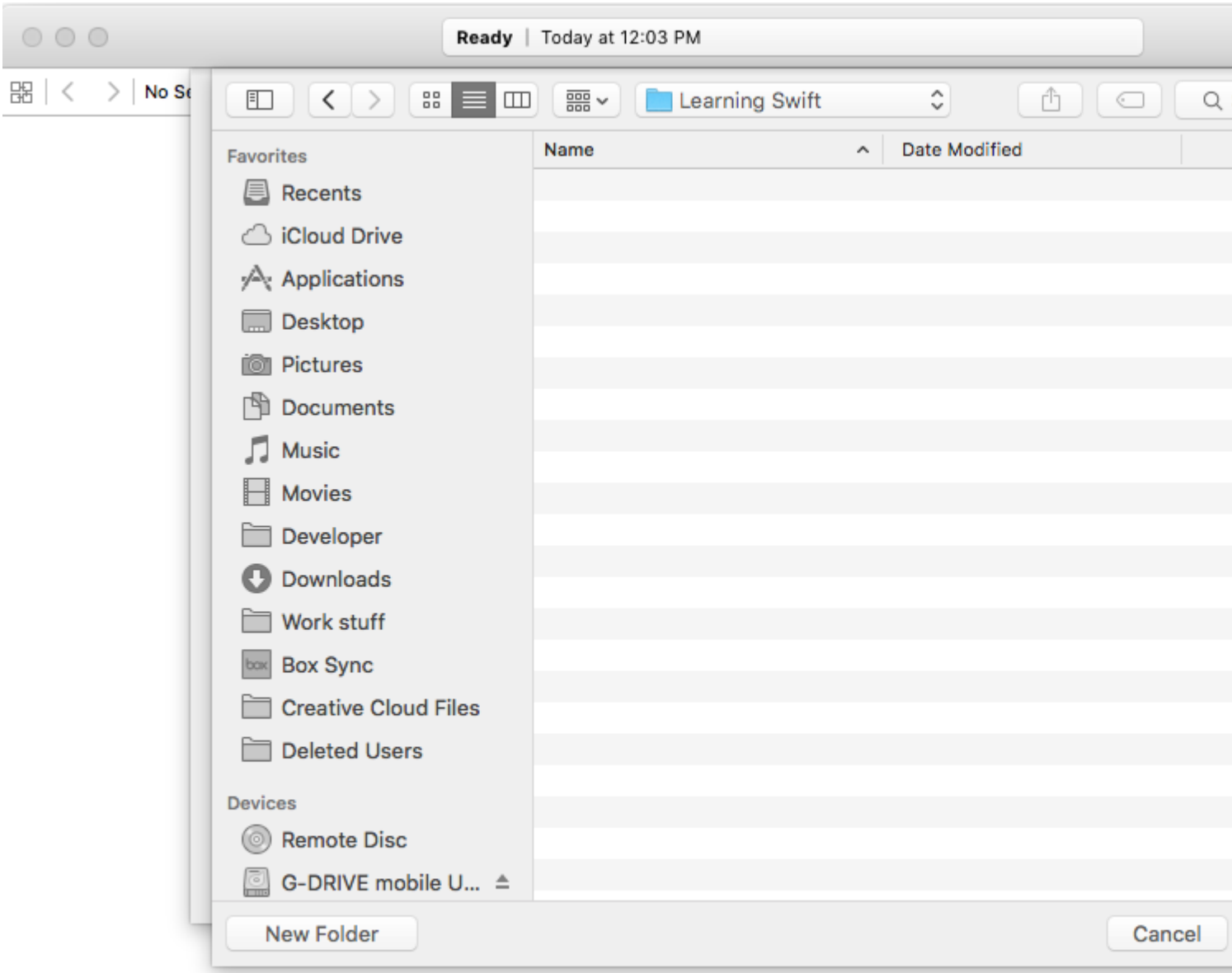
С вашего Mac загрузите и установите Xcode из Mac App Store по этой ссылке .

По завершении установки откройте Xcode и выберите « **Начать с игровой площадки»** :

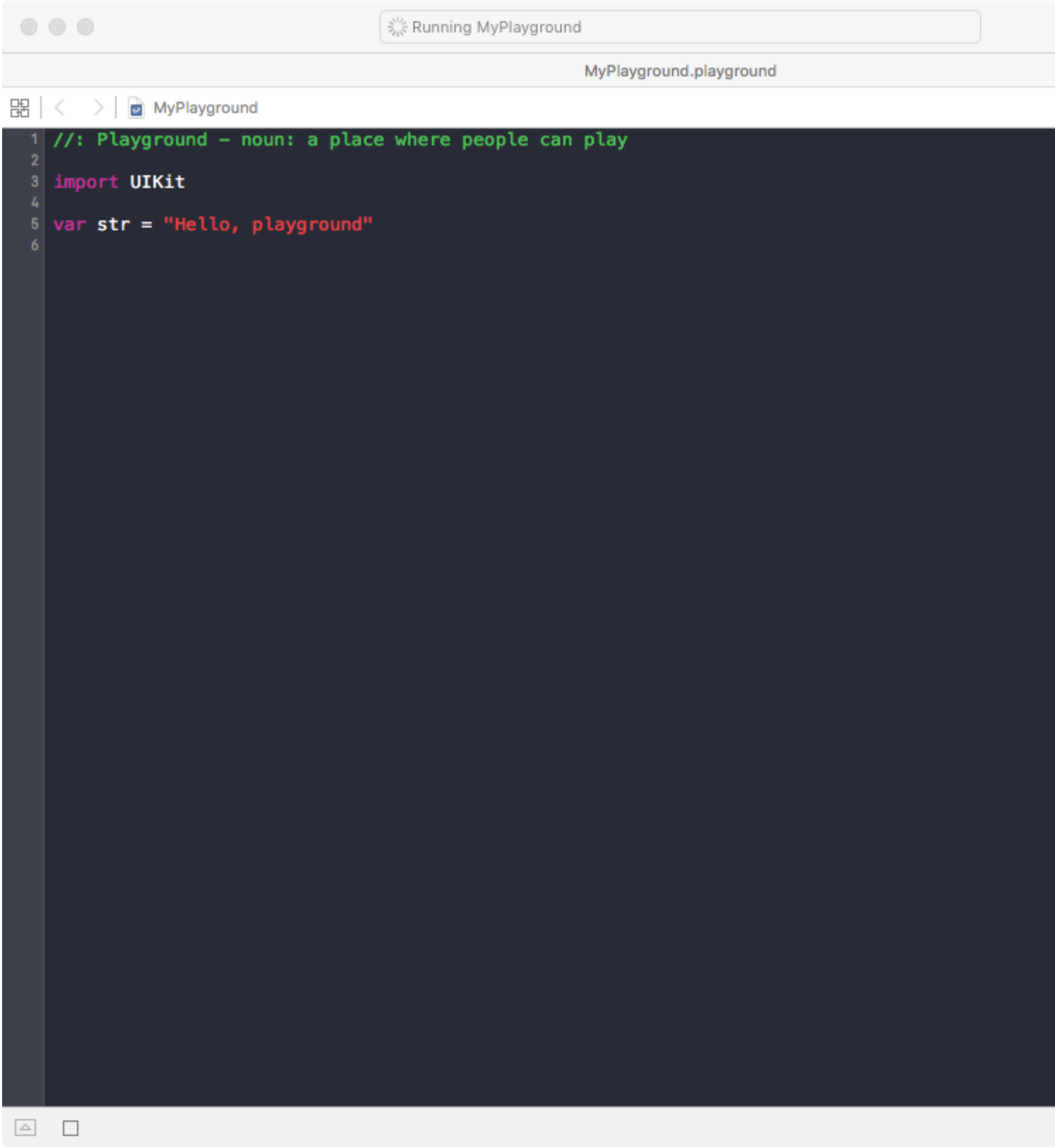
На следующей панели вы можете `MyPlayground` площадке имя или оставить ее на `MyPlayground` и нажать **Next** :



Выберите место для сохранения игровой площадки и нажмите « **Создать»** :



Игровая площадка откроется, и ваш экран должен выглядеть примерно так:

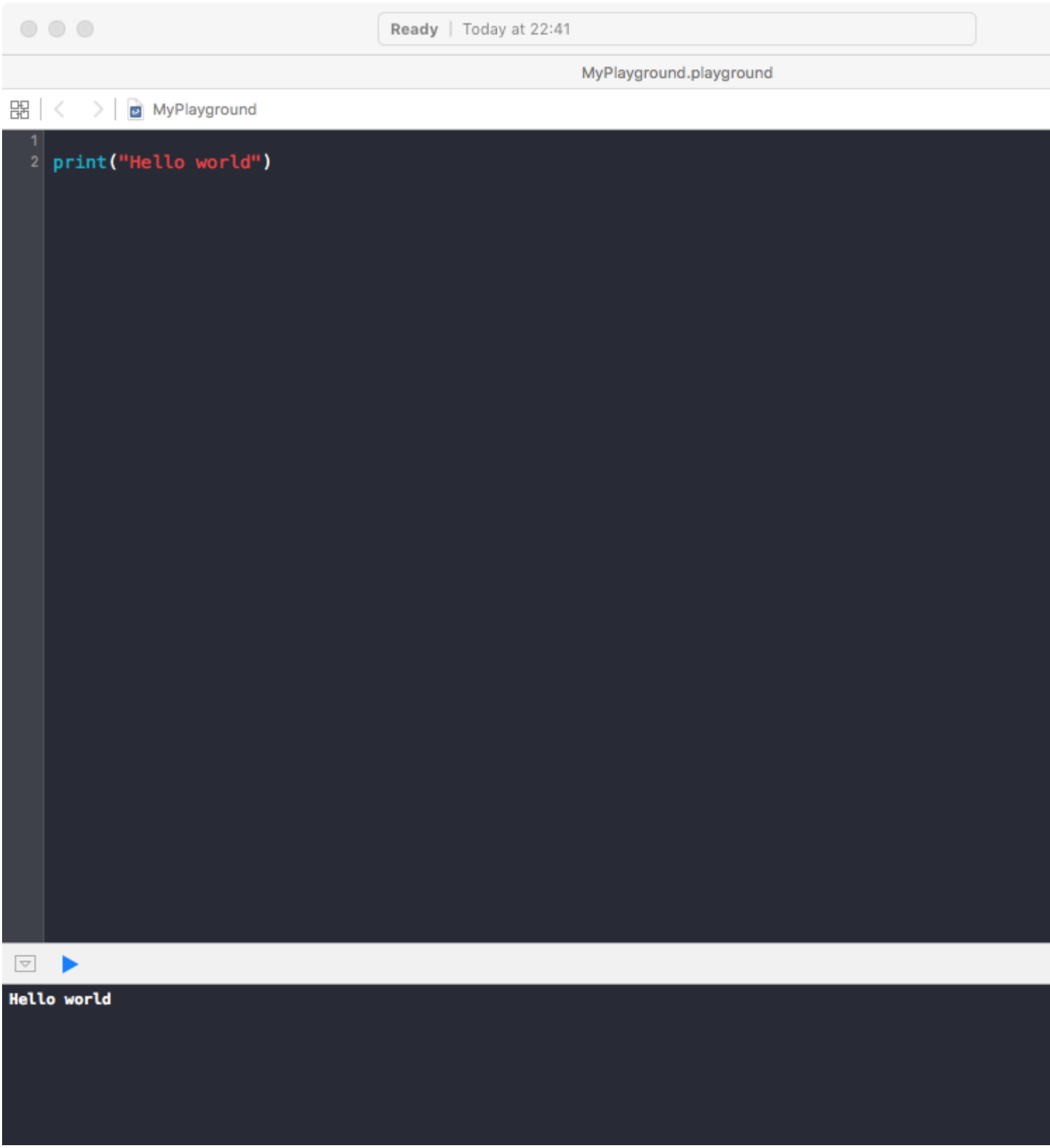


Теперь, когда игровая площадка находится на экране, нажмите + cmd + Y, чтобы отобразить область **отладки** .

Наконец, удалите текст внутри игровой площадки и введите:

```
print("Hello world")
```

Вы должны увидеть «Hello world» в области **Debug** и «Hello world \ n» в правой **боковой панели** :



Поздравляем! Вы создали свою первую программу в Swift!

Ваша первая программа в приложении Swift Playgrounds на iPad

Приложение Swift Playgrounds - отличный способ начать кодирование Swift на ходу. Чтобы использовать его:

1- Загрузите Swift Playgrounds для iPad из App Store.

2. Откройте приложение

3. На вкладке « **Мои игровые площадки** » нажмите « + »

в верхнем левом углу, а затем выберите «Пустой шаблон».

4 Введите свой код.

5- Нажмите «Запустить мой код», чтобы запустить код.

6- В передней части каждой строки результат будет сохранен на небольшом квадрате. Нажмите его, чтобы показать результат.

7- Чтобы медленно пройти по коду, чтобы проследить его, нажмите кнопку рядом с «Запустить мой код».

Необязательное значение и необязательное перечисление

Тип опций, который обрабатывает отсутствие значения. Опционы говорят, что «есть значение, и оно равно x» или «вообще нет значения».

Необязательный - это тип сам по себе, на самом деле один из новых сверхмощных перечислений Swift. Он имеет два возможных значения: `None` и `Some(T)` , где T - связанное значение правильного типа данных, доступного в Swift.

Давайте посмотрим на этот фрагмент кода, например:

```
let x: String? = "Hello World"

if let y = x {
   print(y)
}
```

Фактически, если вы добавите **`print(x.dynamicType)`** в приведенный выше код, вы увидите это в консоли:

```
Optional<String>
```

Строка? на самом деле является синтаксическим сахаром для Необязательного, а Необязательный - это сам по себе.

Вот упрощенная версия заголовка Необязательно, которую вы можете увидеть, нажав команду на слово Необязательно в коде из Xcode:

```
enum Optional<Wrapped> {

 /// The absence of a value.
 case none

 /// The presence of a value, stored as `Wrapped`.
 case some(Wrapped)
}
```

Необязательно на самом деле перечисление, определенное в отношении общего типа Wrapped. У этого есть два случая: `.none` чтобы представить отсутствие значения, и `.some` чтобы представить наличие значения, которое сохраняется как его связанное значение типа Wrapped.

Позвольте мне пройти через это снова: `String?` не является `String` а `Optional<String>` . Тот факт, что `Optional` - это тип, означает, что он имеет свои собственные методы, например `map` и `flatMap` .

Прочитайте Начало работы с Swift Language онлайн: https://riptutorial.com/ru/swift/topic/202/начало-работы-с-swift-language

глава 2: (Небезопасные) Буферные указатели

Вступление

«Буферный указатель используется для низкоуровневого доступа к области памяти. Например, вы можете использовать указатель буфера для эффективной обработки и передачи данных между приложениями и службами ».

Выдержка из: Apple Inc. «Использование Swift с Cocoa и Objective-C (Swift 3.1 Edition)». IBooks. https://itun.es/us/utTW7.l

Вы несете ответственность за обработку жизненного цикла любой памяти, с которой работаете, с помощью указателей буфера, чтобы избежать утечек или неопределенного поведения.

замечания

Тщательно выровненные концепции, **необходимые** для полного понимания (Unsafe) BufferPointers.

- MemoryLayout (*макет памяти типа, описывающий его размер, шаг и выравнивание*).
- Неуправляемый (*тип для распространения ссылки на неуправляемый объект*).
- UnsafeBufferPointer (Неприемлемый *интерфейс коллекции для буфера элементов, хранящихся в памяти в памяти* .)
- UnsafeBufferPointerIterator (*Итератор для элементов в буфере, на который ссылается экземпляр UnsafeBufferPointer или UnsafeMutableBufferPointer*).
- UnsafeMutableBufferPointer (Неприемлемый *интерфейс коллекции для буфера изменяемых элементов, хранящихся в памяти в памяти*).
- UnsafeMutablePointer (*указатель для доступа и управления данными определенного типа*).
- UnsafeMutableRawBufferPointer (*изменяемый интерфейс сбора недействительных байтов в области памяти*).
- UnsafeMutableRawBufferPointer.Iterator (*Итератор по байтам, просмотренный указателем необработанного буфера*).
- UnsafeMutableRawPointer (*необработанный указатель для доступа и обработки нетипизированных данных*).
- UnsafePointer (*указатель для доступа к данным определенного типа*).
- UnsafeRawBufferPointer (*Недействующий интерфейс сбора байтов в области памяти.*)
- UnsafeRawBufferPointer.Iterator (*Итератор по байтам, просматриваемый указателем необработанного буфера*).

- UnsafeRawPointer (*необработанный указатель для доступа к нетипизированным данным.*)

  (Источник, Swiftdoc.org)

Examples

UnsafeMutablePointer

```
struct UnsafeMutablePointer<Pointee>
```

Указатель для доступа и обработки данных определенного типа.

Вы используете экземпляры типа UnsafeMutablePointer для доступа к данным определенного типа в памяти. Тип данных, к которым может обращаться указатель, - тип Pointee указателя. UnsafeMutablePointer не обеспечивает автоматические функции управления памятью или выравнивания. Вы несете ответственность за обработку жизненного цикла любой памяти, с которой работаете, с помощью небезопасных указателей, чтобы избежать утечек или неопределенного поведения.

Память, которую вы вручную управляете, может быть либо непечатана, либо привязана к определенному типу. Тип UnsafeMutablePointer используется для доступа и управления памятью, привязанной к определенному типу. (Источник)

```
import Foundation

let arr = [1,5,7,8]

let pointer = UnsafeMutablePointer<[Int]>.allocate(capacity: 4)
pointer.initialize(to: arr)

let x = pointer.pointee[3]

print(x)

pointer.deinitialize()
pointer.deallocate(capacity: 4)

class A {
  var x: String?

  convenience init (_ x: String) {
    self.init()
    self.x = x
  }

  func description() -> String {
    return x ?? ""
  }
}
```

```
let arr2 = [A("OK"), A("OK 2")]
let pointer2 = UnsafeMutablePointer<[A]>.allocate(capacity: 2)
pointer2.initialize(to: arr2)

pointer2.pointee
let y = pointer2.pointee[1]

print(y)

pointer2.deinitialize()
pointer2.deallocate(capacity: 2)
```

Преобразован в Swift 3.0 из исходного источника

Практический пример использования для буферов

Деконструирование использования небезопасного указателя в библиотечном методе Swift;

```
public init?(validatingUTF8 cString: UnsafePointer<CChar>)
```

Цель:

Создает новую строку, копируя и проверяя данные UTF-8 с нулевым завершением, на которые ссылается данный указатель.

Этот инициализатор не пытается восстановить неправильно сформированные последовательности кода UTF-8. Если они найдены, результат инициализатора равен `nil` . Следующий пример вызывает этот инициализатор с указателями на содержимое двух разных массивов `CChar` - первый с хорошо сформированными последовательностями кода кода UTF-8, а второй с плохо сформированной последовательностью в конце.

> **Source** , **Apple Inc., файл заголовка Swift 3** (для доступа к заголовку: на игровой площадке, Cmd + щелкните по слову Swift) в строке кода:
>
> ```
> import Swift
> ```

```
let validUTF8: [CChar] = [67, 97, 102, -61, -87, 0]
     validUTF8.withUnsafeBufferPointer { ptr in
         let s = String(validatingUTF8: ptr.baseAddress!)
         print(s as Any)
     }
     // Prints "Optional(Café)"

     let invalidUTF8: [CChar] = [67, 97, 102, -61, 0]
     invalidUTF8.withUnsafeBufferPointer { ptr in
      let s = String(validatingUTF8: ptr.baseAddress!)
      print(s as Any)
     }
// Prints "nil"
```

> (Источник, Apple Inc., пример файла Swift Header)

Прочитайте (Небезопасные) Буферные указатели онлайн:
https://riptutorial.com/ru/swift/topic/9140/-небезопасные--буферные-указатели

глава 3: Conditionals

Вступление

Условные выражения, включающие ключевые слова, такие как if, else if и else, предоставляют программам Swift возможность выполнять разные действия в зависимости от логического условия: True или False. В этом разделе рассматриваются использование условий Swift, логической логики и трехмерных выражений.

замечания

Дополнительные сведения об условных операторах см. В разделе Быстрый язык программирования .

Examples

Использование гвардии

2,0

Guard проверяет состояние, и если оно ложно, оно входит в ветвь. Отделения проверки проверки должны оставить свой закрывающий блок либо через `return` , `break` или `continue` (если применимо); неспособность сделать это приводит к ошибке компилятора. Это имеет то преимущество, что, когда `guard` написан, невозможно, чтобы поток продолжался случайно (как это возможно при использовании `if`).

Использование охранников может помочь снизить уровень гнездования , что обычно улучшает читаемость кода.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard также может проверить, есть ли значение в опции , а затем разворачивать его во внешней области:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
```

```
    print(unwrappedNum)
}
```

Guard может объединить дополнительный развёрток и проверить состояние , используя , `where` ключевое слово:

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Основные условные обозначения: if-statements

Оператор **`if`** проверяет, является ли условие Bool `true` :

```
let num = 10

if num == 10 {
    // Code inside this block only executes if the condition was true.
    print("num is 10")
}

let condition = num == 10   // condition's type is Bool
if condition {
    print("num is 10")
}
```

`if` утверждения принимают **`else if`** и **`else`** блоки, которые могут проверять альтернативные условия и предоставлять резервную копию:

```
let num = 10
if num < 10 {  // Execute the following code if the first condition is true.
    print("num is less than 10")
} else if num == 10 {  // Or, if not, check the next condition...
    print("num is 10")
} else {  // If all else fails...
    print("all other conditions were false, so num is greater than 10")
}
```

Основные операторы типа `&&` и `||` может использоваться для нескольких условий:

Логический оператор И

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

Если `num == 10` было ложным, второе значение не будет оцениваться. Это называется оценкой короткого замыкания.

Логический оператор OR

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

Если `num == 10` истинно, второе значение не будет оцениваться.

Логический оператор NOT

```
if !str.isEmpty {
    print("str is not empty")
}
```

Необязательные обязательства и предложения «где»

Опционы должны быть *развернуты,* прежде чем их можно будет использовать в большинстве выражений. `if let` является *необязательной привязкой* , которая преуспевает, если необязательное значение **не** равно **`nil`** :

```
let num: Int? = 10 // or: let num: Int? = nil

if let unwrappedNum = num {
    // num has type Int?; unwrappedNum has type Int
    print("num was not nil: \(unwrappedNum + 1)")
} else {
    print("num was nil")
}
```

Вы можете повторно использовать **одно и** то **же имя** для новой связанной переменной, затеняя оригинал:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

1,2 3,0

Объедините несколько необязательных привязок с запятыми (,):

```
if let unwrappedNum = num, let unwrappedStr = str {
    // Do something with unwrappedNum & unwrappedStr
} else if let unwrappedNum = num {
    // Do something with unwrappedNum
} else {
```

```
    // num was nil
}
```

Примените дополнительные ограничения после необязательного связывания с помощью предложения `where` :

```
if let unwrappedNum = num where unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}
```

Если вы чувствуете себя авантюрно, чередуйте любое количество необязательных привязок и `where` :

```
if let num = num                            // num must be non-nil
    where num % 2 == 1,                     // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first    // str must also be non-empty
    where firstChar != "x"                  // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

3.0

В Swift 3, `where` были заменены предложения (SE-0099): просто используйте другое , чтобы отделить дополнительные привязки и логические условия.

```
if let unwrappedNum = num, unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}

if let num = num,                           // num must be non-nil
    num % 2 == 1,                           // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first,   // str must also be non-empty
    firstChar != "x"                        // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

Тернарный оператор

Условия могут также оцениваться в одной строке с использованием тернарного оператора:

Если вы хотите определить минимум и максимум две переменные, вы можете использовать операторы if, например:

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
```

```
    min = b
}

let max: Int

if a > b {
    max = a
} else {
    max = b
}
```

Тернарный условный оператор принимает условие и возвращает одно из двух значений, в зависимости от того, было ли условие истинным или ложным. Синтаксис выглядит следующим образом: это эквивалентно выражению:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

Вышеприведенный код можно переписать с помощью тернарного условного оператора, как показано ниже:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

В первом примере условие равно <b. Если это верно, результат, назначенный обратно в мин, будет равным; если оно ложно, результатом будет значение b.

Примечание. Поскольку поиск большего или меньшего из двух чисел является такой общей операцией, стандартная библиотека Swift предоставляет для этой цели две функции: max и min.

Nil-Coalescing Operator

Оператор nil-coalescing <OPTIONAL> ?? <DEFAULT VALUE> разворачивает <OPTIONAL> OPTIONAL <OPTIONAL> если оно содержит значение, или возвращает <DEFAULT VALUE> если nil. <OPTIONAL> всегда имеет необязательный тип. <DEFAULT VALUE> должен соответствовать типу, который хранится внутри <OPTIONAL> .

Оператор nil-coalescing является сокращением для кода ниже, который использует тернарный оператор:

```
a != nil ? a! : b
```

это можно проверить по приведенному ниже коду:

```
(a ?? b) == (a != nil ? a! : b) // ouputs true
```

Время для примера

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // ouputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // ouputs "Fast"
```

Прочитайте Conditionals онлайн: https://riptutorial.com/ru/swift/topic/475/conditionals

глава 4: Loops

Синтаксис

- для постоянной последовательности {операторов}
- для константы в последовательности, где условие {утверждения}
- для переменной var в последовательности {statements}
- для _ в последовательности {statements}
- для случая пусть константа в последовательности {statements}
- для случая пусть константа в последовательности, где условие {утверждения}
- для переменной var var в последовательности {statements}
- в то время как условие {statements}
- повторять {утверждения} при условии
- sequence.forEach (body: (Element) throws -> Void)

Examples

Входной контур

Цикл **for-in** позволяет выполнять итерацию по любой последовательности.

Итерирование в диапазоне

Вы можете перебирать как полуоткрытые, так и закрытые диапазоны:

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// Both print:
// 0
// 1
// 2
```

Итерация по массиву или установка

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
// Emily
// Miles
```

2.1 2.2

Если вам нужен индекс для каждого элемента в массиве, вы можете использовать метод enumerate() в SequenceType .

```
for (index, name) in names.enumerate() {
```

```
    print("The index of \(name) is \(index).")
}

// The index of James is 0.
// The index of Emily is 1.
// The index of Miles is 2.
```

enumerate() возвращает ленивую последовательность, содержащую пары элементов с последовательным Int s, начиная с 0. Поэтому с массивами эти числа будут соответствовать данному индексу каждого элемента, однако это может быть не так, как в случае с другими типами коллекций.

3.0

В Swift 3, enumerate() было переименовано в enumerated() :

```
for (index, name) in names.enumerated() {
    print("The index of \(name) is \(index).")
}
```

Итерация по словарю

```
let ages = ["James": 29, "Emily": 24]

for (name, age) in ages {
    print(name, "is", age, "years old.")
}

// Emily is 24 years old.
// James is 29 years old.
```

Итерация в обратном направлении

2.1 2.2

Вы можете использовать метод reverse() для SequenceType , чтобы перебирать любую последовательность в обратном порядке:

```
for i in (0..<3).reverse() {
    print(i)
}

for i in (0...2).reverse() {
    print(i)
}

// Both print:
// 2
// 1
// 0

let names = ["James", "Emily", "Miles"]

for name in names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

3.0

В Swift 3, reverse() был переименован в reverse() reversed() :

```
for i in (0..<3).reversed() {
    print(i)
}
```

Итерирование по диапазонам с пользовательским шагом

2.1 2.2

Используя методы stride(_:_:) на Strideable вы можете перебирать диапазон с помощью специального шага:

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}

// 4
// 2
// 0
```

1,2 3,0

В Swift 3 методы stride(_:_:) на Stridable были заменены глобальными stride(_:_:_:) :

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

Повторите цикл while

Подобно циклу while, после цикла вычисляется только оператор управления. Поэтому цикл будет выполняться как минимум один раз.

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

while loop

В while цикл будет выполняться до тех пор , пока условие истинно.

```
var count = 1

while count < 10 {
    print("This is the \(count) run of the loop")
    count += 1
}
```

Тип последовательности для каждого блока

Тип, который соответствует протоколу SequenceType, может выполнять итерацию через его элементы в закрытии:

```
collection.forEach { print($0) }
```

То же самое можно сделать и с именованным параметром:

```
collection.forEach { item in
    print(item)
}
```

* Примечание. Операторы потока управления (такие как break или continue) не могут использоваться в этих блоках. Возврат может быть вызван, и если он вызван, он немедленно вернет блок для текущей итерации (как и для продолжения). Затем будет выполнена следующая итерация.

```
let arr = [1,2,3,4]

arr.forEach {

    // blocks for 3 and 4 will still be called
    if $0 == 2 {
        return
    }
}
```

Цикл ввода-вывода с фильтрацией

1. **where** пункт

Добавляя предложение where , вы можете ограничить итерации теми, которые удовлетворяют данному условию.

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
}

// James
// Miles
```

2. **case** раздел

Это полезно, когда вам нужно итерации только через значения, которые соответствуют некоторому шаблону:

```
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("point on x-axis")
}

//point on x-axis
//point on x-axis
```

Также вы можете фильтровать необязательные значения и разворачивать их, если необходимо, путем добавления ? знак после константы привязки:

```
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
//5
```

Разрыв петли

Цикл будет выполняться до тех пор, пока его условие остается верным, но вы можете остановить его вручную, используя ключевое слово **break** . Например:

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak is the element located on position [\(positionOfNovak)] in peopleArray.")
//prints out: Novak is the element located on position 5 in peopleArray. (which is true)
```

Прочитайте Loops онлайн: https://riptutorial.com/ru/swift/topic/1186/loops

глава 5: NSRegularExpression в Swift

замечания

Специальные символы

```
   *?+[(){}^$|\./
```

Examples

Расширение строки для простого сопоставления шаблонов

```
extension String {
    func matchesPattern(pattern: String) -> Bool {
        do {
            let regex = try NSRegularExpression(pattern: pattern,
                                                options: NSRegularExpressionOptions(rawValue:
0))
            let range: NSRange = NSMakeRange(0, self.characters.count)
            let matches = regex.matchesInString(self, options: NSMatchingOptions(), range:
range)
            return matches.count > 0
        } catch _ {
            return false
        }
    }
}

// very basic examples - check for specific strings
dump("Pinkman".matchesPattern("(White|Pinkman|Goodman|Schrader|Fring)"))

// using character groups to check for similar-sounding impressionist painters
dump("Monet".matchesPattern("(M[oa]net)"))
dump("Manet".matchesPattern("(M[oa]net)"))
dump("Money".matchesPattern("(M[oa]net)"))     // false

// check surname is in list
dump("Skyler White".matchesPattern("\\w+ (White|Pinkman|Goodman|Schrader|Fring)"))

// check if string looks like a UK stock ticker
dump("VOD.L".matchesPattern("[A-Z]{2,3}\\.L"))
dump("BP.L".matchesPattern("[A-Z]{2,3}\\.L"))

// check entire string is printable ASCII characters
dump("tab\tformatted text".matchesPattern("^[\u{0020}-\u{007e}]*$"))

// Unicode example: check if string contains a playing card suit
dump("♠".matchesPattern("[\u{2660}-\u{2667}]"))
dump("♡".matchesPattern("[\u{2660}-\u{2667}]"))
dump("".matchesPattern("[\u{2660}-\u{2667}]"))    // false

// NOTE: regex needs Unicode-escaped characters
dump("♣".matchesPattern("♣"))           // does NOT work
```

Ниже приведен еще один пример, который опирается на приведенное выше, чтобы сделать что-то полезное, что нелегко сделать любым другим методом и хорошо подходит для решения регулярных выражений.

```
// Pattern validation for a UK postcode.
// This simply checks that the format looks like a valid UK postcode and should not fail on
false positives.
private func isPostcodeValid(postcode: String) -> Bool {
    return postcode.matchesPattern("^[A-Z]{1,2}([0-9][A-Z]|[0-9]{1,2})\\s[0-9][A-Z]{2}")
}

// valid patterns (from
https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Validation)
// will return true
dump(isPostcodeValid("EC1A 1BB"))
dump(isPostcodeValid("W1A 0AX"))
dump(isPostcodeValid("M1 1AE"))
dump(isPostcodeValid("B33 8TH"))
dump(isPostcodeValid("CR2 6XH"))
dump(isPostcodeValid("DN55 1PT"))

// some invalid patterns
// will return false
dump(isPostcodeValid("EC12A 1BB"))
dump(isPostcodeValid("CRB1 6XH"))
dump(isPostcodeValid("CR 6XH"))
```

Основное использование

При реализации регулярных выражений в Swift существует несколько соображений.

```
let letters = "abcdefg"
let pattern = "[a,b,c]"
let regEx = try NSRegularExpression(pattern: pattern, options: [])
let nsString = letters as NSString
let matches = regEx.matches(in: letters, options: [], range: NSMakeRange(0, nsString.length))
let output = matches.map {nsString.substring(with: $0.range)}
//output = ["a", "b", "c"]
```

Чтобы получить точную длину диапазона, которая поддерживает все типы символов, входная строка должна быть преобразована в NSString.

Для соответствия безопасности шаблону следует заключить в блок блокировки do для обработки отказа

```
let numbers = "121314"
let pattern = "1[2,3]"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
    output
} catch let error as NSError {
    print("Matching failed")
}
//output = ["12", "13"]
```

Функциональность регулярных выражений часто помещается в расширение или помощник для отдельных проблем.

Замена подстрок

Шаблоны могут использоваться для замены части входной строки.

Приведенный ниже пример заменяет символ cent символом доллара.

```
var money = "¢¥€£$¥€£¢"
let pattern = "¢"
do {
    let regEx = try NSRegularExpression (pattern: pattern, options: [])
    let nsString = money as NSString
    let range = NSMakeRange(0, nsString.length)
    let correct$ = regEx.stringByReplacingMatches(in: money, options: .withTransparentBounds,
range: range, withTemplate: "$")
} catch let error as NSError {
    print("Matching failed")
}
//correct$ = "$¥€£$¥€£$"
```

Специальные символы

Для соответствия специальным символам следует использовать Double Backslash \. becomes \\.

Персонажи, с которыми вам придется бежать, включают

```
(){}[]/\+*$>.|^?
```

В приведенном ниже примере приведены три вида открывающих скобок

```
let specials = "(){}[]"
let pattern = "(\\(|\\{|\\[)"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = specials as NSString
    let matches = regEx.matches(in: specials, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
} catch let error as NSError {
    print("Matching failed")
}
//output = ["(", "{", "["]
```

Проверка

Регулярные выражения могут использоваться для проверки входов путем подсчета количества совпадений.

```
var validDate = false

let numbers = "35/12/2016"
let usPattern =  "^(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\\d\\d$"
let ukPattern = "^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[012])[-/](19|20)\\d\\d$"
do {
    let regEx = try NSRegularExpression(pattern: ukPattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))

    if matches.count > 0 {
        validDate = true
```

```
    }

    validDate

} catch let error as NSError {
    print("Matching failed")
}
//output = false
```

NSRegularExpression для проверки почты

```
func isValidEmail(email: String) -> Bool {

    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: email)
}
```

или вы можете использовать расширение строки следующим образом:

```
extension String
{
    func isValidEmail() -> Bool {

        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

        let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
        return emailTest.evaluate(with: self)
    }
}
```

Прочитайте NSRegularExpression в Swift онлайн:
https://riptutorial.com/ru/swift/topic/5763/nsregularexpression-в-swift

глава 6: Optionals

Вступление

«Необязательное значение либо содержит значение, либо содержит nil, чтобы указать, что значение отсутствует»

Выдержка из: Apple Inc. «Быстрый язык программирования (издание Swift 3.1)». IBooks. https://itun.es/us/k5SW7.l

Основные необязательные варианты использования включают: для константы (let), использование необязательного в цикле (if-let), безопасное развертывание необязательного значения внутри метода (guard-let) и как часть циклов переключения (case-let), по умолчанию - значение, если nil, используя оператор coalesce (??)

Синтаксис

- var optionalName: optionalType? // объявлять необязательный тип, по умолчанию - nil
- var optionalName: optionalType? = значение // объявляет необязательный параметр со значением
- var optionalName: optionalType! // объявлять неявно развернутый необязательный
- необязательный! // принудительно разворачиваем опцию

замечания

Дополнительные сведения об опциях см. В разделе «Быстрый язык программирования» .

Examples

Типы опций

Опционы - это общий тип перечисления, который действует как обертка. Эта оболочка позволяет переменной иметь одно из двух состояний: значение пользовательского типа или nil , которое представляет отсутствие значения.

Эта способность особенно важна в Swift, потому что одна из заявленных целей дизайна языка - это хорошо работать с каркасами Apple. Многие (большинство) инфраструктур Apple используют nil из-за простоты использования и значимости для шаблонов программирования и дизайна API в Objective-C.

В Swift для переменной, имеющей значение nil , она должна быть необязательной. Опционы могут быть созданы путем добавления a ! или ? к типу переменной. Например, чтобы сделать Int необязательным, вы можете использовать

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

? optionals должны быть явно развернуты и должны использоваться, если вы не уверены, будет ли переменная иметь значение при ее доступе. Например, при преобразовании строки в Int результат является необязательным значением Int? , потому что nil будет возвращен, если строка не является допустимым числом

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

! опции автоматически распаковываются и должны использоваться *только* тогда, когда вы *уверены*, что переменная будет иметь значение при ее доступе. Например, глобальный UIButton! переменная,

которая инициализируется в viewDidLoad()

```
//myButton will not be accessed until viewDidLoad is called,
//so a ! optional can be used here
var myButton: UIButton!

override func viewDidLoad(){
    self.myButton = UIButton(frame: self.view.frame)
    self.myButton.backgroundColor = UIColor.redColor()
    self.view.addSubview(self.myButton)
}
```

Развертывание необязательного

Чтобы получить доступ к значению необязательного, его необходимо развернуть.

Вы можете *условно разворачивать* опцию, используя опциональную привязку и *разворот силы* а Необязательно с помощью ! оператор.

Условно отказоустойчиво спрашивает: «Имеет ли эта переменная ценность?» в то время как развертывание силы говорит: «Эта переменная имеет значение!».

Если вы принудительно разворачиваете переменную, которая равна nil , ваша программа будет *неожиданно обнаруживать нуль при развертывании необязательного* исключения и сбоя, поэтому вам следует тщательно подумать, если вы используете ! является целесообразным.

```
var text: String? = nil
var unwrapped: String = text! //crashes with "unexpectedly found nil while unwrapping an
Optional value"
```

Для безопасного развертывания вы можете использовать оператор if-let , который не будет генерировать исключение или сбой, если обернутое значение равно nil :

```
var number: Int?
if let unwrappedNumber = number {       // Has `number` been assigned a value?
    print("number: \(unwrappedNumber)") // Will not enter this line
} else {
    print("number was not assigned a value")
}
```

Или заявление охранника :

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

Обратите внимание, что область переменной unwrappedNumber находится внутри оператора if-let и вне guard блока.

Вы можете перехватывать множество опций, это в основном полезно в случаях, когда вашему коду требуется более чем переменная для правильной работы:

```
var firstName:String?
var lastName:String?

if let fn = firstName, let ln = lastName {
    print("\(fn) + \(ln)")//pay attention that the condition will be true only if both
```

```
optionals are not nil.
}
```

Обратите внимание, что все переменные должны быть развернуты, чтобы успешно пройти тест, иначе у вас не было бы способа определить, какие переменные были развернуты, а какие нет.

Вы можете связать условные операторы с помощью своих опций сразу же после их разворачивания. Это означает, что нет вложенных операторов if-else!

```
var firstName:String? = "Bob"
var myBool:Bool? = false

if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName is bob and myBool was false!")
}
```

Nil Coalescing Operator

Вы можете использовать *оператор* объединения *nil* для разворачивания значения, если он не равен нулю, иначе укажите другое значение:

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "Fallback String"
}
print(fallbackIfNil("Hi")) // Prints "Hi"
print(fallbackIfNil(nil)) // Prints "Fallback String"
```

Этот оператор способен к короткому замыканию , что означает, что если левый операнд не равен нулю, правый операнд не будет оцениваться:

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "a string"
let str = foo ?? someExpensiveComputation()
```

В этом примере, поскольку foo равен нулю, someExpensiveComputation() не будет вызываться.

Вы также можете объединить несколько операторов объединения nil:

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

В этом примере baz будет присвоено развернутое значение foo если оно не равно nil, иначе ему будет присвоено развернутое значение bar если оно не равно nil, иначе ему будет присвоено значение возврата.

Необязательная цепочка

Вы можете использовать Необязательный Chaining , чтобы вызвать метод , получить доступ к свойству или индексам как необязательный. Это делается путем размещения ? между данной необязательной переменной и данным элементом (метод, свойство или индекс).

```
struct Foo {
    func doSomething() {
        print("Hello World!")
    }
```

```
}

var foo : Foo? = Foo()

foo?.doSomething() // prints "Hello World!" as foo is non-nil
```

Если foo содержит значение, на doSomething() будет вызываться doSomething() . Если foo равно nil , тогда ничего плохого не произойдет - код просто терпит неудачу и продолжит выполнение.

```
var foo : Foo? = nil

foo?.doSomething() // will not be called as foo is nil
```

(Это похоже на поведение отправки сообщений в nil в Objective-C)

Причина, по которой «Факультативный цепочек» называется как таковой, заключается в том, что «опциональность» будет распространяться через членов, которые вы вызываете / получаете доступ. Это означает, что возвращаемые значения любых членов, используемых с опциональной цепочкой, будут необязательными, независимо от того, набираются ли они как необязательные или нет.

```
struct Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

Здесь foo?.bar возвращает Int? даже если bar является необязательным, поскольку сам foo является необязательным.

Когда опциональность распространяется, методы, возвращающие Void , вернут Void? при вызове с опциональной цепочкой. Это может быть полезно для определения того, был ли вызван метод или нет (и, следовательно, если опция имеет значение).

```
let foo : Foo? = Foo()

if foo?.doSomething() != nil {
    print("foo is non-nil, and doSomething() was called")
} else {
    print("foo is nil, therefore doSomething() wasn't called")
}
```

Здесь мы сравниваем Void? возвращаемое значение с помощью nil , чтобы определить, был ли вызван метод (и, следовательно, foo равен нулю).

Обзор - Почему варианты?

Часто при программировании необходимо провести некоторое различие между переменной, которая имеет значение, а другая - нет. Для ссылочных типов, таких как C-указатели, специальное значение, такое как null может использоваться для указания того, что переменная не имеет значения. Для внутренних типов, таких как целое число, это сложнее. Номинальное значение, такое как -1, может использоваться, но это зависит от интерпретации значения. Он также устраняет это «особое» значение от нормального использования.

Чтобы решить эту проблему, Swift позволяет объявлять любую переменную как необязательную. Об этом свидетельствует использование a? или же ! после типа (см. Типы опций)

Например,

```
var possiblyInt: Int?
```

объявляет переменную, которая может содержать или не содержать целочисленное значение.

Специальное значение nil указывает, что в настоящее время для этой переменной не назначено значение.

```
possiblyInt = 5      // PossiblyInt is now 5
possiblyInt = nil    // PossiblyInt is now unassigned
```

nil также может использоваться для проверки назначенного значения:

```
if possiblyInt != nil {
    print("possiblyInt has the value \(possiblyInt!)")
}
```

Обратите внимание на использование ! в инструкции печати, чтобы *развернуть* необязательное значение.

В качестве примера общего использования опций используйте функцию, возвращающую целое число из строки, содержащей цифры; Возможно, что строка может содержать нецифровые символы или даже быть пустым.

Как функция, возвращающая простой Int указывает на сбой? Он не может этого сделать, возвращая какое-либо конкретное значение, поскольку это исключает возможность анализа этого значения из строки.

```
var someInt
someInt = parseInt("not an integer") // How would this function indicate failure?
```

Однако в Swift эта функция может просто вернуть *необязательный* Int. Тогда отказ указывается возвращаемым значением nil .

```
var someInt?
someInt = parseInt("not an integer")  // This function returns nil if parsing fails
if someInt == nil {
    print("That isn't a valid integer")
}
```

Прочитайте Optionals онлайн: https://riptutorial.com/ru/swift/topic/247/optionals

глава 7: OptionSet

Examples

Протокол OptionSet

OptionSetType - это протокол, предназначенный для представления типов бит-масок, где отдельные биты представляют собой элементы набора. Набор логических и / или функций обеспечивает правильный синтаксис:

```
struct Features : OptionSet {
  let rawValue : Int
  static let none = Features(rawValue: 0)
  static let feature0 = Features(rawValue: 1 << 0)
  static let feature1 = Features(rawValue: 1 << 1)
  static let feature2 = Features(rawValue: 1 << 2)
  static let feature3 = Features(rawValue: 1 << 3)
  static let feature4 = Features(rawValue: 1 << 4)
  static let feature5 = Features(rawValue: 1 << 5)
  static let all: Features = [feature0, feature1, feature2, feature3, feature4, feature5]
}

Features.feature1.rawValue //2
Features.all.rawValue //63

var options: Features = [.feature1, .feature2, .feature3]

options.contains(.feature1) //true
options.contains(.feature4) //false

options.insert(.feature4)
options.contains(.feature4) //true


var otherOptions : Features = [.feature1, .feature5]

options.contains(.feature5) //false

options.formUnion(otherOptions)
options.contains(.feature5) //true

options.remove(.feature5)
options.contains(.feature5) //false
```

Прочитайте OptionSet онлайн: https://riptutorial.com/ru/swift/topic/1242/optionset

глава 8: RxSwift

Examples

Основы RxSwift

FRP или функционально-реактивное программирование, имеет некоторые основные термины, которые вам нужно знать.

Каждая часть данных может быть представлена как Observable , которая представляет собой асинхронный поток данных. Сила FRP заключается в представлении синхронных и асинхронных событий как потоков, Observable s и предоставления одного и того же интерфейса для работы с ним.

Обычно Observable содержит несколько (или ни одного) событий, содержащих даты - .Next события, а затем он может быть успешно завершен (.Success) или с ошибкой (.Error).

Давайте посмотрим на следующую мраморную диаграмму:

```
--(1)--(2)--(3)|-->
```

В этом примере есть поток значений Int . По .Next происходит три .Next события, а затем поток завершается успешно.

```
--X->
```

Выше диаграмма показывает случай , где не было никаких данных , излучаемого и .Error событие завершает Observable .

Прежде чем двигаться дальше, есть некоторые полезные ресурсы:

1. RxSwift . Посмотрите примеры, прочитайте документы и начните работу.
2. В комнате RxSwift Slack есть несколько каналов для решения проблем образования.
3. Играйте с RxMarbles, чтобы узнать, что делает оператор, и что наиболее полезно в вашем случае.
4. Взгляните на этот пример , исследуйте код самостоятельно.

Создание наблюдаемых

RxSwift предлагает множество способов создания Observable , давайте посмотрим:

```
import RxSwift

let intObservale = Observable.just(123) // Observable<Int>
let stringObservale = Observable.just("RxSwift") // Observable<String>
let doubleObservale = Observable.just(3.14) // Observable<Double>
```

Итак, наблюдаемые создаются. Они имеют только одно значение, а затем завершаются с успехом. Тем не менее, ничего не происходило после его создания. Зачем?

Существует два шага в работе с Observable s: вы **наблюдаете** что-то для *создания* потока, а затем **подписываетесь** на поток или **связываете** его с чем-то, чтобы *взаимодействовать* с ним.

```
Observable.just(12).subscribe {
    print($0)
}
```

Консоль будет печатать:

```
.Next(12)
.Completed()
```

И если мне интересно только работать с данными, которые происходят в .Next событиях, я бы использовал оператор subscribeNext :

```
Observable.just(12).subscribeNext {
    print($0) // prints "12" now
}
```

Если я хочу создать наблюдаемое множество значений, я использую разные операторы:

```
Observable.of(1,2,3,4,5).subscribeNext {
    print($0)
}
// 1
// 2
// 3
// 4
// 5

// I can represent existing data types as Observables also:
[1,2,3,4,5].asObservable().subscribeNext {
    print($0)
}
// result is the same as before.
```

И, наконец, возможно, я хочу, чтобы Observable какую-то работу. Например, удобно Observable<SomeResultType> сетевую операцию в Observable<SomeResultType> . Давайте посмотрим, как можно добиться этого:

```
Observable.create { observer in    // create an Observable ...
    MyNetworkService.doSomeWorkWithCompletion { (result, error) in
        if let e = error {
            observer.onError(e)    // ..that either holds an error
        } else {
            observer.onNext(result) // ..or emits the data
            observer.onCompleted()  // ..and terminates successfully.
        }
    }
    return NopDisposable.instance // here you can manually free any resources
                                  //in case if this observable is being disposed.
}
```

Располагая

После того, как была создана подписка, важно управлять ее правильным освобождением.

Документы сказали нам, что

> Если последовательность завершается за конечное время, не вызывая распоряжаться или не использовать addDisposableTo (disposeBag), это не вызовет утечки постоянного ресурса. Тем не менее, эти ресурсы будут использоваться до тех пор, пока последовательность не завершится, либо завершив производство элементов, либо вернув ошибку.

Существует два способа освобождения ресурсов.

1. Использование disposeBag s и addDisposableTo .
2. Использование оператора takeUntil .

В первом случае вы вручную передаете подписку на объект DisposeBag , который правильно очищает всю DisposeBag память.

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

Вам фактически не нужно создавать DisposeBag s в каждом DisposeBag классе, просто взгляните на *проект сообщества RxSwift* с именем NSObject + Rx . Используя структуру, приведенный выше код можно переписать следующим образом:

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

Во втором случае, если время подписки совпадает с self жизнью объекта, можно осуществить с помощью утилизации takeUntil(rx_deallocated) :

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

Наручники

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "Greetings, \($0)" }
.bindTo(greetingLabel.rx_text)
```

Используя оператор combineLatest каждый раз, когда элемент испускается одним из двух Observables объектов, объедините последний элемент, испускаемый каждым Observable . Таким образом, мы комбинируем результат двух UITextField , создающих новое сообщение с текстом "Greetings, \($0)" используя строчную интерполяцию, чтобы позже связать текст UILabel .

Мы можем привязать данные к любым UITableView и UICollectionView очень простым способом:

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell) in
    cell.title = viewModel.title
    cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

Это оболочка Rx вокруг cellForRowAtIndexPath источника данных cellForRowAtIndexPath . А также Rx заботится о реализации numberOfRowsAtIndexPath , который является необходимым методом в традиционном смысле, но вам не нужно его реализовывать здесь, это позаботится.

RxCocoa и ControlEvents

RxSwift предоставляет не только способы управления вашими данными, но и представление действий пользователя в реактивном режиме.

RxCocoa содержит все, что вам нужно. Он переносит большинство свойств компонентов пользовательского интерфейса в Observable s, но не совсем. Есть некоторые обновленные Observable

s под названием ControlEvent s (которые представляют события) и ControlProperties (которые представляют свойства, удивление!). Эти вещи содержат Observable потоки под капотом, но также имеют некоторые нюансы:

- Это никогда не терпит неудачу, поэтому ошибок нет.
- Он будет Complete последовательность при освобождении элемента управления.
- Он поставляет события в основной поток (MainScheduler.instance).

В принципе, вы можете работать с ними, как обычно:

```
button.rx_tap.subscribeNext { _ in   // control event
    print("User tapped the button!")
}.addDisposableTo(bag)

textField.rx_text.subscribeNext { text in // control property
    print("The textfield contains: \(text)")
}.addDisposableTo(bag)
// notice that ControlProperty generates .Next event on subscription
// In this case, the log will display
// "The textfield contains: "
// at the very start of the app.
```

Это очень важно для использования: пока вы используете Rx, забудьте о вещах @IBAction , все, что вам нужно, вы можете сразу привязать и настроить. Например, метод viewDidLoad вашего контроллера представления является хорошим кандидатом для описания того, как работают пользовательские интерфейсы.

Хорошо, еще один пример: предположим, что у нас есть текстовое поле, кнопка и ярлык. Мы хотим **проверить текст** в текстовом поле, когда мы **нажимаем** кнопку, и **отображаем** результаты в метке. Да, похоже, еще одна задача проверки подлинности электронной почты, да?

Прежде всего, мы button.rx_tap ControlEvent:

```
----()-----------------------()----->
```

Здесь пустые круглые скобки показывают пользовательские краны. Затем мы берем то, что написано в текстовом поле с withLatestFrom операторaLatestFrom (посмотрите на него здесь , представьте, что верхний поток представляет собой отладки пользователя, нижний - текст в текстовом поле).

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")--------------------("123")--->
//  ^ tap   ^ i wrote 123    ^ tap
```

Приятно, у нас есть поток строк для проверки, испускаемый только тогда, когда нам нужно проверить.

В любом Observable есть такие знакомые операторы, как map или filter , мы берем map для проверки текста. Создайте функцию validateEmail самостоятельно, используйте любое регулярное выражение, которое вы хотите.

```
button.rx_tap                                 // ControlEvent<Void>
        .withLatestFrom(textField.rx_text)    // Observable<String>
        .map(validateEmail)                   // Observable<Bool>
        .map { (isCorrect) in
            return isCorrect ? "Email is correct" : "Input the correct one, please"
        }                                     // Observable<String>
        .bindTo(label.rx_text)
        .addDisposableTo(bag)
```

Готово! Если вам нужна дополнительная пользовательская логика (например, отображение ошибок в

случае ошибки, переход на другой экран с успехом ...), просто подпишитесь на последний поток Bool и напишите там.

Прочитайте RxSwift онлайн: https://riptutorial.com/ru/swift/topic/4890/rxswift

глава 9: Typealias

Examples

титалии для замыканий с параметрами

```
typealias SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

Этот блок кода создает псевдоним типа с именем SuccessHandler , только в том же образом var string = "" создает переменную с именем string .

Теперь, когда вы используете SuccessHandler , например:

```
func example(_ handler: SuccessHandler) {}
```

Вы, как правило, пишете:

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

титалии для пустых закрытий

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

Этот блок создает псевдоним типа, который работает как функция Void to Void (не принимает никаких параметров и ничего не возвращает).

Вот пример использования:

```
var func: Handler?

func = {}
```

типы для других типов

```
typealias Number = NSNumber
```

Вы также можете использовать псевдоним типа, чтобы указать тип другого имени, чтобы было легче запомнить его, или сделать ваш код более элегантным.

типы для кортежей

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

И это можно использовать как:

```
func getPerson(for name: String) -> PersonTuple {
    //fetch from db, etc
    return ("name", 45, "address")
}
```

Прочитайте Typealias онлайн: https://riptutorial.com/ru/swift/topic/7552/typealias

Вступление

Алгоритмы являются основой для вычислений. Выбор того, какой алгоритм использовать, в какой ситуации отличает среднее от хорошего программиста. Имея это в виду, здесь приведены определения и примеры кода некоторых из основных алгоритмов.

Examples

Вставка Сортировка

Сортировка вставки - один из наиболее простых алгоритмов в информатике. Вставка сортирует элементы ранжирования путем итерации через элементы коллекции и позиции на основе их значения. Набор делится на отсортированные и несортированные половинки и повторяется до тех пор, пока все элементы не будут отсортированы. Сортировка вставки имеет сложность O (n2). Вы можете поместить его в расширение, как в примере ниже, или вы можете создать для него метод.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Сортировка

Сортировка пузырьков

Это простой алгоритм сортировки, который многократно проходит через отсортированный список, сравнивает каждую пару соседних элементов и свопирует их, если они находятся в неправильном порядке. Проход по списку повторяется до тех пор, пока не потребуются свопы. Хотя алгоритм прост, он слишком медленный и непрактичный для большинства проблем. Он имеет сложность O (n2), но считается медленнее, чем сортировка вставки.

```
extension Array where Element: Comparable {

func bubbleSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryIndex in 0..<self.count {
        let passes = (output.count - 1) - primaryIndex

        //"half-open" range operator
        for secondaryIndex in 0..<passes {
            let key = output[secondaryIndex]

            //compare / swap positions
            if (key > output[secondaryIndex + 1]) {
                swap(&output[secondaryIndex], &output[secondaryIndex + 1])
            }
        }
    }

    return output
}

}
```

Сортировка вставки

Сортировка вставки - один из наиболее простых алгоритмов в информатике. Вставка сортирует элементы ранжирования путем итерации через элементы коллекции и позиции на основе их значения. Набор делится на отсортированные и несортированные половинки и повторяется до тех пор, пока все элементы не будут отсортированы. Сортировка вставки имеет сложность O (n2). Вы можете поместить его в расширение, как в примере ниже, или вы можете создать для него метод.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
```

```
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Выбор сортировки

Сорт выбора отличается простотой. Он начинается с первого элемента массива, сохраняя его как минимальное значение (или максимум, в зависимости от порядка сортировки). Затем он повторяется через массив и заменяет значение min любым другим значением, меньшим, чем min, которое оно находит в пути. Это минимальное значение затем помещается в крайнюю левую часть массива, и процесс повторяется из следующего индекса до конца массива. Сортировка выбора имеет сложность O (n2), но считается медленнее, чем ее аналог. Сортировка сортировки.

func selectionSort () -> Array {// проверяем тривиальный защитный футляр self.count> 1 else {return self}

```
//mutated copy
var output: Array<Element> = self

for primaryindex in 0..<output.count {
    var minimum = primaryindex
    var secondaryindex = primaryindex + 1

    while secondaryindex < output.count {
        //store lowest value as minimum
        if output[minimum] > output[secondaryindex] {
            minimum = secondaryindex
        }
        secondaryindex += 1
    }

    //swap minimum value with array iteration
    if primaryindex != minimum {
        swap(&output[primaryindex], &output[minimum])
    }
}

return output
}
```

Быстрое Сортировка – O (n log n) время сложности

Quicksort – один из передовых алгоритмов. Он имеет временную сложность O (n log n) и применяет стратегию разделения и завоевания. Эта комбинация приводит к передовой алгоритмической работе. Quicksort сначала делит большой массив на два меньших подмассива: низкие элементы и высокие элементы. Quicksort может затем рекурсивно сортировать подмассивы.

Шаги:

Выберите элемент, называемый стержнем, из массива.

Измените порядок массива так, чтобы все элементы со значениями, меньшими, чем точка поворота, приходили перед точкой поворота, тогда как все элементы со значениями больше, чем точка поворота, после него (равные значения могут идти в любом случае). После этого разбиения стержень находится в своем последнем положении. Это называется операцией раздела.

Рекурсивно применяйте вышеуказанные шаги к подматрицу элементов с меньшими значениями и отдельно

к подматрицу элементов с более высокими значениями.

mutating func quickSort () -> Array {

```
func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

Выбор сортировки

Сорт выбора отличается простотой. Он начинается с первого элемента массива, сохраняя его как минимальное значение (или максимум, в зависимости от порядка сортировки). Затем он повторяется через массив и заменяет значение min любым другим значением, меньшим, чем min, которое оно находит в пути. Это минимальное значение затем помещается в крайнюю левую часть массива, и процесс повторяется из следующего индекса до конца массива. Сортировка выбора имеет сложность O (n2), но считается медленнее, чем ее аналог. Сортировка сортировки.

```
func selectionSort() -> Array<Element> {
    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {
        var minimum = primaryindex
        var secondaryindex = primaryindex + 1
```

```
        while secondaryindex < output.count {
            //store lowest value as minimum
            if output[minimum] > output[secondaryindex] {
                minimum = secondaryindex
            }
            secondaryindex += 1
        }

        //swap minimum value with array iteration
        if primaryindex != minimum {
            swap(&output[primaryindex], &output[minimum])
        }
    }

    return output
}
```

Асимптотический анализ

Поскольку у нас есть много разных алгоритмов на выбор, когда мы хотим сортировать массив, нам нужно знать, какой из них сделает это. Поэтому нам нужен какой-то метод измерения скорости и надежности алгоритма. Именно здесь начинается асимптотический анализ. Асимптотический анализ - это процесс описания эффективности алгоритмов по мере роста их входного размера (n). В информатике асимптотика обычно выражается в общем формате, известном как Big O Notation.

- **Линейное время O (n)** : когда каждый элемент массива должен оцениваться, чтобы функция достигла цели, это означает, что функция становится менее эффективной по мере увеличения количества элементов. *Говорят, что такая функция работает в линейном времени, потому что ее скорость зависит от ее размера ввода.*
- **Полиномиальное время O (n2)** : если сложность функции растет экспоненциально (это означает, что для n элементов массива сложность функции равна n квадрату), эта функция работает в полиномиальном времени. Обычно это функции с вложенными циклами. Два вложенных цикла приводят к сложности O (n2), три вложенных цикла приводят к сложности O (n3) и так далее ...
- **Логарифмическое время O (log n)**: сложность логарифмических временных функций сводится к минимуму при увеличении размера входных данных (n). Это тип функций, к которым стремится каждый программист.

Быстрое Сортировка - O (n log n) время сложности

Quicksort - один из передовых алгоритмов. Он имеет временную сложность O (n log n) и применяет стратегию разделения и завоевания. Эта комбинация приводит к передовой алгоритмической работе. Quicksort сначала делит большой массив на два меньших подмассива: низкие элементы и высокие элементы. Quicksort может затем рекурсивно сортировать подмассивы.

Шаги:

1. Выберите элемент, называемый стержнем, из массива.
2. Измените порядок массива так, чтобы все элементы со значениями, меньшими, чем точка поворота, приходили перед точкой поворота, тогда как все элементы со значениями больше, чем точка поворота, после него (равные значения могут идти в любом случае). После этого разбиения стержень находится в своем последнем положении. Это называется операцией раздела.
3. Рекурсивно применяйте вышеуказанные шаги к подматрицу элементов с меньшими значениями и отдельно к подматрицу элементов с более высокими значениями.

```
mutating func quickSort() -> Array<Element> {

func qSort(start startIndex: Int, _ pivot: Int) {
```

```
    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
```

}

```
mutating func qPartition (start startIndex: Int, _ pivot: Int) -> Int {
```

```
var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
```

```
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

График, Trie, Stack

график

В информатике граф представляет собой абстрактный тип данных, который предназначен для реализации неориентированного графика и ориентированных графических концепций из математики. Структура данных графа состоит из конечного (и, возможно, изменяемого) множества вершин или узлов или точек вместе с набором неупорядоченных пар этих вершин для неориентированного графа или набора упорядоченных пар для ориентированного графа. Эти пары известны как ребра, дуги или линии для неориентированного графа и как стрелки, направленные ребра, направленные дуги или направленные линии для ориентированного графа. Вершины могут быть частью структуры графа или могут быть внешними объектами, представленными целыми индексами или ссылками. Структура данных графа также может ассоциировать с каждым ребром некоторое значение края, такое как символическая метка или числовой атрибут (стоимость, емкость, длина и т. Д.). (Википедия, источник)

```
//
//  GraphFactory.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 6/7/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation
```

```
public class SwiftGraph {


    //declare a default directed graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool


    init() {
        canvas = Array<Vertex>()
        isDirected = true
    }


    //create a new vertex
    func addVertex(key: String) -> Vertex {


        //set the key
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //add the vertex to the graph canvas
        canvas.append(childVertex)


        return childVertex
    }



    //add edge to source vertex
    func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {


        //create a new edge
        let newEdge = Edge()


        //establish the default properties
        newEdge.neighbor = neighbor
        newEdge.weight = weight
        source.neighbors.append(newEdge)


        print("The neighbor of vertex: \(source.key as String!) is \(neighbor.key as
String!)..")


        //check condition for an undirected graph
        if isDirected == false {


           //create a new reversed edge
           let reverseEdge = Edge()


           //establish the reversed properties
```

```
            reverseEdge.neighbor = source
            reverseEdge.weight = weight
            neighbor.neighbors.append(reverseEdge)

            print("The neighbor of vertex: \(neighbor.key as String!) is \(source.key as
String!)..")

        }


    }




    /* reverse the sequence of paths given the shortest path.
       process analagous to reversing a linked list. */

    func reversePath(_ head: Path!, source: Vertex) -> Path! {


        guard head != nil else {
            return head
        }

        //mutated copy
        var output = head


        var current: Path! = output
        var prev: Path!
        var next: Path!


        while(current != nil) {
            next = current.previous
            current.previous = prev
            prev = current
            current = next
        }


        //append the source path to the sequence
        let sourcePath: Path = Path()

        sourcePath.destination = source
        sourcePath.previous = prev
        sourcePath.total = nil

        output = sourcePath


        return output

    }




    //process Dijkstra's shortest path algorthim
```

```
func processDijkstra(_ source: Vertex, destination: Vertex) -> Path? {


    var frontier: Array<Path> = Array<Path>()
    var finalPaths: Array<Path> = Array<Path>()


    //use source edges to create the frontier
    for e in source.neighbors {

        let newPath: Path = Path()


        newPath.destination = e.neighbor
        newPath.previous = nil
        newPath.total = e.weight


        //add the new path to the frontier
        frontier.append(newPath)

    }


    //construct the best path
    var bestPath: Path = Path()


    while frontier.count != 0 {

        //support path changes using the greedy approach
        bestPath = Path()
        var pathIndex: Int = 0


        for x in 0..<frontier.count {

            let itemPath: Path = frontier[x]

            if  (bestPath.total == nil) || (itemPath.total < bestPath.total) {
                bestPath = itemPath
                pathIndex = x
            }

        }



        //enumerate the bestPath edges
        for e in bestPath.destination.neighbors {

            let newPath: Path = Path()

            newPath.destination = e.neighbor
            newPath.previous = bestPath
            newPath.total = bestPath.total + e.weight


            //add the new path to the frontier
            frontier.append(newPath)
```

```
            }


            //preserve the bestPath
            finalPaths.append(bestPath)


            //remove the bestPath from the frontier
            //frontier.removeAtIndex(pathIndex) - Swift2
            frontier.remove(at: pathIndex)



        } //end while



        //establish the shortest path as an optional
        var shortestPath: Path! = Path()


        for itemPath in finalPaths {

            if (itemPath.destination.key == destination.key) {

                if  (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
                    shortestPath = itemPath
                }

            }

        }


        return shortestPath

    }



    ///an optimized version of Dijkstra's shortest path algorthim
    func processDijkstraWithHeap(_ source: Vertex, destination: Vertex) -> Path! {


        let frontier: PathHeap = PathHeap()
        let finalPaths: PathHeap = PathHeap()


        //use source edges to create the frontier
        for e in source.neighbors {

            let newPath: Path = Path()


            newPath.destination = e.neighbor
            newPath.previous = nil
            newPath.total = e.weight


            //add the new path to the frontier
            frontier.enQueue(newPath)
```

```
        }


        //construct the best path
        var bestPath: Path = Path()


        while frontier.count != 0 {

            //use the greedy approach to obtain the best path
            bestPath = Path()
            bestPath = frontier.peek()


            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: Path = Path()

                newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //add the new path to the frontier
                frontier.enQueue(newPath)

            }


            //preserve the bestPaths that match destination
            if (bestPath.destination.key == destination.key) {
                finalPaths.enQueue(bestPath)
            }


            //remove the bestPath from the frontier
            frontier.deQueue()


        } //end while



        //obtain the shortest path from the heap
        var shortestPath: Path! = Path()
        shortestPath = finalPaths.peek()


        return shortestPath

    }


    //MARK: traversal algorithms


    //bfs traversal with inout closure function
    func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {
```

```
    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        var vitem: Vertex = graphQueue.deQueue() as Vertex!


        //add unvisited vertices to the queue
        for e in vitem.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        /*
        notes: this demonstrates how to invoke a closure with an inout parameter.
        By passing by reference no return value is required.
        */

        //invoke formula
        formula(&vitem)


    } //end while


    print("graph traversal complete..")


}




//breadth first search
func traverse(_ startingv: Vertex) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!
```

```
        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        vitem!.visited = true
        print("traversed vertex: \(vitem!.key!)..")


    } //end while


    print("graph traversal complete..")


} //end function



//use bfs with trailing closure to update all values
func update(startingv: Vertex, formula:((Vertex) -> Bool)) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        //apply formula..
        if formula(vitem!) == false {
            print("formula unable to update: \(vitem!.key)")
        }
```

```
            else {
                print("traversed vertex: \(vitem!.key!)..")
            }

            vitem!.visited = true


        } //end while


        print("graph traversal complete..")


    }




}
```

Trie

В информатике trie, также называемое цифровым деревом, а иногда и деревом radix или деревом префикса (по мере их поиска по префиксам), является своего рода деревом поиска - упорядоченной структурой данных дерева, которая используется для хранения динамического набора или ассоциативного массив, где ключи обычно являются строками. (Википедия, источник)

```
//
//  Trie.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 10/14/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class Trie {

    private var root: TrieNode!


    init(){
        root = TrieNode()
    }



    //builds a tree hierarchy of dictionary content
    func append(word keyword: String) {


        //trivial case
        guard keyword.length > 0 else {
            return
        }


        var current: TrieNode = root
```

```
    while keyword.length != current.level {

        var childToUse: TrieNode!
        let searchKey = keyword.substring(to: current.level + 1)


        //print("current has \(current.children.count) children..")


        //iterate through child nodes
        for child in current.children {

            if (child.key == searchKey) {
                childToUse = child
                break
            }

        }


        //new node
        if childToUse == nil {

            childToUse = TrieNode()
            childToUse.key = searchKey
            childToUse.level = current.level + 1
            current.children.append(childToUse)
        }


        current = childToUse


    } //end while


    //final end of word check
    if (keyword.length == current.level) {
        current.isFinal = true
        print("end of word reached!")
        return
    }



} //end function




//find words based on the prefix
func search(forWord keyword: String) -> Array<String>! {


    //trivial case
    guard keyword.length > 0 else {
        return nil
    }
```

```
        var current: TrieNode = root
        var wordList = Array<String>()


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)


            //print("looking for prefix: \(searchKey)..")


            //iterate through any child nodes
            for child in current.children {

                if (child.key == searchKey) {
                    childToUse = child
                    current = childToUse
                    break
                }

            }


            if childToUse == nil {
               return nil
            }


        } //end while



        //retrieve the keyword and any descendants
        if ((current.key == keyword) && (current.isFinal)) {
            wordList.append(current.key)
        }


        //include only children that are words
        for child in current.children {

            if (child.isFinal == true) {
                wordList.append(child.key)
            }

        }


        return wordList


    } //end function


}
```

(GitHub, источник)

стек

В информатике стек представляет собой абстрактный тип данных, который служит в качестве набора элементов с двумя основными операциями: push, который добавляет элемент в коллекцию и pop, который удаляет последний добавленный элемент, который еще не был удален. Порядок, в котором элементы выходят из стека, приводит к его альтернативному имени LIFO (для последнего, первого выхода). Кроме того, операция peek может дать доступ к вершине без изменения стека. (Википедия, источник)

См. Информацию о лицензии ниже и исходный источник кода в (github)

```
//
//  Stack.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 8/1/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


class Stack<T> {

    private var top: Node<T>

    init() {
        top = Node<T>()
    }


    //the number of items - O(n)
    var count: Int {


        //return trivial case
        guard top.key != nil else {
          return 0
        }


        var current = top
        var x: Int = 1


        //cycle through list
        while current.next != nil {
            current = current.next!
            x += 1
        }

        return x

    }


    //add item to the stack
    func push(withKey key: T) {


        //return trivial case
        guard top.key != nil else {
            top.key = key
            return
        }
```

```
        //create new item
        let childToUse = Node<T>()
        childToUse.key = key


        //set new created item at top
        childToUse.next = top
        top = childToUse

    }


    //remove item from the stack
    func pop() {

        if self.count > 1 {
            top = top.next
        }
        else {
            top.key = nil
        }

    }


    //retrieve the top most item
    func peek() -> T! {


        //determine instance
        if let topitem = top.key {
            return topitem
        }

        else {
            return nil
        }

    }



    //check for value
    func isEmpty() -> Bool {

        if self.count == 0 {
            return true
        }

        else {
            return false
        }

    }


}
```

Лицензия MIT (MIT)

Прочитайте Алгоритмы с Swift онлайн: https://riptutorial.com/ru/swift/topic/9116/алгоритмы-с-swift

глава 11: Блоки

Вступление

Из Swift Documentarion

Говорят, что замыкание выходит из функции, когда замыкание передается как аргумент функции, но вызывается после возвращения функции. Когда вы объявляете функцию, которая принимает замыкание как один из ее параметров, вы можете написать @escaping перед типом параметра, чтобы указать, что закрытие разрешено.

Examples

Неэкранирующее закрытие

В Swift 1 и 2 по умолчанию закрываются параметры закрытия. Если вы знали, что ваше закрытие не ускользнет от тела функции, вы можете пометить параметр атрибутом @noescape.

В Swift 3 это наоборот: параметры закрытия по умолчанию не экранируются. Если вы намерены избежать этой функции, вы должны пометить ее атрибутом @escaping.

```
class ClassOne {
  // @noescape is applied here as default
  func methodOne(completion: () -> Void) {
    //
  }
}

class ClassTwo {
  let obj = ClassOne()
  var greeting = "Hello, World!"

  func methodTwo() {
    obj.methodOne() {
      // self.greeting is required
      print(greeting)
    }
  }
}
```

Закрытие крышки

Из Swift Documentarion

> @escaping
>
> Примените этот атрибут к типу параметра в объявлении метода или функции, чтобы указать, что значение параметра может быть сохранено для последующего выполнения. Это означает, что значение позволяет пережить время жизни вызова. Параметры типа функции с атрибутом типа escaping требуют явного использования self. для свойств или методов.

```
class ClassThree {

    var closure: (() -> ())?

    func doSomething(completion: @escaping () -> ()) {
        closure = finishBlock
    }
}
```

В приведенном выше примере блок завершения сохраняется в закрытии и будет буквально жить вне вызова функции. Таким образом, complier заставит отмечать блок завершения как @ escaping.

Прочитайте Блоки онлайн: https://riptutorial.com/ru/swift/topic/8623/блоки

глава 12: Булевы

Examples

Что такое Bool?

Bool - это логический тип с двумя возможными значениями: true и false .

```
let aTrueBool = true
let aFalseBool = false
```

Bools используются в операциях управления потоком как условия. Оператор if использует логическое условие для определения того, какой блок кода запускать:

```
func test(_ someBoolean: Bool) {
    if someBoolean {
        print("IT'S TRUE!")
    }
    else {
        print("IT'S FALSE!")
    }
}
test(aTrueBool)  // prints "IT'S TRUE!"
```

Отмените Bool с помощью префикса! оператор

Префикс ! оператор возвращает **логическое отрицание** своего аргумента. То есть, !true возвращает false , а !false возвращает true .

```
print(!true)  // prints "false"
print(!false) // prints "true"

func test(_ someBoolean: Bool) {
    if !someBoolean {
        print("someBoolean is false")
    }
}
```

Логические логические операторы

Оператор OR (||) возвращает true, если один из двух операндов имеет значение true, иначе он возвращает false. Например, следующий код оценивается как истинный, поскольку по крайней мере одно из выражений с обеих сторон оператора OR равно true:

```
if (10 < 20) || (20 < 10) {
    print("Expression is true")
}
```

Оператор AND (&&) возвращает true, только если оба операнда оцениваются как истинные. Следующий пример вернет false, потому что только одно из двух выражений операнда принимает значение true:

```
if (10 < 20) && (20 < 10) {
    print("Expression is true")
}
```

Оператор XOR (^) возвращает true, если один и только один из двух операндов оценивается как

true. Например, следующий код вернет true, поскольку только один оператор имеет значение true:

```
if (10 < 20) ^ (20 < 10) {
    print("Expression is true")
}
```

Булевы и встроенные условия

Чистый способ обработки булевых строк - использовать inline-условие с a? b: c тройной оператор, который является частью основных операций Swift.

Введенное условие состоит из трех компонентов:

```
question ? answerIfTrue : answerIfFalse
```

где question - это логическое значение, которое вычисляется, и answerIfTrue - это значение, возвращаемое, если вопрос верен, а answerIfFalse - это значение, возвращаемое, если вопрос неверен.

Вышеприведенное выражение такое же, как:

```
if question {
    answerIfTrue
} else {
    answerIfFalse
}
```

С встроенными условными выражениями вы возвращаете значение на основе логического:

```
func isTurtle(_ value: Bool) {
    let color = value ? "green" : "red"
    print("The animal is \(color)")
}

isTurtle(true) // outputs 'The animal is green'
isTurtle(false) // outputs 'The animal is red'
```

Вы также можете вызывать методы на основе логического значения:

```
func actionDark() {
    print("Welcome to the dark side")
}

func actionJedi() {
    print("Welcome to the Jedi order")
}

func welcome(_ isJedi: Bool) {
    isJedi ? actionJedi() : actionDark()
}

welcome(true) // outputs 'Welcome to the Jedi order'
welcome(false) // outputs 'Welcome to the dark side'
```

Внутренние условные обозначения позволяют выполнять чистое однострочное вычисление булевых оценок

Прочитайте Булевы онлайн: https://riptutorial.com/ru/swift/topic/735/булевы

глава 13: Быстрые функции Advance

Вступление

flatMap функции, такие как map , flatMap , filter и reduce , используются для работы с различными типами коллекций, такими как Array и Dictionary. Предварительные функции обычно требуют небольшого кода и могут быть соединены вместе, чтобы создать сложную логику в краткой форме.

Examples

Введение с предварительными функциями

Давайте рассмотрим сценарий, чтобы лучше понять функцию продвижения,

```
struct User {
    var name: String
    var age: Int
    var country: String?
}

//User's information
let user1 = User(name: "John", age: 24, country: "USA")
let user2 = User(name: "Chan", age: 20, country: nil)
let user3 = User(name: "Morgan", age: 30, country: nil)
let user4 = User(name: "Rachel", age: 20, country: "UK")
let user5 = User(name: "Katie", age: 23, country: "USA")
let user6 = User(name: "David", age: 35, country: "USA")
let user7 = User(name: "Bob",age: 22, country: nil)

//User's array list
let arrUser = [user1, user2, user3, user4, user5, user6, user7]
```

Функция карты:

Используйте карту для циклического перемещения по коллекции и применяйте одну и ту же операцию к каждому элементу коллекции. Функция map возвращает массив, содержащий результаты применения функции преобразования или преобразования для каждого элемента.

```
//Fetch all the user's name from array
let arrUserName = arrUser.map({ $0.name }) // ["John", "Chan", "Morgan", "Rachel", "Katie",
"David", "Bob"]
```

Функция плоской карты:

Самое простое использование - это то, что название предполагает сгладить коллекцию коллекций.

```
// Fetch all user country name & ignore nil value.
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

Функция фильтра:

Используйте фильтр, чтобы перебирать коллекцию и возвращать массив, содержащий только те элементы, которые соответствуют условию include.

```
// Filtering USA user from the array user list.
let arrUSAUsers = arrUser.filter({ $0.country == "USA" }) // [user1, user5, user6]

// User chaining methods to fetch user's name who live in USA
```

```
let arrUserList = arrUser.filter({ $0.country == "USA" }).map({ $0.name }) // ["John",
"Katie", "David"]
```

Сокращение:

Используйте сокращение, чтобы объединить все элементы в коллекции, чтобы создать одно новое значение.

Swift 2.3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, combine: { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", combine: { $0 == "" ? $1 : $0 + ", " + $1 }) // John,
Chan, Morgan, Rachel, Katie, David, Bob
```

Swift 3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", { $0 == "" ? $1 : $0 + ", " + $1 }) // John, Chan,
Morgan, Rachel, Katie, David, Bob
```

Сгладить многомерный массив

Чтобы сгладить многомерный массив в единый размер, используются функции flatMap advance. Другим вариантом использования является пренебрежение значением nil из значений массива и отображения. Давайте проверим, например:

Предположим, у нас есть многомерный массив городов, и мы хотим отсортировать список имен городов в порядке возрастания. В этом случае мы можем использовать функцию flatMap, например: -

```
let arrStateName = [["Alaska", "Iowa", "Missouri", "New Mexico"], ["New York", "Texas",
"Washington", "Maryland"], ["New Jersey", "Virginia", "Florida", "Colorado"]]
```

Подготовка одномерного списка из многомерного массива,

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["Alaska", "Iowa", "Missouri", "New
Mexico", "New York", "Texas", "Washington", "Maryland", "New Jersey", "Virginia", "Florida",
"Colorado"]
```

Для сортировки значений массива мы можем использовать операцию цепочки или сортировать плоский массив. Ниже приведен пример операции цепочки,

```
// Swift 2.3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sort(<) // ["Alaska",     "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]

// Swift 3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sorted(by: <) // ["Alaska", "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]
```

Прочитайте Быстрые функции Advance онлайн: https://riptutorial.com/ru/swift/topic/9279/быстрые-функции-advance

глава 14: Быстрый HTTP-сервер от Kitura

Вступление

Kitura сервер с Kitura

Kitura - это веб-фреймворк, написанный быстро, который создан для веб-сервисов. Очень легко настроить HTTP-запросы. Для среды требуется либо OS X с установленной XCode, либо Linux, работающая под управлением Quick 3.0.

Examples

Привет, мир

конфигурация

Сначала создайте файл Package.swift. Это файл, который сообщает swift-компилятору, где находятся библиотеки. В этом приветственном мире мы используем репозитории GitHub. Нам нужны Kitura и HeliumLogger . Поместите следующий код внутри Package.swift. В нем указано название проекта как *kitura-helloworld*, а также URL-адреса зависимостей.

```
import PackageDescription
let package = Package(
    name: "kitura-helloworld",
        dependencies: [
            .Package(url: "https://github.com/IBM-Swift/HeliumLogger.git", majorVersion: 1,
minor: 6),
            .Package(url: "https://github.com/IBM-Swift/Kitura.git", majorVersion: 1, minor:
6) ] )
```

Затем создайте папку под названием «Источники». Внутри создайте файл с именем main.swift. Это файл, в котором мы реализуем всю логику для этого приложения. Введите следующий код в этот основной файл.

Импортировать библиотеки и включить ведение журнала

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

Добавление маршрутизатора. Маршрутизатор указывает путь, тип и т. Д. HTTP-запроса. Здесь мы добавляем обработчик запроса GET, который печатает *Hello world* , а затем почтовый запрос, который читает обычный текст из запроса и затем отправляет его обратно.

```
let router = Router()

router.get("/get") {
    request, response, next in
    response.send("Hello, World!")
    next()
}

router.post("/post") {
    request, response, next in
    var string: String?
    do{
        string = try request.readString()
```

```
    } catch let error {
        string = error.localizedDescription
    }
    response.send("Value \(string!) received.")
    next()
}
```

Укажите порт для запуска службы

```
let port = 8080
```

Свяжите маршрутизатор и порт вместе и добавьте их как HTTP-сервис

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

казнить

Перейдите в корневую папку с папкой Package.swift и папкой Resources. Выполните следующую команду. Компилятор Swift автоматически загрузит указанные ресурсы в папку Package.swift в папку Packages, а затем скомпилирует эти ресурсы с помощью main.swift

```
swift build
```

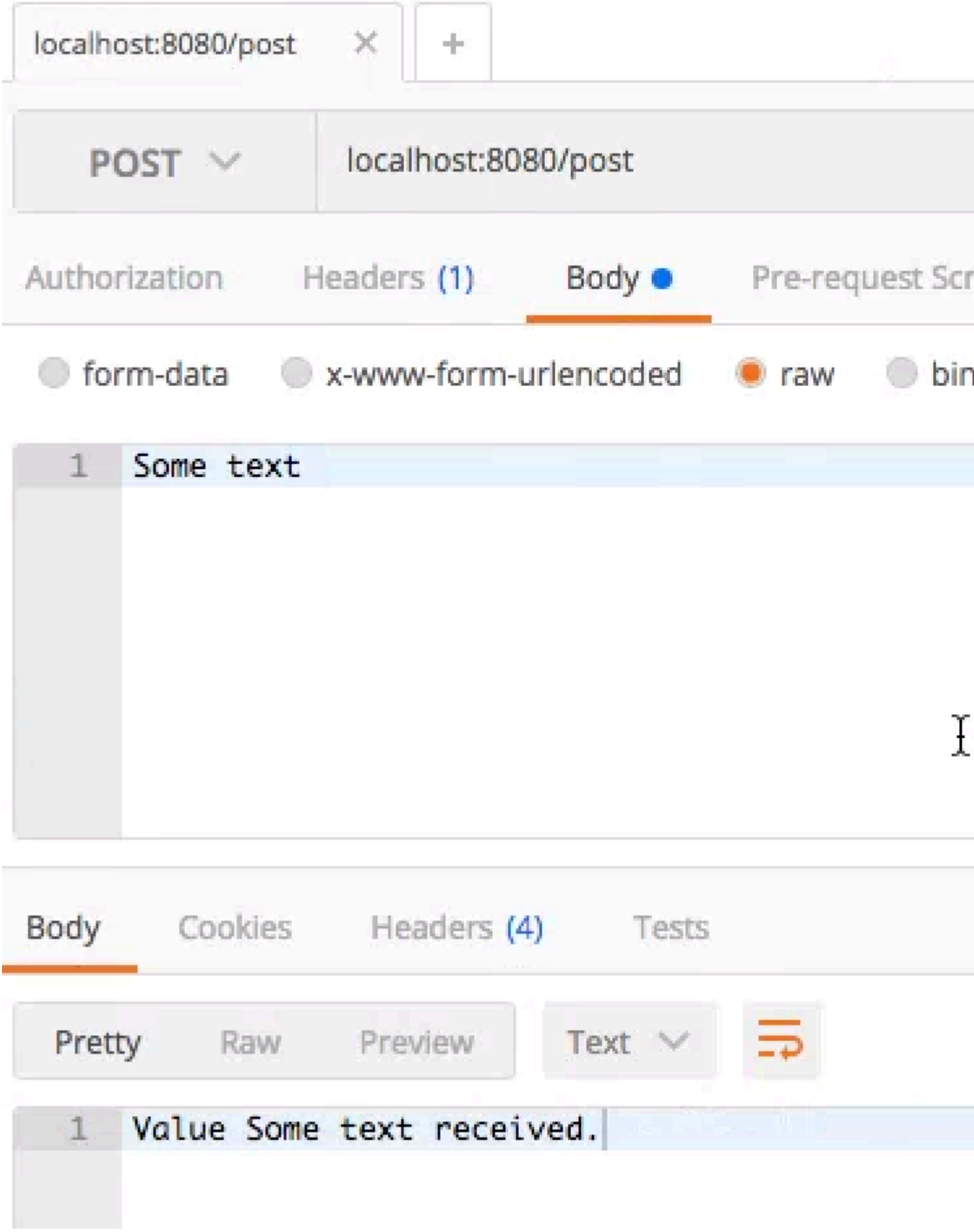
Когда сборка завершена, исполняемый файл будет размещен в этом месте. Дважды щелкните этот исполняемый файл, чтобы запустить сервер.

```
.build/debug/kitura-helloworld
```

утверждать

Откройте браузер, введите localhost:8080/get как url и нажмите enter. Привет, мировая страница должна выйти.

Откройте приложение HTTP-запроса, разместите обычный текст на localhost:8080/post . Строка ответа отобразит введенный текст правильно.



Прочитайте Быстрый HTTP-сервер от Kitura онлайн: https://riptutorial.com/ru/swift/topic/10690/быстрый-http-сервер-от-kitura

глава 15: Внедрение зависимости

Examples

Инъекция зависимостей с контроллерами просмотра

Вводная инъекция Dependenct

Приложение состоит из множества объектов, которые взаимодействуют друг с другом. Объекты обычно зависят от других объектов для выполнения какой-либо задачи. Когда объект отвечает за ссылки на свои собственные зависимости, он приводит к высокосвязному, труднодоступному и трудно меняющемуся коду.

Инъекция зависимостей - это шаблон проектирования программного обеспечения, который реализует инверсию управления для разрешения зависимостей. Инъекция проходит зависимость от зависимого объекта, который будет ее использовать. Это позволяет отделить зависимости клиента от поведения клиента, что позволяет непринужденно связывать приложение.

Не следует путать с приведенным выше определением - инъекция зависимости просто означает предоставление объекту своих переменных экземпляра.

Это так просто, но это дает много преимуществ:

- проще протестировать ваш код (используя автоматические тесты, такие как тесты на единицу и UI)
- при использовании в тандеме с программно-ориентированным программированием упрощает изменение реализации определенного класса - проще рефакторировать
- он делает код более модульным и многоразовым

Существуют три наиболее часто используемых способа: Инъекция зависимостей (DI) может быть реализована в приложении:

1. Ввод инициализатора
2. Впрыск недвижимости
3. Использование сторонних инфраструктур DI (например, Swinject, Cleanse, Dip или Typhoon)

Есть интересная статья со ссылками на другие статьи о Injection Dependency, поэтому проверьте, хотите ли вы углубиться в принцип DI и инверсии управления.

Давайте покажем, как использовать DI с контроллерами View - ежедневную задачу для среднего разработчика iOS.

Пример без DI

У нас будет два контроллера просмотра: **LoginViewController** и **TimelineViewController** . LoginViewController используется для входа в систему и после успешного завершения, он переключится на TimelineViewController. Оба контроллера просмотра зависят от **FirebaseNetworkService** .

LoginViewController

```
class LoginViewController: UIViewController {

    var networkService = FirebaseNetworkService()

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

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService = FirebaseNetworkService()

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

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

FirebaseNetworkService

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // Implementation not important for this example
    }

    func logutCurrentUser() {
        // Implementation not important for this example
    }
}
```

Этот пример очень прост, но предположим, что у вас есть 10 или 15 различных контроллеров представления, а некоторые из них также зависят от FirebaseNetworkService. В какой-то момент вы хотите изменить Firebase в качестве своего вспомогательного сервиса с помощью внутренней поддержки вашей компании. Для этого вам нужно будет пройти через каждый контроллер представления и изменить FirebaseNetworkService с помощью CompanyNetworkService. И если некоторые из методов в CompanyNetworkService изменились, у вас будет много работы.

Тестирование модулей и UI не входит в объем этого примера, но если вы хотите объединить тестовые контроллеры представлений с плотно связанными зависимостями, вам будет очень тяжело это делать.

Давайте перепишем этот пример и добавим Network Service в наши контроллеры представлений.

Пример с инъекцией зависимостей

Чтобы максимально использовать Injection Dependency Injection, давайте определим функциональность сетевой службы в протоколе. Таким образом, контроллеры представлений, зависящие от сетевой службы, даже не должны знать о реальной ее реализации.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Добавьте реализацию протокола NetworkService:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }

    func logutCurrentUser() {
        // Firebase implementation
```

```
    }
}
```

Давайте изменим LoginViewController и TimelineViewController для использования нового протокола NetworkService вместо FirebaseNetworkService.

LoginViewController

```
class LoginViewController: UIViewController {

    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

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

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService: NetworkService?

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

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService?.logutCurrentUser()
    }
}
```

Теперь встает вопрос: как мы вставляем правильную реализацию NetworkService в LoginViewController и TimelineViewController?

Поскольку LoginViewController является начальным контроллером представления и будет отображаться каждый раз, когда приложение запускается, мы можем вводить все зависимости в **AppDelegate** .

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // This logic will be different based on your project's structure or whether
    // you have a navigation controller or tab bar controller for your starting view
controller
    if let loginVC = window?.rootViewController as? LoginViewController {
        loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

В AppDelegate мы просто берем ссылку на первый контроллер представления (LoginViewController) и вводим реализацию NetworkService с использованием метода впрыска свойств.

Теперь следующая задача - внедрить реализацию NetworkService в TimelineViewController. Самый простой способ - сделать это, когда LoginViewController переходит к TimlineViewController.

Мы добавим код инъекции в метод prepareForSegue в LoginViewController (если вы используете другой подход для навигации по контроллерам представлений, поместите там код для инъекций).

Теперь класс LoginViewController выглядит следующим образом:

```
class LoginViewController: UIViewController {
    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

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

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // Injecting the NetworkService implementation
                timelineVC.networkService = networkService
            }
        }
    }
}
```

Мы закончили, и все так просто.

Теперь представьте, что мы хотим переключить реализацию NetworkService с Firebase на реализацию бэкэнд нашей заказной компании. Все, что нам нужно сделать, это:

Добавить класс реализации NetworkService:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Переключите FirebaseNetworkServiceImpl с новой реализацией в AppDelegate:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // This logic will be different based on your project's structure or whether
        // you have a navigation controller or tab bar controller for your starting view
controller
        if let loginVC = window?.rootViewController as? LoginViewController {
            loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

Вот и все, мы включили всю подкладочную реализацию протокола NetworkService, даже не коснувшись LoginViewController или TimelineViewController.

Поскольку это простой пример, вы можете не увидеть все преимущества прямо сейчас, но если вы попытаетесь использовать DI в своих проектах, вы увидите преимущества и всегда будете использовать Injection Dependency.

Типы инъекций зависимостей

В этом примере будет показано, как использовать шаблон проектирования зависимостей (**DI**) в Swift, используя следующие методы:

1. **Инициализатор Инъекции** (правильный термин - Инъекция конструктора, но поскольку Swift имеет инициализаторы, он называется инициализационной инъекцией)
2. **Инъекция недвижимости**
3. **Метод инъекции**

Пример настройки без DI

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("Engine stopped")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

Инъекция зависимостей инициализатора

Как видно из названия, все зависимости вводятся через инициализатор класса. Чтобы вводить зависимости через инициализатор, мы добавим инициализатор в класс Train .

Класс Train теперь выглядит следующим образом:

```
class Train {
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

Когда мы хотим создать экземпляр класса Train, мы будем использовать инициализатор для инъекции конкретной реализации Engine:

```
let train = Train(engine: TrainEngine())
```

ПРИМЕЧАНИЕ . Основным преимуществом инъекции инициализатора по сравнению с введением свойств является то, что мы можем установить переменную как частную переменную или даже сделать ее константой с ключевым словом let (как это было в нашем примере). Таким образом, мы можем убедиться, что никто не может получить к нему доступ или изменить его.

Свойства Инъекции зависимостей

Использование свойств DI еще проще, если использовать инициализатор. Давайте введем зависимость PassengerCar к объекту поезда, который мы уже создали, используя свойства DI:

```
train.mainCar = PassengerCar()
```

Вот и все. Наш mainCar нашего поезда теперь является экземпляром PassengerCar .

Инъекция зависимостей метода

Этот тип инъекций зависимостей немного отличается от предыдущих двух, поскольку он не будет влиять на весь объект, но он будет вводить только зависимость, которая будет использоваться в рамках одного конкретного метода. Когда зависимость используется только в одном методе, обычно нехорошо вносить в нее весь объект. Давайте добавим новый метод в класс Train:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

Теперь, если мы назовем метод класса нового поезда, мы будем вводить TrainCar используя инъекцию зависимости метода.

```
train.reparkCar(trainCar: RestaurantCar())
```

Прочитайте Внедрение зависимости онлайн: https://riptutorial.com/ru/swift/topic/8198/внедрение-зависимости

глава 16: Вход в Swift

замечания

println и debugPrintln где удалено в Swift 2.0.

Источники:

https://developer.apple.com/library/content/technotes/tn2347/_index.html
http://ericasadun.com/2015/05/22/swift-logging/

http://www.dotnetperls.com/print-swift

Examples

Отладка печати

Debug Print показывает представление экземпляра, наиболее подходящее для отладки.

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

Урожайность

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

Эта дополнительная информация может быть очень важной, например:

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

Урожайность

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

Обратите внимание, как на первом выходе появляется, что в массиве есть 4 элемента, а не 3. Для таких целей предпочтительнее, когда отладка используется debugPrint

Обновление классов для отладки и печати значений

```
struct Foo: Printable, DebugPrintable {
    var description: String {return "Clear description of the object"}
    var debugDescription: String {return "Helpful message for debugging"}
```

```
}

var foo = Foo()

print(foo)
debugPrint(foo)

>>> Clear description of the object
>>> Helpful message for debugging
```

свалка

dump печатает содержимое объекта посредством отражения (зеркалирования).

Детальный вид массива:

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

Печать:

```
▿ 4 элемента
- [0]: Джо
- [1]: Джейн
- [2]: Джим
- [3]: Джойс
```

Для словаря:

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

Печать:

```
▿ 3 пары ключ / значение
▿ [0]: (2 элемента)
- .0: bar
- .1: 33
▿ [1]: (2 элемента)
- .0: baz
- .1: 42
▿ [2]: (2 элемента)
- .0: foo
- .1: 10
```

dump объявлен как dump(_:name:indent:maxDepth:maxItems:) .

Первый параметр не имеет метки.

Доступны другие параметры, такие как name для установки метки для проверяемого объекта:

```
dump(attributes, name: "mirroring")
```

Печать:

```
▿ зеркалирование: 3 пары ключ / значение
▿ [0]: (2 элемента)
- .0: bar
- .1: 33
▿ [1]: (2 элемента)
- .0: baz
```

```
    - .1: 42
    ▾ [2]: (2 элемента)
    - .0: foo
    - .1: 10
```

Вы также можете печатать только определенное количество элементов с помощью maxItems: чтобы проанализировать объект до определенной глубины с помощью maxDepth: и изменить отступ отпечатанных объектов с помощью indent:

print () vs dump ()

Многие из нас начинают отладку с помощью простого print() . Допустим, у нас такой класс:

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

и у нас есть экземпляр Abc как таковой:

```
let abc = Abc()
```

Когда мы запускаем print() для переменной, выход

```
App.Abc
```

а выходы dump()

```
App.Abc #0
- a: "aa"
- b: "bb"
```

Как видно, dump() выводит всю иерархию классов, а print() просто выводит имя класса.

Поэтому dump() особенно полезен для отладки пользовательского интерфейса

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

С dump(view) мы получаем:

```
 - <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
    - super: UIResponder
      - NSObject
```

Во время print(view) мы получаем:

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

Больше информации о классе с dump() , и поэтому он более полезен при отладке самого класса.

печать против NSLog

В быстром режиме мы можем использовать функции print() и NSLog() для печати на консоли Xcode.

Но есть много отличий в функциях print() и NSLog() , таких как:

1 TimeStamp: NSLog() будет печатать метку времени вместе с переданной ей строкой, но print() не будет печатать метку времени.

например

```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

Выход:

```
[1, 2, 3, 4, 5]
2017-05-31 13: 14: 38.582 ProjetName [2286: 7473287] [1, 2, 3, 4, 5]
```

Он также будет печатать имя **проекта** вместе с меткой времени.

2 Только строка: NSLog() принимает только String как вход, но print() может печатать любые входные данные, переданные ему.
например

```
let array = [1, 2, 3, 4, 5]
print(array) //prints [1, 2, 3, 4, 5]
NSLog(array) //error: Cannot convert value of type [Int] to expected argument type 'String'
```

3 Производительность: функция NSLog() работает очень **медленно по** сравнению с функцией print() .

4 Синхронизация: NSLog() обрабатывает одновременное использование из многопоточной среды и выводит результат без перекрытия. Но print() не будет обрабатывать такие случаи и сжиматься при выводе prating.

5 Консоль устройств: выходы NSLog() на консоли устройства также можно увидеть, подключив наше устройство к Xcode. print() не будет выводить вывод на консоль устройства.

Прочитайте Вход в Swift онлайн: https://riptutorial.com/ru/swift/topic/3966/вход-в-swift

глава 17: Вывод ключа PBKDF2

Examples

Основанный на пароле вывод 2 (Swift 3)

Деривация ключа на основе пароля может использоваться как для получения ключа шифрования из текста пароля, так и для сохранения пароля для целей аутентификации.

Существует несколько алгоритмов хэширования, которые могут использоваться, включая SHA1, SHA256, SHA512, которые предоставляются этим примером кода.

Параметр rounds используется для того, чтобы сделать расчет медленным, чтобы злоумышленник должен был тратить значительное время на каждую попытку. Типичные значения задержки падают в диапазоне от 100 мс до 500 мс, более короткие значения могут использоваться, если есть неприемлемая производительность.

Этот пример требует Common Crypto
Необходимо иметь заголовок заголовка проекта:
#import <CommonCrypto/CommonCrypto.h>
Добавьте в проект Security.framework .

Параметры:

```
password     password String
salt         salt Data
keyByteCount number of key bytes to generate
rounds       Iteration rounds

returns      Derived key


func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int, rounds:
Int) -> Data? {
    let passwordData = password.data(using:String.Encoding.utf8)!
    var derivedKeyData = Data(repeating:0, count:keyByteCount)

    let derivationStatus = derivedKeyData.withUnsafeMutableBytes {derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
```

```
                derivedKeyBytes, derivedKeyData.count)
        }
    }
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKeyData
}
```

Пример использования:

```
let password     = "password"
//let salt       = "saltData".data(using: String.Encoding.utf8)!
let salt         = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds       = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt, keyByteCount:keyByteCount,
rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
```

Пример:

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Вывод пароля на основе пароля 2 (Swift 2.3)

См. Пример Swift 3 для использования информации и заметок

```
func pbkdf2SHA1(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA256(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA512(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: [UInt8], keyCount: Int, rounds:
UInt32!) -> [UInt8]! {
    let derivedKey   = [UInt8](count:keyCount, repeatedValue:0)
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    let derivationStatus = CCKeyDerivationPBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer<Int8>(passwordData.bytes), passwordData.length,
        UnsafePointer<UInt8>(salt), salt.count,
        CCPseudoRandomAlgorithm(hash),
        rounds,
        UnsafeMutablePointer<UInt8>(derivedKey),
        derivedKey.count)
```

```
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKey
}
```

Пример использования:

```
let password = "password"
// let salt  = [UInt8]("saltData".utf8)
let salt     = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let rounds   = 100_000
let keyCount = 16

let derivedKey = pbkdf2SHA1(password, salt:salt, keyCount:keyCount, rounds:rounds)
print("derivedKey (SHA1):   \(NSData(bytes:derivedKey!, length:derivedKey!.count))")
```

Пример:

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Калибровка вывода ключа на основе пароля (Swift 2.3)

См. Пример Swift 3 для использования информации и заметок

```
func pbkdf2SHA1Calibrate(password:String, salt:[UInt8], msec:Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Пример использования:

```
let saltData       = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Пример:

```
     За 100 мс задержка, раунды: 94339
```

Калибровка вывода ключа на основе пароля (Swift 3)

Определите количество раундов PRF для использования для определенной задержки на текущей платформе.

Несколько параметров дефолтны для репрезентативных значений, которые не должны существенно влиять на количество раундов.

```
password Sample password.
salt     Sample salt.
msec     Targeted duration we want to achieve for a key derivation.

returns  The number of iterations to use for the desired processing time.


func pbkdf2SHA1Calibrate(password: String, salt: Data, msec: Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Пример использования:

```
let saltData       = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(password:passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Пример:

```
For 100 msec delay, rounds: 93457
```

Прочитайте Вывод ключа PBKDF2 онлайн: https://riptutorial.com/ru/swift/topic/7053/вывод-ключа-pbkdf2

глава 18: Дженерики

замечания

Общий код позволяет писать гибкие, многократно используемые функции и типы, которые могут работать с любым типом, в зависимости от требований, которые вы определяете. Вы можете написать код, который избегает дублирования и выражает свое намерение четким, абстрактным образом.

Generics - одна из самых мощных функций Swift, и большая часть стандартной библиотеки Swift построена с общим кодом. Например, типы Array Swift и Dictionary являются общими коллекциями. Вы можете создать массив, который содержит значения Int , или массив, который содержит значения String , или действительно массив для любого другого типа, который может быть создан в Swift. Аналогичным образом, вы можете создать словарь для хранения значений любого указанного типа, и нет ограничений на то, что может быть этим типом.

Источник: Быстрый язык программирования Apple

Examples

Ограничение общих типов закладок

Можно заставить параметры типа универсального класса реализовать протокол , например Equatable

```
class MyGenericClass<Type: Equatable>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func valueEquals(anotherValue: Type) -> Bool{
        return self.value == anotherValue
    }
}
```

Всякий раз, когда мы создаем новый MyGenericClass , параметр type должен реализовывать Equatable протокол (гарантируя, что параметр типа можно сравнить с другой переменной того же типа, используя ==)

```
let myFloatGeneric = MyGenericClass<Double>(value: 2.71828) // valid
let myStringGeneric = MyGenericClass<String>(value: "My String") // valid

// "Type [Int] does not conform to protocol 'Equatable'"
let myInvalidGeneric = MyGenericClass<[Int]>(value: [2])

let myIntGeneric = MyGenericClass<Int>(value: 72)
print(myIntGeneric.valueEquals(72)) // true
print(myIntGeneric.valueEquals(-274)) // false

// "Cannot convert value of type 'String' to expected argument type 'Int'"
print(myIntGeneric.valueEquals("My String"))
```

Основы дженериков

Generics - это заполнители для типов, позволяющие писать гибкий код, который можно применять для нескольких типов. Преимущество использования дженериков над Any заключается в том, что они все еще позволяют компилятору обеспечить сильную безопасность типов.

Общий заполнитель определяется в угловых скобках <> .

Общие функции

Для функций этот заполнитель помещается после имени функции:

```
/// Picks one of the inputs at random, and returns it
func pickRandom<T>(_ a:T, _ b:T) -> T {
    return arc4random_uniform(2) == 0 ? a : b
}
```

В этом случае общим заполнителем является T Когда вы приходите на вызов функции, Swift может вывести тип T для вас (поскольку он просто выступает в качестве заполнителя для фактического типа).

```
let randomOutput = pickRandom(5, 7) // returns an Int (that's either 5 or 7)
```

Здесь мы передаем две целые функции. Поэтому Swift выводит T == Int - таким образом, сигнатура функции выводится как (Int, Int) -> Int .

Из-за сильной безопасности типа, которую предлагают дженерики - оба аргумента и возврат функции должны быть *одного* типа. Поэтому следующее не будет компилироваться:

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // error: cannot convert value of type 'Int' to expected
argument type 'Foo'
```

Общие типы

Чтобы использовать дженерики с классами , структурами или перечислениями , вы можете определить общий заполнитель после имени типа.

```
class Bar<T> {
    var baz : T

    init(baz:T) {
        self.baz = baz
    }
}
```

Этот общий заполнитель потребует тип, когда вы приступите к использованию класса Bar . В этом случае это можно сделать из инициализации init(baz:T) .

```
let bar = Bar(baz: "a string") // bar's type is Bar<String>
```

Здесь общий заполнитель T определяется как тип String , создавая таким образом экземпляр Bar<String> . Вы также можете указать тип явно:

```
let bar = Bar<String>(baz: "a string")
```

При использовании с типом данный общий заполнитель будет сохранять свой тип на протяжении всего срока службы данного экземпляра и не может быть изменен после инициализации. Поэтому, когда вы

получаете доступ к свойству baz , он всегда будет иметь тип String для данного экземпляра.

```
let str = bar.baz // of type String
```

Передача общих типов

Когда вы сталкиваетесь с общими типами, в большинстве случаев вы должны быть явно о типичном типе-заполнителе, который вы ожидаете. Например, как вход функции:

```
func takeABarInt(bar:Bar<Int>) {
    ...
}
```

Эта функция будет принимать только Bar<Int> . Попытка передать экземпляр Bar где общий тип заполнителя не является Int приведет к ошибке компилятора.

Именование общего места

Общие имена заполнителей не ограничиваются только отдельными буквами. Если данный заполнитель представляет собой содержательную концепцию, вы должны дать ему описательное имя. Например, в Array Swift имеется общий заполнитель, называемый Element , который определяет тип элемента заданного экземпляра Array .

```
public struct Array<Element> : RandomAccessCollection, MutableCollection {
    ...
}
```

Примеры общего класса

Общий класс с типом параметра Type

```
class MyGenericClass<Type>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func setValue(value: Type){
        self.value = value
    }
}
```

Теперь мы можем создавать новые объекты с использованием параметра типа

```
let myStringGeneric = MyGenericClass<String>(value: "My String Value")
let myIntGeneric = MyGenericClass<Int>(value: 42)

print(myStringGeneric.getValue()) // "My String Value"
print(myIntGeneric.getValue()) // 42

myStringGeneric.setValue("Another String")
myIntGeneric.setValue(1024)

print(myStringGeneric.getValue()) // "Another String"
```

```
print(myIntGeneric.getValue()) // 1024
```

Дженерики также могут быть созданы с несколькими параметрами типа

```
class AnotherGenericClass<TypeOne, TypeTwo, TypeThree>{

    var value1: TypeOne
    var value2: TypeTwo
    var value3: TypeThree
    init(value1: TypeOne, value2: TypeTwo, value3: TypeThree){
        self.value1 = value1
        self.value2 = value2
        self.value3 = value3
    }

    func getValueOne() -> TypeOne{return self.value1}
    func getValueTwo() -> TypeTwo{return self.value2}
    func getValueThree() -> TypeThree{return self.value3}
}
```

И используется таким же образом

```
let myGeneric = AnotherGenericClass<String, Int, Double>(value1: "Value of pi", value2: 3,
value3: 3.14159)

print(myGeneric.getValueOne() is String) // true
print(myGeneric.getValueTwo() is Int) // true
print(myGeneric.getValueThree() is Double) // true
print(myGeneric.getValueTwo() is String) // false

print(myGeneric.getValueOne()) // "Value of pi"
print(myGeneric.getValueTwo()) // 3
print(myGeneric.getValueThree()) // 3.14159
```

Наследование общего класса

Родовые классы можно унаследовать:

```
// Models
class MyFirstModel {
}

class MySecondModel: MyFirstModel {
}

// Generic classes
class MyFirstGenericClass<T: MyFirstModel> {

    func doSomethingWithModel(model: T) {
        // Do something here
    }

}

class MySecondGenericClass<T: MySecondModel>: MyFirstGenericClass<T> {

    override func doSomethingWithModel(model: T) {
        super.doSomethingWithModel(model)
```

```
        // Do more things here
    }

}
```

Использование генераторов для упрощения функций массива

Функция, расширяющая функциональность массива путем создания объектно-ориентированной функции удаления.

```
// Need to restrict the extension to elements that can be compared.
// The `Element` is the generics name defined by Array for its item types.
// This restriction also gives us access to `index(of:_)` which is also
// defined in an Array extension with `where Element: Equatable`.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        if let index = self.index(of: element ) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

использование

```
var myArray = [1,2,3]
print(myArray)

// Prints [1,2,3]
```

Используйте эту функцию для удаления элемента без необходимости индексирования. Просто передайте объект для удаления.

```
myArray.remove(2)
print(myArray)

// Prints [1,3]
```

Использование дженериков для повышения безопасности типов

Давайте рассмотрим этот пример без использования дженериков

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

Объявление протокола кажется прекрасным, если вы его не используете.

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

Почему вы должны TestObject результат в TestObject ? Из-за Any типа возврата в объявлении протокола.

Используя дженерики, вы можете избежать этой проблемы, которая может вызвать ошибки времени выполнения (и мы не хотим их использовать!)

```
protocol JSONDecodable {
    associatedtype Element
    static func from(_ json: [String: Any]) -> Element?
}

struct TestObject: JSONDecodable {
    static func from(_ json: [String: Any]) -> TestObject? {
    }
}

let testObject = TestObject.from(myJson) // testObject is now automatically `TestObject?`
```

Расширенные ограничения типов

Можно указать несколько ограничений типа для генериков с использованием предложения where :

```
func doSomething<T where T: Comparable, T: Hashable>(first: T, second: T) {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

Также допустимо написать предложение where после списка аргументов:

```
func doSomething<T>(first: T, second: T) where T: Comparable, T: Hashable {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

Расширения могут быть ограничены типами, удовлетворяющими условиям. Функция доступна только для экземпляров, которые удовлетворяют условиям типа:

```
// "Element" is the generics type defined by "Array". For this example, we
// want to add a function that requires that "Element" can be compared, that
// is: it needs to adhere to the Equatable protocol.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        // We could also use "self.index(of: element)" here, as "index(of:_)"
        // is also defined in an extension with "where Element: Equatable".
        // For the sake of this example, explicitly make use of the Equatable.
        if let index = self.index(where: { $0 == element }) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/swift/topic/774/дженерики

глава 19: Затворы

Синтаксис

- var closVar: (<parameters>) -> (<returnType>) // В качестве переменной или типа свойства
- typealias ClosureType = (<parameters>) -> (<returnType>)
- {[<captureList>] (<parameters>) <throws-ness> -> <returnType> в <statements>} // Полный синтаксис закрытия

замечания

Дополнительную информацию о закрытии Swift см. В документации Apple .

Examples

Основы закрытия

Закрытие (также известное как **блоки** или **лямбда**) - это фрагменты кода, которые можно хранить и передавать внутри вашей программы.

```
let sayHi = { print("Hello") }
// The type of sayHi is "() -> ()", aka "() -> Void"

sayHi()  // prints "Hello"
```

Как и другие функции, замыкания могут принимать аргументы и возвращать результаты или бросать ошибки :

```
let addInts = { (x: Int, y: Int) -> Int in
    return x + y
}
// The type of addInts is "(Int, Int) -> Int"

let result = addInts(1, 2)  // result is 3

let divideInts = { (x: Int, y: Int) throws -> Int in
    if y == 0 {
        throw MyErrors.DivisionByZero
    }
    return x / y
}
// The type of divideInts is "(Int, Int) throws -> Int"
```

Закрытие может **захватывать** значения из их объема:

```
// This function returns another function which returns an integer
func makeProducer(x: Int) -> (() -> Int) {
    let closure = { x }  // x is captured by the closure
    return closure
}

// These two function calls use the exact same code,
// but each closure has captured different values.
let three = makeProducer(3)
let four = makeProducer(4)
three()  // returns 3
four()  // returns 4
```

Замыкания могут передаваться непосредственно в функции:

```
let squares = (1...10).map({ $0 * $0 })  // returns [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
let squares = (1...10).map { $0 * $0 }

NSURLSession.sharedSession().dataTaskWithURL(myURL,
    completionHandler: { (data: NSData?, response: NSURLResponse?, error: NSError?) in
        if let data = data {
            print("Request succeeded, data: \(data)")
        } else {
            print("Request failed: \(error)")
        }
    }).resume()
```

Варианты синтаксиса

Основной синтаксис закрытия

```
{ [ list list ] ( parameters ) throws-ness -> return type in body } .
```

Многие из этих частей могут быть опущены, поэтому существует несколько эквивалентных способов написания простых замыканий:

```
let addOne = { [] (x: Int) -> Int in return x + 1 }
let addOne = { [] (x: Int) -> Int in x + 1 }
let addOne = { (x: Int) -> Int in x + 1 }
let addOne = { x -> Int in x + 1 }
let addOne = { x in x + 1 }
let addOne = { $0 + 1 }

let addOneOrThrow = { [] (x: Int) throws -> Int in return x + 1 }
let addOneOrThrow = { [] (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { x throws -> Int in x + 1 }
let addOneOrThrow = { x throws in x + 1 }
```

- Список захвата может быть опущен, если он пуст.
- Параметры не нужны аннотации типов, если их типы можно вывести.
- Тип возвращаемого значения не нужно указывать, если это можно сделать.
- Параметры не обязательно должны быть названы; вместо этого они могут ссылаться на $0 , $1 , $2 и т. д.
- Если в закрытии содержится одно выражение, значение которого должно быть возвращено, ключевое слово return может быть опущено.
- Если замыкание выведено для того, чтобы вызвать ошибку, оно записывается в контексте, который ожидает закрытия броска или не вызывает ошибку, throws могут быть опущены.

```
// The closure's type is unknown, so we have to specify the type of x and y.
// The output type is inferred to be Int, because the + operator for Ints returns Int.
let addInts = { (x: Int, y: Int) in x + y }

// The closure's type is specified, so we can omit the parameters' type annotations.
let addInts: (Int, Int) -> Int = { x, y in x + y }
let addInts: (Int, Int) -> Int = { $0 + $1 }
```

Передача замыканий в функции

Функции могут принимать замыкания (или другие функции) в качестве параметров:

```
func foo(value: Double, block: () -> Void) { ... }
func foo(value: Double, block: Int -> Int) { ... }
func foo(value: Double, block: (Int, Int) -> String) { ... }
```

Синтаксис закрытия трейлинга

Если последним параметром функции является замыкание, замыкающие скобки { / } могут быть записаны **после** вызова функции:

```
foo(3.5, block: { print("Hello") })

foo(3.5) { print("Hello") }

dispatch_async(dispatch_get_main_queue(), {
    print("Hello from the main queue")
})

dispatch_async(dispatch_get_main_queue()) {
    print("Hello from the main queue")
}
```

Если единственным аргументом функции является замыкание, вы можете также опустить пару круглых скобок () при вызове с помощью синтаксиса закрытия закрытия:

```
func bar(block: () -> Void) { ... }
```

```
bar() { print("Hello") }

bar { print("Hello") }
```

Параметры `@noescape`

Параметры закрытия, отмеченные как @noescape , гарантированно выполняются до вызова функции, поэтому используйте self. не требуется внутри корпуса закрывания:

```
func executeNow(@noescape block: () -> Void) {
    // Since `block` is @noescape, it's illegal to store it to an external variable.
    // We can only call it right here.
    block()
}

func executeLater(block: () -> Void) {
    dispatch_async(dispatch_get_main_queue()) {
        // Some time in the future...
        block()
    }
}
```

```
class MyClass {
    var x = 0
    func showExamples() {
        // error: reference to property 'x' in closure requires explicit 'self.' to make
capture semantics explicit
        executeLater { x = 1 }

        executeLater { self.x = 2 }  // ok, the closure explicitly captures self
```

```
        // Here "self." is not required, because executeNow() takes a @noescape block.
        executeNow { x = 3 }

        // Again, self. is not required, because map() uses @noescape.
        [1, 2, 3].map { $0 + x }
    }
}
```

Свифт 3 Примечание:

Обратите внимание, что в Swift 3 вы больше не отмечаете блоки как @noescape. Блоки теперь **не** экранируются по умолчанию. В Swift 3 вместо того, чтобы отмечать закрытие как не-экранирование, вы отмечаете параметр функции, который является закрывающим закрытием, как escape-код, используя ключевое слово «@escaping».

---

throws И rethrows

Закрытие, как и другие функции, может вызывать ошибки :

```
func executeNowOrIgnoreError(block: () throws -> Void) {
    do {
        try block()
    } catch {
        print("error: \(error)")
    }
}
```

Функция может, конечно, передать ошибку вместе со своим вызывающим:

```
func executeNowOrThrow(block: () throws -> Void) throws {
    try block()
}
```

Однако, если переданный блок *не* выбрасывает, вызывающий объект все еще застревает с функцией throwing:

```
// It's annoying that this requires "try", because "print()" can't throw!
try executeNowOrThrow { print("Just printing, no errors here!") }
```

Решением является **rethrows** , что **rethrows** , что функция может только бросать, **если его параметр закрытия бросает** :

```
func executeNowOrRethrow(block: () throws -> Void) rethrows {
    try block()
}

// "try" is not required here, because the block can't throw an error.
executeNowOrRethrow { print("No errors are thrown from this closure") }

// This block can throw an error, so "try" is required.
try executeNowOrRethrow { throw MyError.Example }
```

Многие стандартные библиотечные функции используют rethrows , включая map() , filter() и indexOf() .

Захваты, сильные / слабые ссылки и циклы сохранения

```
class MyClass {
    func sayHi() { print("Hello") }
    deinit { print("Goodbye") }
}
```

Когда замыкание фиксирует ссылочный тип (экземпляр класса), по умолчанию имеет сильную ссылку:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a strong reference to `obj`: the object will be kept alive
    // as long as the closure itself is alive.
    closure = { obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope
closure()  // The object is still alive; prints "Hello"
```

Список **захвата** закрытия можно использовать для указания слабой или неопубликованной ссылки:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a weak reference to `obj`: the closure will not keep the object alive;
    // the object becomes optional inside the closure.
    closure = { [weak obj] in obj?.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // `obj` is nil from inside the closure; this does not print anything.
```

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures an unowned reference to `obj`: the closure will not keep the object alive;
    // the object is always assumed to be accessible while the closure is alive.
    closure = { [unowned obj] in obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // crash! obj is being accessed after it's deallocated.
```

Для получения дополнительной информации см. Раздел « Управление памятью» и раздел « Автоматический подсчет ссылок » на языке «Быстрый язык программирования».

Сохранять циклы

Если объект удерживается на замыкании, который также содержит сильную ссылку на объект, это **цикл сохранения** . Если цикл не сломан, память, хранящая объект и закрытие, будет просочиться (никогда не будет исправлена).

```
class Game {
    var score = 0
    let controller: GCController
    init(controller: GCController) {
        self.controller = controller

        // BAD: the block captures self strongly, but self holds the controller
        // (and thus the block) strongly, which is a cycle.
        self.controller.controllerPausedHandler = {
            let curScore = self.score
            print("Pause button pressed; current score: \(curScore)")
```

```
        }

        // SOLUTION: use `weak self` to break the cycle.
        self.controller.controllerPausedHandler = { [weak self] in
            guard let strongSelf = self else { return }
            let curScore = strongSelf.score
            print("Pause button pressed; current score: \(curScore)")
        }
    }
}
```

Использование замыканий для асинхронного кодирования

Закрытие часто используется для асинхронных задач, например, при получении данных с веб-сайта.

3.0

```
func getData(urlString: String, callback: (result: NSData?) -> Void) {

    // Turn the URL string into an NSURLRequest.
    guard let url = NSURL(string: urlString) else { return }
    let request = NSURLRequest(URL: url)

    // Asynchronously fetch data from the given URL.
    let task = NSURLSession.sharedSession().dataTaskWithRequest(request) {(data: NSData?,
response: NSURLResponse?, error: NSError?) in

        // We now have the NSData response from the website.
        // We can get it "out" of the function by using the callback
        // that was passed to this function as a parameter.

        callback(result: data)
    }

    task.resume()
}
```

Эта функция асинхронна, поэтому не будет блокировать поток, на который он вызывается (он не будет замораживать интерфейс, если вызван в основном потоке вашего приложения GUI).

3.0

```
print("1. Going to call getData")

getData("http://www.example.com") {(result: NSData?) -> Void in

    // Called when the data from http://www.example.com has been fetched.
    print("2. Fetched data")
}

print("3. Called getData")
```

Поскольку задача является асинхронной, вывод будет выглядеть следующим образом:

```
"1. Going to call getData"
"3. Called getData"
"2. Fetched data"
```

Поскольку код внутри закрытия, print("2. Fetched data") не будет вызываться до тех пор, пока не

будут получены данные из URL-адреса.

Закрытие и псевдоним типа

Закрытие может быть определено с помощью typealias . Это обеспечивает удобный заполнитель типа, если одна и та же подпись закрытия используется в нескольких местах. Например, общие обратные вызовы сетевых запросов или обработчики событий пользовательского интерфейса делают отличные кандидаты для «имени» с псевдонимом типа.

```
public typealias ClosureType = (x: Int, y: Int) -> Int
```

Затем вы можете определить функцию с помощью typealias:

```
public func closureFunction(closure: ClosureType) {
    let z = closure(1, 2)
}

closureFunction() { (x: Int, y: Int) -> Int in return x + y }
```

Прочитайте Затворы онлайн: https://riptutorial.com/ru/swift/topic/262/затворы

глава 20: Инициализаторы

Examples

Установка значений свойств по умолчанию

Вы можете использовать инициализатор для установки значений свойств по умолчанию:

```
struct Example {
    var upvotes: Int
    init() {
        upvotes = 42
    }
}
let myExample = Example() // call the initializer
print(myExample.upvotes) // prints: 42
```

Или укажите значения свойств по умолчанию в качестве части объявления свойства:

```
struct Example {
    var upvotes = 42 // the type 'Int' is inferred here
}
```

Классы и структуры **должны** установить все сохраненные свойства на соответствующее начальное значение к моменту создания экземпляра. Этот пример не будет компилироваться, потому что инициализатор не дал начального значения для downvotes :

```
struct Example {
    var upvotes: Int
    var downvotes: Int
    init() {
         upvotes = 0
    } // error: Return from initializer without initializing all stored properties
}
```

Настройка инициализации с помощью параметров

```
struct MetricDistance {
    var distanceInMeters: Double

    init(fromCentimeters centimeters: Double) {
        distanceInMeters = centimeters / 100
    }
    init(fromKilometers kilos: Double) {
        distanceInMeters = kilos * 1000
    }
}

let myDistance = MetricDistance(fromCentimeters: 42)
// myDistance.distanceInMeters is 0.42
let myOtherDistance = MetricDistance(fromKilometers: 42)
// myOtherDistance.distanceInMeters is 42000
```

Обратите внимание: вы не можете опустить метки параметров:

```
let myBadDistance = MetricDistance(42) // error: argument labels do not match any available
overloads
```

Чтобы разрешить пропуски меток параметров, используйте знак подчеркивания _ в качестве метки:

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ meters: Double) {
        distanceInMeters = meters
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

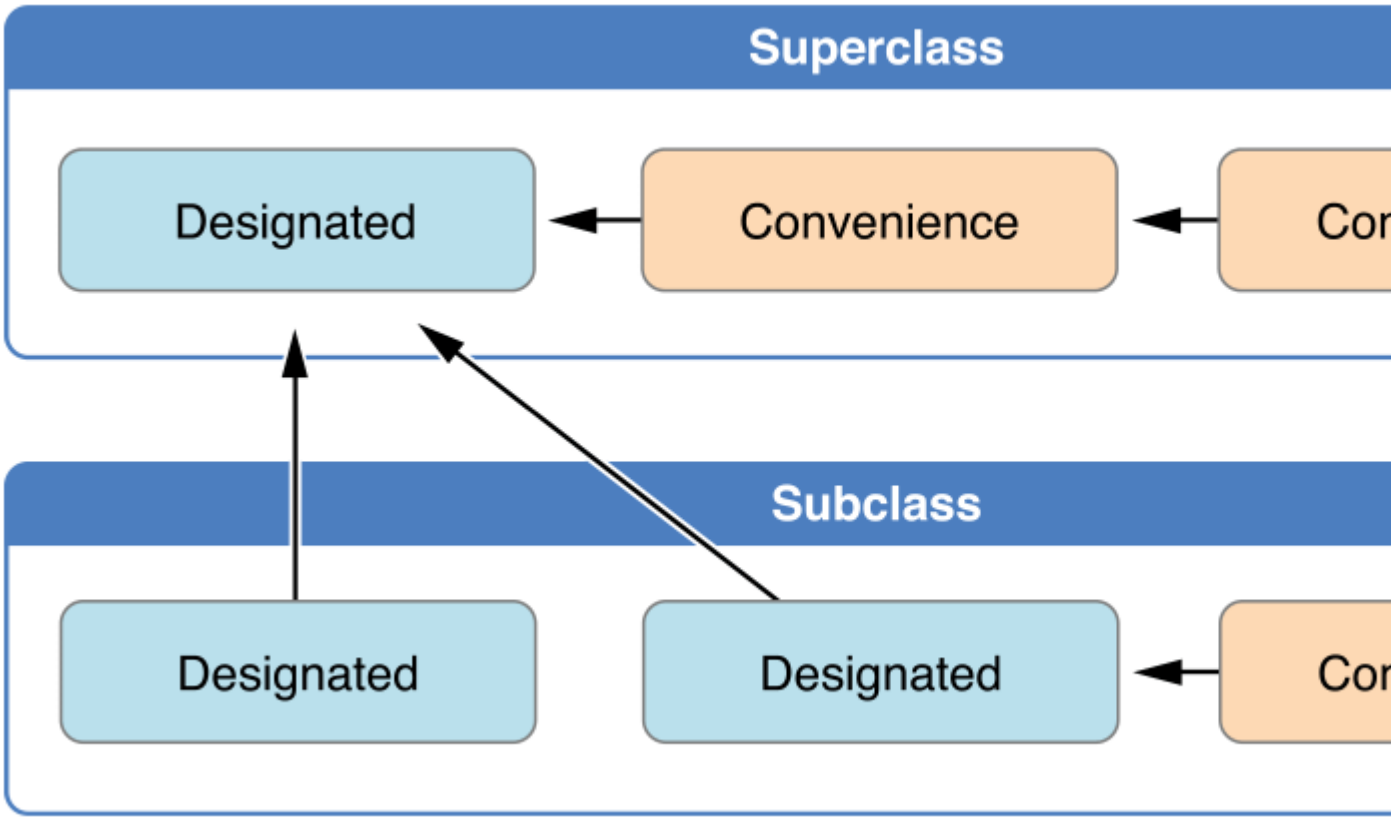
Если ваши метки аргументов обмениваются именами с одним или несколькими свойствами, используйте self чтобы явно установить значения свойств:

```
struct Color {
    var red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
}
```

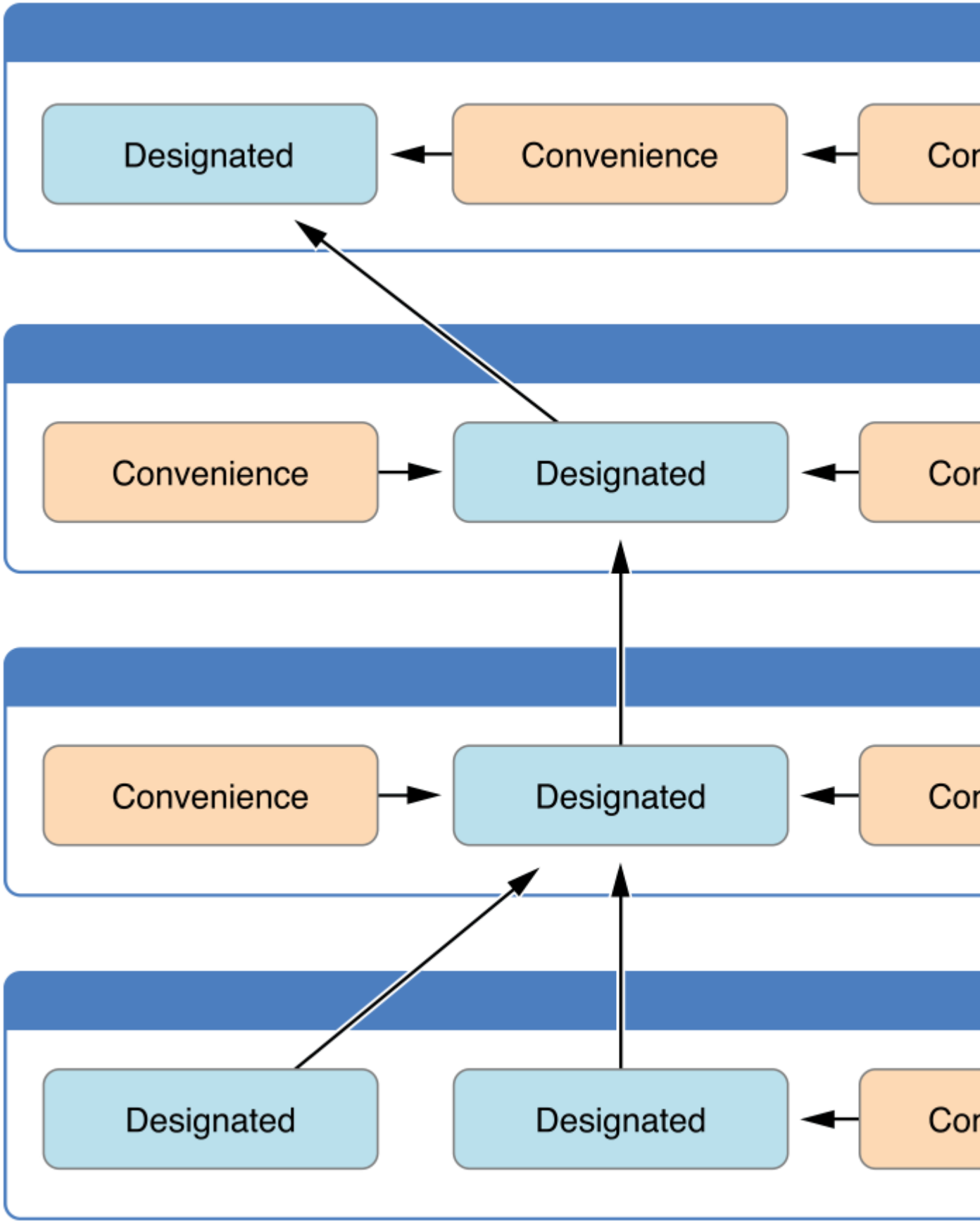
Удобство инициализации

Классы Swift поддерживают несколько способов инициализации. Следуя спецификациям Apple, эти 3 правила должны соблюдаться:

1. Назначенный инициализатор должен вызывать назначенный инициализатор из своего непосредственного суперкласса.



2. Инициализатор удобства должен вызывать другой инициализатор из того же класса.
3. Инициализатор удобства должен в конечном счете вызвать назначенный инициализатор.




```
class Foo {

    var someString: String
    var someValue: Int
    var someBool: Bool
```

```
    // Designated Initializer
    init(someString: String, someValue: Int, someBool: Bool)
    {
        self.someString = someString
        self.someValue = someValue
        self.someBool = someBool
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(otherString: "")
    }

    // A convenience initializer must ultimately call a designated initializer.
convenience init(otherString: String)
    {
        self.init(someString: otherString, someValue:  0, someBool: false)
    }
}


class Baz: Foo
{
    var someFloat: Float

    // Designed initializer
    init(someFloat: Float)
    {
        self.someFloat = someFloat

        // A designated initializer must call a designated initializer from its immediate
superclass.
        super.init(someString: "", someValue: 0, someBool: false)
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(someFloat: 0)
    }
}
```

Назначенный инициализатор

```
let c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

Удобство init ()

```
let a = Foo()
```

Удобство init (otherString: String)

```
let b = Foo(otherString: "Some string")
```

Назначенный инициализатор (вызовет назначенный инициализатор суперкласса)

```
let d = Baz(someFloat: 3)
```

Удобство init ()

```
let e = Baz()
```

Источник изображения: Swift Programming Languag e

Хирургический Инициализатор

Использование обработки ошибок, чтобы сделать инициализатор Struct (или class) в качестве запускаемого инициализатора:

Пример Обработка ошибок:

```
enum ValidationError: Error {
    case invalid
}
```

Вы можете использовать перечисление обработки ошибок, чтобы проверить, что параметр для Struct (или класса) соответствует ожидаемому требованию

```
struct User {
    let name: String

    init(name: String?) throws {

        guard let name = name else {
           ValidationError.invalid
        }

        self.name = name
    }
}
```

Теперь вы можете использовать запустимый инициализатор:

```
do {
   let user = try User(name: "Sample name")

   // success
}
catch ValidationError.invalid {
     // handle error
}
```

Прочитайте Инициализаторы онлайн: https://riptutorial.com/ru/swift/topic/1778/инициализаторы

глава 21: Классы

замечания

init() - это специальный метод в классах, который используется для объявления инициализатора для класса. Более подробную информацию можно найти здесь: Инициализаторы

Examples

Определение класса

Вы определяете класс следующим образом:

```
class Dog {}
```

Класс также может быть подклассом другого класса:

```
class Animal {}
class Dog: Animal {}
```

В этом примере Animal также может быть протоколом, который соответствует Dog .

Справочная семантика

Классы являются **ссылочными типами** , что означает, что несколько переменных могут ссылаться на один и тот же экземпляр.

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"

let otherDog = firstDog  // otherDog points to the same Dog instance
otherDog.name = "Rover"  // modifying otherDog also modifies firstDog

print(firstDog.name)  // prints "Rover"
```

Поскольку классы являются ссылочными типами, даже если класс является константой, его свойства переменной все еще могут быть изменены.

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5)// This instance is a constant.
var variableDog = Dog(name: "Spot", age 7)// This instance is a variable.

constantDog.name = "Fido" // Not an error because name is a variable property.
constantDog.age = 6 // Error because age is a constant property.
constantDog = Dog(name: "Fido", age: 6)
```

```
/* The last one is an error because you are changing the actual reference, not
just what the reference points to. */

variableDog.name = "Ace" // Not an error because name is a variable property.
variableDog.age = 8 // Error because age is a constant property.
variableDog = Dog(name: "Ace", age: 8)
/* The last one is not an error because variableDog is a variable instance and
therefore the actual reference can be changed. */
```

Проверьте, *идентичны* ли два объекта (укажите один и тот же экземпляр) с помощью === :

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// Consider two dogs equal if their names are equal.
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// Create two Dog instances which have the same name.
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2   // true, because the dogs are equal
spot1 != spot2   // false

spot1 === spot2  // false, because the dogs are different instances
spot1 !== spot2  // true
```

Свойства и методы

Классы могут определять свойства, которые могут использовать экземпляры класса. В этом примере у Dog есть два свойства: name и dogYearAge :

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

Вы можете получить доступ к свойствам с помощью синтаксиса точек:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Классы могут также определять методы, которые могут быть вызваны в экземплярах, они объявляются аналогичными нормальным функциям , только внутри класса:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

В методах вызова используется также точечный синтаксис:

```
dog.bark()
```

Классы и множественное наследование

Swift не поддерживает множественное наследование. То есть вы не можете наследовать более одного класса.

```
class Animal { ... }
class Pet { ... }

class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

Вместо этого вам рекомендуется использовать композицию при создании ваших типов. Это можно сделать с помощью протоколов .

Deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```

Прочитайте Классы онлайн: https://riptutorial.com/ru/swift/topic/459/классы

глава 22: Контроль доступа

Синтаксис

- проект частного класса
- let car = Car («Форд», модель: «Escape») // default internal
- public enum Жанр
- private func calculateMarketCap ()
- переопределить внутреннюю функцию func setupView ()
- private (set) var area = 0

замечания

1. ***Основное замечание:***

Ниже приведены три уровня доступа от наивысшего доступа (наименее ограничивающий) до самого низкого доступа (наиболее ограничительного)

Открытый доступ позволяет получить доступ к классам, структурам, переменным и т. Д. Из любого файла в модели, но, что более важно, вне модуля, если внешний файл импортирует модуль, содержащий общедоступный код доступа. Популярно использовать публичный доступ при создании фреймворка.

Внутренний доступ позволяет файлам только с модулем сущностей использовать сущности. По умолчанию все объекты имеют **внутренний** уровень доступа (за некоторыми исключениями).

Частный доступ запрещает использование объекта вне этого файла.

2. ***Подкласс Замечание:***

Подкласс не может иметь более высокий доступ, чем его суперкласс.

3. ***Getter & Setter Примечание:***

Если установщик свойств является приватным, то getter является внутренним (что является значением по умолчанию). Также вы можете назначить уровень доступа как для получателя, так и для сеттера. Эти принципы также применимы и к *индексам*

4. ***Общее замечание:***

Другие типы объектов включают: Инициализаторы, Протоколы, Расширения, Дженерики и Типовые Псевдонимы

Examples

Основной пример с использованием Struct

3.0

В Swift 3 есть несколько уровней доступа. В этом примере используются все, кроме open :

```
public struct Car {

    public let make: String
    let model: String //Optional keyword: will automatically be "internal"
    private let fullName: String
    fileprivate var otherName: String
```

```
    public init(_ make: String, model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
```

Предположим, что myCar был инициализирован следующим образом:

```
let myCar = Car("Apple", model: "iCar")
```

Car.make (общедоступный)

```
print(myCar.make)
```

Эта печать будет работать везде, включая цели, которые импортируют Car .

Car.model (внутренний)

```
print(myCar.model)
```

Это будет скомпилировано, если код находится в той же цели, что и Car .

Car.otherName (fileprivate)

```
print(myCar.otherName)
```

Это будет работать, только если код находится *в том же файле*, что и Car .

Car.fullName (частный)

```
print(myCar.fullName)
```

Это не будет работать в Swift 3. Доступ к private свойствам возможен только в пределах одной и той же struct / class .

```
public struct Car {

    public let make: String       //public
    let model: String             //internal
    private let fullName: String! //private

    public init(_ make: String, model model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
    }
}
```

Если объект имеет несколько связанных уровней доступа, Swift ищет самый низкий уровень доступа. Если частная переменная существует в открытом классе, переменная все равно будет считаться закрытой.

Пример подкласса

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

Пример Getters и Setters

```
struct Square {
    private(set) var area = 0

    var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
}

public struct Square {
    public private(set) var area = 0
    public var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
    public init() {}
}
```

Прочитайте Контроль доступа онлайн: https://riptutorial.com/ru/swift/topic/1075/контроль-доступа

глава 23: Кортеж

Вступление

Тип кортежа представляет собой список типов, разделенных запятыми, заключенных в круглые скобки.

Этот список типов также может иметь имя элементов и использовать эти имена для обозначения значений отдельных элементов.

Имя элемента состоит из идентификатора, за которым сразу следует двоеточие (:).

Общего пользования -

Мы можем использовать тип кортежа в качестве возвращаемого типа функции, чтобы функция возвращала один кортеж, содержащий несколько значений

замечания

Кортежи считаются типами значений. Более подробную информацию о кортежах можно найти в документации:
developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.

Examples

Что такое кортежи?

Кортежи группируют несколько значений в одно составное значение. Значения внутри кортежа могут быть любого типа и не должны быть одного типа друг с другом.

Кортежи создаются путем группировки любых значений:

```
let tuple = ("one", 2, "three")

// Values are read using index numbers starting at zero
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

Также можно указать индивидуальные значения, когда определяется кортеж:

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// Values can be read with the named property
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// And still with the index number
print(namedTuple.2)      // 3
```

Их также можно назвать при использовании в качестве переменной и даже иметь возможность иметь необязательные значения внутри:

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)?

//Later On
numbers = (nil, "dos", 3)

print(numbers.optionalFirst)// nil
```

```
print(numbers.middle)//"dos"
print(numbers.last)//3
```

Разложение на отдельные переменные

Кортежи можно разложить на отдельные переменные со следующим синтаксисом:

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple

print(first) // "Some Name"
print(second)  // 26
```

Этот синтаксис можно использовать независимо от того, имеет ли кортеж неназванные свойства:

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

Специфические свойства можно игнорировать с помощью подчеркивания (_):

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

Кортежи как возвращаемое значение функций

Функции могут возвращать кортежи:

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

Если вы назначаете имена параметров, их можно использовать из возвращаемого значения:

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

Использование typealias для обозначения типа вашего кортежа

Иногда вы можете использовать один и тот же тип кортежа в нескольких местах всего кода. Это может быстро запутаться, особенно если ваш кортеж сложный:

```
// Define a circle tuple by its center point and radius
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0), 1.0)
```

```
func doubleRadius(ofCircle circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) ->
(center: (x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

Если вы используете определенный тип кортежа в нескольких местах, вы можете использовать ключевое слово typealias чтобы назвать тип вашего кортежа.

```
// Define a circle tuple by its center point and radius
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)

let unitCircle: Circle = ((0.0, 0.0), 1)

func doubleRadius(ofCircle circle: Circle) -> Circle {
    // Aliased tuples also have access to value labels in the original tuple type.
    return (circle.center, circle.radius * 2.0)
}
```

Однако, если вы слишком часто это делаете, вам следует рассмотреть возможность использования struct .

Обмен значениями

Кортежи полезны для обмена значениями между 2 (или более) переменными без использования временных переменных.

Пример с 2 переменными

Учитывая 2 переменные

```
var a = "Marty McFly"
var b = "Emmett Brown"
```

мы можем легко менять значения

```
(a, b) = (b, a)
```

Результат:

```
print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Пример с 4 переменными

```
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

Кортежи как случай в коммутаторе

Использовать кортежи в переключателе

```
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
 case (true, false):
    // do something
 case (true, true):
    // do something
 case (false, true):
    // do something
 case (false, false):
    // do something
}
```

Или в сочетании с Enum Например, с классами размера:

```
    let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

    switch switchTuple {
    case (.Regular, .Compact):
        //statement
    case (.Regular, .Regular):
        //statement
    case (.Compact, .Regular):
        //statement
    case (.Compact, .Compact):
        //statement
    }
```

Прочитайте Кортеж онлайн: https://riptutorial.com/ru/swift/topic/574/кортеж

глава 24: Криптографическое Хеширование

Examples

MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

Эти функции будут хешировать либо String, либо Data input с одним из восьми криптографических алгоритмов хеширования.

Параметр name указывает имя хэш-функции как строку
Поддерживаемые функции: MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384 и SHA512

Этот пример требует Common Crypto
Необходимо иметь заголовок заголовка проекта:
#import <CommonCrypto/CommonCrypto.h>
Добавьте в проект Security.framework.

Эта функция принимает хеш-имя и данные для хэширования и возвращает данные:

```
name: A name of a hash function as a String
data: The Data to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, data:Data) -> Data? {
    let algos = ["MD2":    (CC_MD2,    CC_MD2_DIGEST_LENGTH),
                 "MD4":    (CC_MD4,    CC_MD4_DIGEST_LENGTH),
                 "MD5":    (CC_MD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA1":   (CC_SHA1,   CC_SHA1_DIGEST_LENGTH),
                 "SHA224": (CC_SHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (CC_SHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (CC_SHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (CC_SHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[name]  else { return nil }
    var hashData = Data(count: Int(length))

    _ = hashData.withUnsafeMutableBytes {digestBytes in
        data.withUnsafeBytes {messageBytes in
            hashAlgorithm(messageBytes, CC_LONG(data.count), digestBytes)
        }
    }
    return hashData
}
```

Эта функция принимает хеш-имя и String для хэширования и возвращает Data:

```
name: A name of a hash function as a String
string: The String to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, string:String) -> Data? {
    let data = string.data(using:.utf8)!
    return hash(name:name, data:data)
}
```

Примеры:

```
let clearString = "clearData0123456"
let clearData   = clearString.data(using:.utf8)!
print("clearString: \(clearString)")
print("clearData: \(clearData as NSData)")

let hashSHA256 = hash(name:"SHA256", string:clearString)
print("hashSHA256: \(hashSHA256! as NSData)")

let hashMD5 = hash(name:"MD5", data:clearData)
print("hashMD5: \(hashMD5! as NSData)")
```

Выход:

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```

HMAC с MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

Эти функции будут хешировать либо String, либо Data input с одним из восьми криптографических алгоритмов хеширования.

Параметр name указывает имя хэш-функции в качестве поддерживаемых String функций: MD5, SHA1, SHA224, SHA256, SHA384 и SHA512

Этот пример требует Common Crypto
Необходимо иметь заголовок заголовка проекта:
#import <CommonCrypto/CommonCrypto.h>
Добавьте в проект Security.framework.

Эти функции принимают имя хеша, сообщение хэширования, ключ и возвращают дайджест:

```
hashName: name of a hash function as String
message:  message as Data
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:Data, key:Data) -> Data? {
    let algos = ["SHA1":   (kCCHmacAlgSHA1,   CC_SHA1_DIGEST_LENGTH),
                 "MD5":    (kCCHmacAlgMD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA224": (kCCHmacAlgSHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (kCCHmacAlgSHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (kCCHmacAlgSHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (kCCHmacAlgSHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[hashName]  else { return nil }
    var macData = Data(count: Int(length))

    macData.withUnsafeMutableBytes {macBytes in
        message.withUnsafeBytes {messageBytes in
            key.withUnsafeBytes {keyBytes in
                CCHmac(CCHmacAlgorithm(hashAlgorithm),
                       keyBytes,     key.count,
                       messageBytes, message.count,
                       macBytes)
            }
```

```
        }
    }
    return macData
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as String
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:String) -> Data? {
    let messageData = message.data(using:.utf8)!
    let keyData = key.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:keyData)
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:Data) -> Data? {
    let messageData = message.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:key)
}
```

```
// Примеры

let clearString = "clearData0123456"
let keyString   = "keyData8901234562"
let clearData   = clearString.data(using:.utf8)!
let keyData     = keyString.data(using:.utf8)!
print("clearString: \(clearString)")
print("keyString:   \(keyString)")
print("clearData: \(clearData as NSData)")
print("keyData:   \(keyData as NSData)")

let hmacData1 = hmac(hashName:"SHA1", message:clearData, key:keyData)
print("hmacData1: \(hmacData1! as NSData)")

let hmacData2 = hmac(hashName:"SHA1", message:clearString, key:keyString)
print("hmacData2: \(hmacData2! as NSData)")

let hmacData3 = hmac(hashName:"SHA1", message:clearString, key:keyData)
print("hmacData3: \(hmacData3! as NSData)")
```

Выход:

```
clearString: clearData0123456
keyString:   keyData8901234562
clearData: <636c6561 72446174 61303132 33343536>
keyData:   <6b657944 61746138 39303132 33343536 32>

hmacData1: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
hmacData2: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

```
hmacData3: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

Прочитайте Криптографическое Хеширование онлайн: https://riptutorial.com/ru/swift/topic/7885/криптографическое-хеширование

глава 25: Кэширование на диске

Вступление

Кэширование видео, изображений и аудио с помощью URLSession и FileManager

Examples

экономия

```
let url = "https://path-to-media"
let request = URLRequest(url: url)
let downloadTask = URLSession.shared.downloadTask(with: request) { (location, response, error)
in
  guard let location = location,
        let response = response,
        let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).first else {
    return
  }
  let documentsDirectoryUrl = URL(fileURLWithPath: documentsPath)
  let documentUrl = documentsDirectoryUrl.appendingPathComponent(response.suggestedFilename)
  let _ = try? FileManager.default.moveItem(at: location, to: documentUrl)

  // documentUrl is the local URL which we just downloaded and saved to the FileManager
}.resume()
```

чтение

```
let url = "https://path-to-media"
guard let documentsUrl = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first,
      let searchQuery = url.absoluteString.components(separatedBy: "/").last else {
  return nil
}

do {
  let directoryContents = try FileManager.default.contentsOfDirectory(at: documentsUrl,
includingPropertiesForKeys: nil, options: [])
  let cachedFiles = directoryContents.filter { $0.absoluteString.contains(searchQuery) }

  // do something with the files found by the url
} catch {
  // Could not find any files
}
```

Прочитайте Кэширование на диске онлайн: https://riptutorial.com/ru/swift/topic/8902/кэширование-на-диске

глава 26: Литье под давлением

Синтаксис

- let name = json["name"] as? String ?? "" // Выход: john
- let name = json["name"] as? String // Output: Optional("john")
- let name = rank as? Int // Output: Optional(1)
- let name = rank as? Int ?? 0 // Output: 1
- let name = dictionary as? [String: Any] ?? [:] // Output: ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]]

Examples

понижающее приведение

Переменная может быть понижена до подтипа с использованием *операторов типа литья* as? , и as! ,

Как as? оператор *пытается* применить к подтипу.
Он может выйти из строя, поэтому он возвращает необязательный.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

Как as! оператор *заставляет* бросок.
Он не возвращает необязательный, но сбой, если сбой выполняется.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value…"
```

Обычно используются операторы типа cast с условной разверткой:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
    print(age) // Not executed
}
```

Кастинг с переключателем

Оператор switch также может использоваться для попытки кастинга в разные типы:

```
func checkType(_ value: Any) -> String {
    switch value {

    // The `is` operator can be used to check a type
    case is Double:
        return "value is a Double"

    // The `as` operator will cast. You do not need to use `as?` in a `switch`.
    case let string as String:
        return "value is the string: \(string)"

    default:
        return "value is something else"
    }

}

checkType("Cadena")  // "value is the string: Cadena"
checkType(6.28)      // "value is a Double"
checkType(UILabel()) // "value is something else"
```

Приведение к базовому типу

Оператор as будет перенаправлен на супертип. Поскольку это не может быть неудачно, оно не возвращает необязательный.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Пример использования downcast для параметра функции, включающего подклассирование

С помощью понижающего преобразования можно использовать код и данные подкласса внутри функции, принимающей параметр ее суперкласса.

```
class Rat {
    var color = "white"
}

class PetRat: Rat {
    var name = "Spot"
}

func nameOfRat( : Rat) -> String {
    guard let petRat = ( as? PetRat) else {
        return "No name"
    }

    return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

Тип литья в Swift Language

Литье под давлением

Тип casting - это способ проверить тип экземпляра или обработать этот экземпляр как другой суперкласс или подкласс из другого места в его собственной иерархии классов.

Тип casting в Swift реализуется с помощью операторов is и as. Эти два оператора предоставляют простой и выразительный способ проверить тип значения или присвоить значение другому типу.

понижающее приведение

Константа или переменная определенного типа класса может фактически ссылаться на экземпляр подкласса за кулисами. Если вы считаете, что это так, вы можете попытаться спуститься к типу подкласса с оператором литья типа (как? Или как!).

Поскольку downcasting может выйти из строя, оператор литого типа работает в двух разных формах. Условная форма, как ?, возвращает необязательное значение типа, который вы пытаетесь сбрасывать. Принудительная форма, как !, пытается понизить и принудительно развернуть результат как одно составное действие.

Используйте условную форму оператора литья типа (как?), Если вы не уверены, что преуменьшение будет успешным. Эта форма оператора всегда будет возвращать необязательное значение, и значение будет равно нулю, если нисходящее значение невозможно. Это позволяет проверить успешное нажатие.

Используйте форсированную форму оператора типа cast (as!), Только если вы уверены, что downcast всегда будет успешным. Эта форма оператора вызовет ошибку времени выполнения, если вы попытаетесь сбрасывать неправильный тип класса. Узнать больше.

Преобразование строк в Int & Float: -

```
     let numbers = "888.00"
     let intValue = NSString(string: numbers).integerValue
     print(intValue) // Output - 888


     let numbers = "888.00"
     let floatValue = NSString(string: numbers).floatValue
     print(floatValue) // Output : 888.0
```

Преобразование Float to String

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue) // Output : 888.0

    // Get Float value at particular decimal point
    let numbers = 888.00
    let floatValue = String(format: "%.2f", numbers) // Here %.2f will give 2 numbers after
decimal points we can use as per our need
    print(floatValue) // Output : "888.00"
```

Целое число к значению строки

```
    let numbers = 888
```

```
    let intValue = String(numbers)
    print(intValue) // Output : "888"
```

Значение Float to String

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue)
```

Необязательное значение Float для String

```
    let numbers: Any = 888.00
    let floatValue = String(describing: numbers)
    print(floatValue) // Output : 888.0
```

Необязательное значение String to Int

```
    let hitCount = "100"
    let data :AnyObject = hitCount
    let score = Int(data as? String ?? "") ?? 0
    print(score)
```

Значения Downcasting от JSON

```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Значения Downcasting из дополнительного JSON

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Управление JSON Response с условиями

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //Optional Response

    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
```

```
}
let name = json["name"] as? String ?? ""
print(name) // Output : john
let subjects = json["subjects"] as? [String] ?? []
print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Управление Nil Response с условием

```
let response: Any? = nil
guard let json = response as? [String: Any] else {
    // Handle here nil value
    print("Empty Dictionary")
    // Do something here
    return
}
let name = json["name"] as? String ?? ""
print(name)
let subjects = json["subjects"] as? [String] ?? []
print(subjects)
```

Выход: **Empty Dictionary**

Прочитайте Литье под давлением онлайн: https://riptutorial.com/ru/swift/topic/3082/литье-под-давлением

глава 27: Массивы

Вступление

Массив - это упорядоченный тип сбора данных с произвольным доступом. Массивы являются одним из наиболее часто используемых типов данных в приложении. Мы используем тип Array для хранения элементов одного типа, типа элемента Element. Массив может хранить любые элементы - от целых чисел до строк до классов.

Синтаксис

- Array <Element> // Тип массива с элементами типа Element
- [Элемент] // Синтаксический сахар для типа массива с элементами типа Element
- [element0, element1, element2, ... elementN] // Литерал массива
- [Element] () // Создает новый пустой массив типа [Element]
- Array (count: repeatValue :) // Создает массив элементов count , каждый из которых инициализируется repeatedValue
- Array (_ :) // Создает массив из произвольной последовательности

замечания

Массивы представляют собой *упорядоченный набор* значений. Значения могут повторяться, но *должны* быть одного типа.

Examples

Ценностная семантика

Копирование массива копирует все элементы внутри исходного массива.

Изменение нового массива *не изменит* исходный массив.

```
var originalArray = ["Swift", "is", "great!"]
var newArray = originalArray
newArray[2] = "awesome!"
//originalArray = ["Swift", "is", "great!"]
//newArray = ["Swift", "is", "awesome!"]
```

Скопированные массивы будут использовать то же пространство в памяти, что и оригинал, до тех пор, пока они не будут изменены. В результате этого происходит поражение производительности, когда скопированному массиву присваивается собственное пространство в памяти, поскольку оно изменяется впервые.

Основы массивов

Array - это упорядоченный тип коллекции в стандартной библиотеке Swift. Он обеспечивает O (1) случайный доступ и динамическое перераспределение. Массив - это общий тип , поэтому тип значений, которые он содержит, известен во время компиляции.

Поскольку Array является типом значения , его изменчивость определяется тем, является ли он аннотированным как var (mutable) или let (immutable).

Тип [Int] (значение: массив, содержащий Int s) является синтаксическим сахаром для Array<T> .

Узнайте больше о массивах на языке Swift Programming .

Пустые массивы

Следующие три объявления эквивалентны:

```
// A mutable array of Strings, initially empty.

var arrayOfStrings: [String] = []      // type annotation + array literal
var arrayOfStrings = [String]()        // invoking the [String] initializer
var arrayOfStrings = Array<String>()   // without syntactic sugar
```

Литералы массива

Литерал массива записывается с квадратными скобками, окружающими элементы, разделенные запятыми:

```
// Create an immutable array of type [Int] containing 2, 4, and 7
let arrayOfInts = [2, 4, 7]
```

Компилятор обычно может вывести тип массива на основе элементов в литерале, но явные **аннотации типов** могут переопределять значение по умолчанию:

```
let arrayOfUInt8s: [UInt8] = [2, 4, 7]   // type annotation on the variable
let arrayOfUInt8s = [2, 4, 7] as [UInt8] // type annotation on the initializer expression
let arrayOfUInt8s = [2 as UInt8, 4, 7]   // explicit for one element, inferred for the others
```

Массивы с повторяющимися значениями

```
// An immutable array of type [String], containing ["Example", "Example", "Example"]
let arrayOfStrings = Array(repeating: "Example",count: 3)
```

Создание массивов из других последовательностей

```
let dictionary = ["foo" : 4, "bar" : 6]

// An immutable array of type [(String, Int)], containing [("bar", 6), ("foo", 4)]
let arrayOfKeyValuePairs = Array(dictionary)
```

Многомерные массивы

В Swift многомерный массив создается массивами вложенности: двумерный массив Int - [[Int]] (или Array<Array<Int>>).

```
let array2x3 = [
    [1, 2, 3],
    [4, 5, 6]
]
// array2x3[0][1] is 2, and array2x3[1][2] is 6.
```

Чтобы создать многомерный массив повторяющихся значений, используйте вложенные вызовы инициализатора массива:

```
var array3x4x5 = Array(repeating: Array(repeating: Array(repeating: 0,count: 5),count:
4),count: 3)
```

Доступ к значениям массива

Следующие примеры будут использовать этот массив для демонстрации доступа к значениям

```
var exampleArray:[Int] = [1,2,3,4,5]
//exampleArray = [1, 2, 3, 4, 5]
```

Чтобы получить доступ к значению в известном индексе, используйте следующий синтаксис:

```
let exampleOne = exampleArray[2]
//exampleOne = 3
```

Примечание . Значение в *индексе два является третьим значением* в Array . Array s использует *индекс на основе нуля*, что означает, что первый элемент в Array имеет индекс 0.

```
let value0 = exampleArray[0]
let value1 = exampleArray[1]
let value2 = exampleArray[2]
let value3 = exampleArray[3]
let value4 = exampleArray[4]
//value0 = 1
//value1 = 2
//value2 = 3
//value3 = 4
//value4 = 5
```

Доступ к подмножеству Array с использованием фильтра:

```
var filteredArray = exampleArray.filter({ $0 < 4 })
//filteredArray = [1, 2, 3]
```

Фильтры могут иметь сложные условия, такие как фильтрация только четных чисел:

```
var evenArray = exampleArray.filter({ $0 % 2 == 0 })
//evenArray = [2, 4]
```

Также можно вернуть индекс заданного значения, возвращая nil если значение не было найдено.

```
exampleArray.indexOf(3) // Optional(2)
```

Существуют методы для первого, последнего, максимального или минимального значения в Array . Эти методы возвратят nil если Array пуст.

```
exampleArray.first // Optional(1)
exampleArray.last // Optional(5)
exampleArray.maxElement() // Optional(5)
exampleArray.minElement() // Optional(1)
```

Полезные методы

Определить, пуст ли массив или вернуть его размер

```
var exampleArray = [1,2,3,4,5]
exampleArray.isEmpty //false
exampleArray.count //5
```

Reverse a Array **Примечание. Результат не выполняется в массиве, на который вызывается метод, и должен быть помещен в его собственную переменную.**

```
exampleArray = exampleArray.reverse()
```

```
//exampleArray = [9, 8, 7, 6, 5, 3, 2]
```

Изменение значений в массиве

Существует несколько способов добавления значений в массив

```
var exampleArray = [1,2,3,4,5]
exampleArray.append(6)
//exampleArray = [1, 2, 3, 4, 5, 6]
var sixOnwards = [7,8,9,10]
exampleArray += sixOnwards
//exampleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

и удалить значения из массива

```
exampleArray.removeAtIndex(3)
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9, 10]
exampleArray.removeLast()
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9]
exampleArray.removeFirst()
//exampleArray = [2, 3, 5, 6, 7, 8, 9]
```

Сортировка массива

```
var array = [3, 2, 1]
```

Создание нового отсортированного массива

Поскольку Array соответствует SequenceType , мы можем создать новый массив отсортированных элементов, используя встроенный метод сортировки.

2.1 2.2

В Swift 2 это делается с помощью метода sort() .

```
let sorted = array.sort()  // [1, 2, 3]
```

3.0

С Swift 3 он был переименован в sorted() .

```
let sorted = array.sorted()  // [1, 2, 3]
```

Сортировка существующего массива на месте

Поскольку Array соответствует MutableCollectionType , мы можем сортировать его элементы на месте.

2.1 2.2

В Swift 2 это выполняется с помощью sortInPlace() .

```
array.sortInPlace() // [1, 2, 3]
```

3.0

С Swift 3 он был переименован в sort() .

```
array.sort() // [1, 2, 3]
```

> Примечание. Чтобы использовать вышеуказанные методы, элементы должны соответствовать протоколу Comparable .

Сортировка массива с пользовательским заказом

Вы также можете отсортировать массив, используя закрытие, чтобы определить, должен ли один элемент быть заказан другим, - который не ограничивается массивами, где элементы должны быть Comparable . Например, для Landmark не Comparable но вы все равно можете отсортировать массив ориентиров по высоте или имени.

```
struct Landmark {
    let name : String
    let metersTall : Int
}

var landmarks = [Landmark(name: "Empire State Building", metersTall: 443),
                 Landmark(name: "Eifell Tower", metersTall: 300),
                 Landmark(name: "The Shard", metersTall: 310)]
```

2.1 2.2

```
// sort landmarks by height (ascending)
landmarks.sortInPlace {$0.metersTall < $1.metersTall}

print(landmarks) // [Landmark(name: "Eifell Tower", metersTall: 300), Landmark(name: "The
Shard", metersTall: 310), Landmark(name: "Empire State Building", metersTall: 443)]

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sort {$0.name < $1.name}

print(alphabeticalLandmarks) // [Landmark(name: "Eifell Tower", metersTall: 300),
Landmark(name: "Empire State Building", metersTall: 443), Landmark(name: "The Shard",
metersTall: 310)]
```

3.0

```
// sort landmarks by height (ascending)
landmarks.sort {$0.metersTall < $1.metersTall}

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sorted {$0.name < $1.name}
```

> Примечание. Сравнение строк может дать неожиданные результаты, если строки непоследовательны, см. Раздел Сортировка массива строк .

Преобразование элементов массива с помощью карты (_ :)

Поскольку Array соответствует SequenceType , мы можем использовать map(_:) для преобразования массива A в массив из B используя замыкание типа (A) throws -> B

Например, мы могли бы использовать его для преобразования массива Int s в массив String s следующим образом:

```
let numbers = [1, 2, 3, 4, 5]
let words = numbers.map { String($0) }
print(words) // ["1", "2", "3", "4", "5"]
```

map(_:) будет проходить через массив, применяя данное замыкание к каждому элементу. Результат этого закрытия будет использован для заполнения нового массива преобразованными элементами.

Поскольку String имеет инициализатор, который получает Int мы также можем использовать этот более четкий синтаксис:

```
let words = numbers.map(String.init)
```

Преобразование map(_:) не должно изменять тип массива - например, оно также может использоваться для умножения массива Int s на два:

```
let numbers = [1, 2, 3, 4, 5]
let numbersTimes2 = numbers.map {$0 * 2}
print(numbersTimes2) // [2, 4, 6, 8, 10]
```

Извлечение значений данного типа из массива с помощью flatMap (_ :)

things Array содержат значения Any типа.

```
let things: [Any] = [1, "Hello", 2, true, false, "World", 3]
```

Мы можем извлечь значения данного типа и создать новый массив этого конкретного типа. Предположим, мы хотим извлечь все Int(s) и поместить их в Int Array безопасным способом.

```
let numbers = things.flatMap { $0 as? Int }
```

Теперь numbers определяются как [Int] . Функция flatMap отбрасывает все элементы nil и результат, таким образом, содержит только следующие значения:

```
[1, 2, 3]
```

Фильтрация массива

Вы можете использовать метод filter(_:) для SequenceType , чтобы создать новый массив, содержащий элементы последовательности, которые удовлетворяют заданному предикату, который может быть предоставлен как закрытие .

Например, фильтрация четных чисел из [Int] :

```
let numbers = [22, 41, 23, 30]

let evenNumbers = numbers.filter { $0 % 2 == 0 }

print(evenNumbers)  // [22, 30]
```

Фильтрация [Person] , где их возраст меньше 30:

```
struct Person {
    var age : Int
}

let people = [Person(age: 22), Person(age: 41), Person(age: 23), Person(age: 30)]

let peopleYoungerThan30 = people.filter { $0.age < 30 }

print(peopleYoungerThan30) // [Person(age: 22), Person(age: 23)]
```

Фильтрация нуля из преобразования массива с помощью flatMap (_ :)

Вы можете использовать flatMap(_:) аналогичным образом, чтобы map(_:) , чтобы создать массив, применив преобразование к элементам последовательности.

```
extension SequenceType {
    public func flatMap<T>(@noescape transform: (Self.Generator.Element) throws -> T?)
rethrows -> [T]
}
```

Разница с этой версией flatMap(_:) заключается в том, что она ожидает, что замыкание преобразования вернет необязательное значение T? для каждого из элементов. Затем он безопасно разворачивает каждое из этих необязательных значений, отфильтровывая nil - в результате получается массив из [T] .

Например, вы можете это сделать, чтобы преобразовать [String] в [Int] используя инициализатор String инициализирующий Int , и отфильтровывать любые элементы, которые не могут быть преобразованы:

```
let strings = ["1", "foo", "3", "4", "bar", "6"]

let numbersThatCanBeConverted = strings.flatMap { Int($0) }

print(numbersThatCanBeConverted) // [1, 3, 4, 6]
```

Вы также можете использовать flatMap(_:) , чтобы отфильтровать nil , чтобы просто преобразовать массив необязательных элементов в массив не-опций:

```
let optionalNumbers : [Int?] = [nil, 1, nil, 2, nil, 3]

let numbers = optionalNumbers.flatMap { $0 }

print(numbers) // [1, 2, 3]
```

Подписывание массива с диапазоном

Можно выделить ряд последовательных элементов из массива с использованием диапазона.

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let range = 2...4
let slice = words[range] // ["Bonjour", "Welcome", "Hi"]
```

Subscripting Array с диапазоном возвращает ArraySlice . Это подпоследовательность массива.

В нашем примере у нас есть массив строк, поэтому мы возвращаем ArraySlice<String> .

Хотя ArraySlice соответствует CollectionType и может использоваться с sort , filter и т. Д., Его назначение не для долговременного хранения, а для временных вычислений: оно должно быть преобразовано обратно в массив, как только вы закончите работать с ним.

Для этого используйте инициализатор Array() :

```
let result = Array(slice)
```

Подводя итог на простом примере без промежуточных шагов:

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let selectedWords = Array(words[2...4]) // ["Bonjour", "Welcome", "Hi"]
```

Группировка значений массива

Если у нас есть такая структура

```
struct Box {
    let name: String
    let thingsInside: Int
}
```

и массив Box(es)

```
let boxes = [
    Box(name: "Box 0", thingsInside: 1),
    Box(name: "Box 1", thingsInside: 2),
    Box(name: "Box 2", thingsInside: 3),
    Box(name: "Box 3", thingsInside: 1),
    Box(name: "Box 4", thingsInside: 2),
    Box(name: "Box 5", thingsInside: 3),
    Box(name: "Box 6", thingsInside: 1)
]
```

мы можем сгруппировать ящики с thingsInside свойства thingsInside , чтобы получить Dictionary котором key - это количество вещей, а значение - это массив ящиков.

```
let grouped = boxes.reduce([Int:[Box]]()) { (res, box) -> [Int:[Box]] in
    var res = res
    res[box.thingsInside] = (res[box.thingsInside] ?? []) + [box]
    return res
}
```

Теперь сгруппировано [Int:[Box]] и имеет следующий контент

```
[
    2: [Box(name: "Box 1", thingsInside: 2), Box(name: "Box 4", thingsInside: 2)],
    3: [Box(name: "Box 2", thingsInside: 3), Box(name: "Box 5", thingsInside: 3)],
    1: [Box(name: "Box 0", thingsInside: 1), Box(name: "Box 3", thingsInside: 1), Box(name:
"Box 6", thingsInside: 1)]
]
```

Сглаживание результата преобразования массива с помощью flatMap (_ :)

Помимо возможности создания массива путем фильтрации nil из преобразованных элементов последовательности, существует также версия flatMap(_:) которая ожидает, что замыкание преобразования вернет последовательность S

```
extension SequenceType {
    public func flatMap<S : SequenceType>(transform: (Self.Generator.Element) throws -> S)
rethrows -> [S.Generator.Element]
}
```

Каждая последовательность из преобразования будет конкатенирована, в результате получается массив, содержащий объединенные элементы каждой последовательности - [S.Generator.Element] .

Объединение символов в массив строк

Например, мы можем использовать его, чтобы взять массив простых строк и объединить их символы в один массив:

```
let primes = ["2", "3", "5", "7", "11", "13", "17", "19"]
let allCharacters = primes.flatMap { $0.characters }
// => "["2", "3", "5", "7", "1", "1", "1", "3", "1", "7", "1", "9"]"
```

Прерывая приведенный выше пример:

1. primes - [String] (Поскольку массив представляет собой последовательность, мы можем вызвать flatMap(_:) на нем).
2. Закрытие преобразования принимает один из элементов primes , String (Array<String>.Generator.Element).
3. Затем замыкание возвращает последовательность типа String.CharacterView .
4. Результатом является массив, содержащий объединенные элементы всех последовательностей от каждого из вызовов замыкания трансформации - [String.CharacterView.Generator.Element] .

Сглаживание многомерного массива

Поскольку flatMap(_:) будет конкатенировать последовательности, возвращаемые из вызовов замыкания преобразования, его можно использовать для выравнивания многомерного массива, такого как 2D-массив, в 1D-массив, трехмерный массив в 2D-массив и т. Д.

Это можно просто сделать, возвратив данный элемент $0 (вложенный массив) в замыкание:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A 1D array of type [Int]
let flattenedArray = array2D.flatMap { $0 }

print(flattenedArray) // [1, 3, 4, 6, 8, 10, 11]
```

Сортировка массива строк

3.0

Самый простой способ - использовать sorted() :

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted()
print(sortedWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

или sort()

```
var mutableWords = ["Hello", "Bonjour", "Salute", "Ahola"]
mutableWords.sort()
print(mutableWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

Вы можете передать закрытие в качестве аргумента для сортировки:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted(isOrderedBefore: { $0 > $1 })
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Альтернативный синтаксис с закрывающейся крышкой:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted() { $0 > $1 }
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Но будут неожиданные результаты, если элементы в массиве несовместимы:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let unexpected = words.sorted()
print(unexpected) // ["Hello", "Salute", "ahola", "bonjour"]
```

Чтобы решить эту проблему, выполните сортировку по строчной версии элементов:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.lowercased() < $1.lowercased() }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Или import Foundation и используйте методы сравнения NSString, такие как caseInsensitiveCompare :

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.caseInsensitiveCompare($1) == .orderedAscending }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Кроме того, используйте localizedCaseInsensitiveCompare , который может управлять диакритикой.

Чтобы правильно отсортировать строки по их *числовому* значению, используйте команду compare с опцией .numeric :

```
let files = ["File-42.txt", "File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt"]
let sortedFiles = files.sorted() { $0.compare($1, options: .numeric) == .orderedAscending }
print(sortedFiles) // ["File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt", "File-
42.txt"]
```

Ленецкое сглаживание многомерного массива с плоским ()

Мы можем использовать flatten() , чтобы лениво уменьшить вложенность многомерной последовательности.

Например, ленивое выравнивание 2D-массива в 1D-массив:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A FlattenBidirectionalCollection<[[Int]]>
let lazilyFlattenedArray = array2D.flatten()

print(lazilyFlattenedArray.contains(4)) // true
```

В приведенном выше примере flatten() вернет FlattenBidirectionalCollection , который будет лениво применять сглаживание массива. Поэтому contains(_:) будет требовать array2D первых двух вложенных массивов array2D поскольку он будет array2D при поиске нужного элемента.

Объединение элементов массива с уменьшением (_: объединение :)

reduce(_:combine:) можно использовать для объединения элементов последовательности в одно значение. Он принимает начальное значение для результата, а также замыкание, применяемое к каждому элементу, - которое вернет новое накопленное значение.

Например, мы можем использовать его для суммирования массива чисел:

```
let numbers = [2, 5, 7, 8, 10, 4]

let sum = numbers.reduce(0) {accumulator, element in
    return accumulator + element
```

```
}

print(sum) // 36
```

Мы передаем 0 в начальное значение, так как это логическое начальное значение для суммирования. Если бы мы передали значение N , то полученная sum бы равна N + 36 . Закрытие, переданное для reduce имеет два аргумента. accumulator - это текущее накопленное значение, которому присваивается значение, возвращаемое замыканием на каждой итерации. element - текущий элемент итерации.

Как и в этом примере, мы reduce (Int, Int) -> Int замыкание, которое просто выводит добавление двух входов - мы можем фактически передавать оператор + непосредственно, поскольку операторы являются функциями в Swift:

```
let sum = numbers.reduce(0, combine: +)
```

Удаление элемента из массива без знания его индекса

Как правило, если мы хотим удалить элемент из массива, нам нужно знать его индекс, чтобы мы могли легко его remove(at:) используя remove(at:) function.

Но что, если мы не знаем индекс, но знаем значение элемента, который нужно удалить!

Итак, вот простое расширение массива, которое позволит нам легко удалить элемент из массива, не зная его индекса:

Swift3

```
extension Array where Element: Equatable {

    mutating func remove(_ element: Element) {
        _ = index(of: element).flatMap {
            self.remove(at: $0)
        }
    }
}
```

например

```
    var array = ["abc", "lmn", "pqr", "stu", "xyz"]
    array.remove("lmn")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]

    array.remove("nonexistent")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]
    //if provided element value is not present, then it will do nothing!
```

Также, если по ошибке мы сделали что-то вроде этого: array.remove(25) то есть мы предоставили значение с другим типом данных, компилятор будет вызывать ошибку,
cannot convert value to expected argument type

Поиск минимального или максимального элемента массива

2.1 2.2

Вы можете использовать `minElement()` и `maxElement()` чтобы найти минимальный или максимальный элемент в заданной последовательности. Например, с массивом чисел:

```
let numbers = [2, 6, 1, 25, 13, 7, 9]

let minimumNumber = numbers.minElement() // Optional(1)
let maximumNumber = numbers.maxElement() // Optional(25)
```

3.0

Начиная с Swift 3, методы были переименованы в min() и max() соответственно:

```
let minimumNumber = numbers.min() // Optional(1)
let maximumNumber = numbers.max() // Optional(25)
```

Возвращаемые значения из этих методов являются необязательными, чтобы отражать тот факт, что массив может быть пустым – если это так, возвращается nil .

> Примечание. Вышеупомянутые методы требуют, чтобы элементы соответствовали протоколу Comparable .

Поиск минимального или максимального элемента с пользовательским заказом

Вы также можете использовать вышеуказанные методы с пользовательским закрытием , определяя, должен ли один элемент быть заказан другим, позволяя вам найти минимальный или максимальный элемент в массиве, где элементы не обязательно Comparable .

Например, с массивом векторов:

```
struct Vector2 {
    let dx : Double
    let dy : Double

    var magnitude : Double {return sqrt(dx*dx+dy*dy)}
}

let vectors = [Vector2(dx: 3, dy: 2), Vector2(dx: 1, dy: 1), Vector2(dx: 2, dy: 2)]
```

2.1 2.2

```
// Vector2(dx: 1.0, dy: 1.0)
let lowestMagnitudeVec2 = vectors.minElement { $0.magnitude < $1.magnitude }

// Vector2(dx: 3.0, dy: 2.0)
let highestMagnitudeVec2 = vectors.maxElement { $0.magnitude < $1.magnitude }
```

3.0

```
let lowestMagnitudeVec2 = vectors.min { $0.magnitude < $1.magnitude }
let highestMagnitudeVec2 = vectors.max { $0.magnitude < $1.magnitude }
```

Безопасный доступ к индексам

Добавляя следующее расширение к индексам массива, можно получить доступ, не зная, находится ли индекс внутри границ.

```
extension Array {
    subscript (safe index: Int) -> Element? {
        return indices ~= index ? self[index] : nil
    }
}
```

пример:

```
if let thirdValue = array[safe: 2] {
    print(thirdValue)
}
```

Сравнение 2 массивов с zip

Функция zip принимает 2 параметра типа SequenceType и возвращает Zip2Sequence где каждый элемент содержит значение из первой последовательности и один из второй последовательности.

пример

```
let nums = [1, 2, 3]
let animals = ["Dog", "Cat", "Tiger"]
let numsAndAnimals = zip(nums, animals)
```

nomsAndAnimals теперь содержит следующие значения

sequence1	sequence1
1	"Dog"
2	"Cat"
3	"Tiger"

Это полезно, если вы хотите выполнить какое-то сравнение между n-м элементом каждого массива.

пример

Учитывая 2 массива Int(s)

```
let list0 = [0, 2, 4]
let list1 = [0, 4, 8]
```

вы хотите проверить, является ли каждое значение в list1 двойным из связанного значения в list0 .

```
let list1HasDoubleOfList0 = !zip(list0, list1).filter { $0 != (2 * $1)}.isEmpty
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/swift/topic/284/массивы

глава 28: Менеджер пакетов Swift

Examples

Создание и использование простого пакета Swift

Чтобы создать пакет Swift, откройте терминал и создайте пустую папку:

```
mkdir AwesomeProject
cd AwesomeProject
```

И запустите репозиторий Git:

```
git init
```

Затем создайте сам пакет. Можно создать структуру пакета вручную, но есть простой способ использования команды CLI.

Если вы хотите сделать исполняемый файл:

```
swift package init --type executable
```

Будет создано несколько файлов. Среди них *main.swift* станет точкой входа для вашего приложения.

Если вы хотите создать библиотеку:

```
swift package init --type library
```

Сгенерированный файл *AwesomeProject.swift* будет использоваться в качестве основного файла для этой библиотеки.

В обоих случаях вы можете добавить другие файлы Swift в папку « *Источники* » (применяются обычные правила для контроля доступа).

Сам файл *Package.swift* будет автоматически заполнен этим контентом:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

Версии пакета выполняются с помощью тегов Git:

```
git tag '1.0.0'
```

После нажатия на удаленный или локальный репозиторий Git ваш пакет будет доступен для других проектов.

Теперь ваш пакет готов к компиляции:

```
swift build
```

Скомпилированный проект будет доступен в *папке .build / debug* .

Ваш собственный пакет также может разрешать зависимости для других пакетов. Например, если вы хотите включить «SomeOtherPackage» в свой собственный проект, измените файл *Package.swift*, чтобы включить зависимость:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

Затем снова создайте свой проект: диспетчер пакетов Swift автоматически разрешит, загрузит и создаст зависимости.

Прочитайте Менеджер пакетов Swift онлайн: https://riptutorial.com/ru/swift/topic/5144/менеджер-пакетов-swift

глава 29: Метод Swizzling

замечания

При использовании метода swizzling в Swift существуют два требования, которые должны соответствовать вашим классам / методам:

- Ваш класс должен расширять NSObject
- Функции, которые вы хотите выполнить, должны иметь dynamic атрибут

Чтобы получить полное объяснение, почему это необходимо, ознакомьтесь с использованием Swift с Cocoa и Objective-C :

> **Требуется динамическая отправка**
>
> Хотя атрибут @objc предоставляет ваш Swift API во время выполнения Objective-C, он не гарантирует динамическую отправку свойства, метода, индекса или инициализатора. *Компилятор Swift может по-прежнему девиртуализировать или встроить членский доступ для оптимизации производительности вашего кода, минуя время выполнения Objective-C* . Когда вы отмечаете объявление участника dynamic модификатором, доступ к этому члену всегда динамически отправляется. Поскольку объявления, помеченные dynamic модификатором, отправляются с использованием среды выполнения Objective-C, они неявно помечены атрибутом @objc .
>
> Требование динамической отправки редко требуется. **Однако вы должны использовать dynamic модификатор, когда знаете, что реализация API заменяется во время выполнения** . Например, вы можете использовать функцию method_exchangeImplementations в среде выполнения Objective-C для замены реализации метода во время работы приложения. Если компилятор Swift встраивает реализацию метода или девиртуализированный доступ к нему, *новая реализация не будет использоваться* .

связи

Ссылка на объект Objective-C Runtime

Метод Swizzling на NSHipster

Examples

Расширение UIViewController и Swizzling viewDidLoad

В Objective-C метод swizzling - это процесс изменения реализации существующего селектора. Это возможно из-за того, как селектора отображаются в таблице рассылки или в таблице указателей на функции или методы.

Методы Pure Swift не динамически отправляются во время выполнения Objective-C, но мы все же можем использовать эти трюки в любом классе, который наследуется от NSObject .

Здесь мы расширим UIViewController и swizzle viewDidLoad чтобы добавить некоторые пользовательские протоколирования:

```
extension UIViewController {

    // We cannot override load like we could in Objective-C, so override initialize instead
    public override static func initialize() {

        // Make a static struct for our dispatch token so only one exists in memory
        struct Static {
            static var token: dispatch_once_t = 0
        }
```

```
        // Wrap this in a dispatch_once block so it is only run once
        dispatch_once(&Static.token) {
            // Get the original selectors and method implementations, and swap them with our
new method
            let originalSelector = #selector(UIViewController.viewDidLoad)
            let swizzledSelector = #selector(UIViewController.myViewDidLoad)

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            // class_addMethod can fail if used incorrectly or with invalid pointers, so check
to make sure we were able to add the method to the lookup table successfully
            if didAddMethod {
                class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
                method_exchangeImplementations(originalMethod, swizzledMethod);
            }
        }
    }

    // Our new viewDidLoad function
    // In this example, we are just logging the name of the function, but this can be used to
run any custom code
    func myViewDidLoad() {
        // This is not recursive since we swapped the Selectors in initialize().
        // We cannot call super in an extension.
        self.myViewDidLoad()
        print(#function) // logs myViewDidLoad()
    }
}
```

Основы Swift Swizzling

Давайте methodOne() реализацию методов methodOne() и methodTwo() в нашем классе TestSwizzling :

```
class TestSwizzling : NSObject {
    dynamic func methodOne()->Int{
        return 1
    }
}

extension TestSwizzling {

    //In Objective-C you'd perform the swizzling in load(),
    //but this method is not permitted in Swift
    override class func initialize()
    {

        struct Inner {
            static let i: () = {

                let originalSelector = #selector(TestSwizzling.methodOne)
                let swizzledSelector = #selector(TestSwizzling.methodTwo)
                let originalMethod = class_getInstanceMethod(TestSwizzling.self,
originalSelector);
```

```
                let swizzledMethod = class_getInstanceMethod(TestSwizzling.self,
swizzledSelector)
                method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
        let _ = Inner.i
    }

    func methodTwo()->Int{
        // It will not be a recursive call anymore after the swizzling
        return methodTwo()+1
    }
}

var c = TestSwizzling()
print(c.methodOne())
print(c.methodTwo())
```

Основы Swizzling - Objective-C

Пример Objective-C для swizzling initWithFrame: UIView initWithFrame: метод

```
static IMP original_initWithFrame;

+ (void)swizzleMethods {
    static BOOL swizzled = NO;
    if (!swizzled) {
        swizzled = YES;

        Method initWithFrameMethod =
            class_getInstanceMethod([UIView class], @selector(initWithFrame:));
        original_initWithFrame = method_setImplementation(
            initWithFrameMethod, (IMP)replacement_initWithFrame);
    }
}

static id replacement_initWithFrame(id self, SEL _cmd, CGRect rect) {

    // This will be called instead of the original initWithFrame method on UIView
    // Do here whatever you need...

    // Bonus: This is how you would call the original initWithFrame method
    UIView *view =
        ((id (*)(id, SEL, CGRect))original_initWithFrame)(self, _cmd, rect);

    return view;
}
```

Прочитайте Метод Swizzling онлайн: https://riptutorial.com/ru/swift/topic/1436/метод-swizzling

глава 30: наборы

Examples

Объявление наборов

Наборы представляют собой неупорядоченные коллекции уникальных значений. Уникальные значения должны быть одного типа.

```
var colors = Set<String>()
```

Вы можете объявить набор со значениями, используя синтаксис литерала массива.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Изменение значений в наборе

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Вы можете использовать метод insert(_:) для добавления нового элемента в набор.

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

Вы можете использовать метод remove(_:) для удаления элемента из набора. Он возвращает необязательное содержащее значение, которое было удалено, или значение nil, если значение не было в наборе.

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

Проверка того, содержит ли набор значение

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

Вы можете использовать метод contains(_:) чтобы проверить, содержит ли набор значение. Он вернет true, если набор содержит это значение.

```
if favoriteColors.contains("Blue") {
    print("Who doesn't like blue!")
}
// Prints "Who doesn't like blue!"
```

Выполнение операций над наборами

Общие значения из обоих наборов:

Вы можете использовать метод intersect(_:) для создания нового набора, содержащего все значения, общие для обоих наборов.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

Все значения из каждого набора:

Вы можете использовать метод union(_:) для создания нового набора, содержащего все уникальные значения из каждого набора.

```
let union = favoriteColors.union(newColors) // a OR b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

Обратите внимание, как значение «Зеленый» появляется только один раз в новом наборе.

Значения, которые не существуют в обоих наборах:

Вы можете использовать метод exclusiveOr(_:) для создания нового набора, содержащего уникальные значения, но не оба набора.

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

Обратите внимание, как значение «Зеленый» не отображается в новом наборе, так как оно находилось в обоих наборах.

Значения, которые не входят в набор:

Вы можете использовать метод subtract(_:) для создания нового набора, содержащего значения, которые не находятся в определенном наборе.

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

Обратите внимание, как значение «Зеленый» не отображается в новом наборе, так как оно также находилось во втором наборе.

Добавление значений моего собственного типа в Set

Чтобы определить Set вашего собственного типа, вам необходимо соответствовать типу Hashable

```
struct Starship: Hashable {
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:Starship, right: Starship) -> Bool {
    return left.name == right.name
}
```

Теперь вы можете создать Set Starship(s)

```
let ships : Set<Starship> = [Starship(name:"Enterprise D"), Starship(name:"Voyager"),
Starship(name:"Defiant") ]
```

CountedSet

3.0

Swift 3 представляет класс CountedSet (это Swift-версия класса NSCountedSet Objective-C).

CountedSet, как указано в названии, отслеживает, сколько раз присутствует значение.

```
let countedSet = CountedSet()
countedSet.add(1)
countedSet.add(1)
countedSet.add(1)
countedSet.add(2)

countedSet.count(for: 1) // 3
countedSet.count(for: 2) // 1
```

Прочитайте наборы онлайн: https://riptutorial.com/ru/swift/topic/371/наборы

глава 31: Начало работы с программно-ориентированным программированием

замечания

Дополнительную информацию по этой теме см. В разделе « Протокольное программирование WWDC 2015» в Swift .

Существует также большое письменное руководство по тому же: Введение в протокол-ориентированное программирование в Swift 2 .

Examples

Использование программно-ориентированного программирования для модульного тестирования

Программно-ориентированное программирование - полезный инструмент, позволяющий легко писать лучшие модульные тесты для нашего кода.

Предположим, мы хотим протестировать UIViewController, который полагается на класс ViewModel.

Необходимые шаги для производственного кода:

1. Определите протокол, который предоставляет открытый интерфейс класса ViewModel со всеми свойствами и методами, необходимыми для UIViewController.
2. Реализовать реальный класс ViewModel, соответствующий этому протоколу.
3. Используйте метод инъекции зависимостей, чтобы позволить контроллеру представления использовать нужную нам реализацию, передавая его как протокол, а не конкретный экземпляр.

```
protocol ViewModelType {
   var title : String {get}
   func confirm()
}

class ViewModel : ViewModelType {
   let title : String

   init(title: String) {
       self.title = title
   }
   func confirm() { ... }
}

class ViewController : UIViewController {
   // We declare the viewModel property as an object conforming to the protocol
   // so we can swap the implementations without any friction.
   var viewModel : ViewModelType!
   @IBOutlet var titleLabel : UILabel!

   override func viewDidLoad() {
       super.viewDidLoad()
       titleLabel.text = viewModel.title
   }

   @IBAction func didTapOnButton(sender: UIButton) {
       viewModel.confirm()
   }
}

// With DI we setup the view controller and assign the view model.
// The view controller doesn't know the concrete class of the view model,
// but just relies on the declared interface on the protocol.
```

```
let viewController = //... Instantiate view controller
viewController.viewModel = ViewModel(title: "MyTitle")
```

Затем, при модульном тесте:

1. Внедрить макет ViewModel, который соответствует одному протоколу
2. Передайте его тестируемому UIViewController с использованием инъекции зависимостей вместо реального экземпляра.
3. Тестовое задание!

```
class FakeViewModel : ViewModelType {
   let title : String = "FakeTitle"

   var didConfirm = false
   func confirm() {
       didConfirm = true
   }
}

class ViewControllerTest : XCTestCase {
    var sut : ViewController!
    var viewModel : FakeViewModel!

    override func setUp() {
        super.setUp()

        viewModel = FakeViewModel()
        sut = // ... initialization for view controller
        sut.viewModel = viewModel

        XCTAssertNotNil(self.sut.view) // Needed to trigger view loading
    }

    func testTitleLabel() {
        XCTAssertEqual(self.sut.titleLabel.text, "FakeTitle")
    }

    func testTapOnButton() {
        sut.didTapOnButton(UIButton())
        XCTAssertTrue(self.viewModel.didConfirm)
    }
}
```

Использование протоколов в качестве типов первого класса

Программно ориентированное по протоколу программное обеспечение может использоваться в качестве основного шаблона проектирования Swift.

Различные типы могут соответствовать одному протоколу, типы значений могут даже соответствовать нескольким протоколам и даже обеспечивать реализацию метода по умолчанию.

Первоначально определены протоколы, которые могут представлять собой обычно используемые свойства и / или методы с конкретными или генерическими типами.

```
protocol ItemData {

    var title: String { get }
    var description: String { get }
    var thumbnailURL: NSURL { get }
    var created: NSDate { get }
```

```
    var updated: NSDate { get }

}

protocol DisplayItem {

    func hasBeenUpdated() -> Bool
    func getFormattedTitle() -> String
    func getFormattedDescription() -> String

}

protocol GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void)
}
```

Может быть создана реализация по умолчанию для метода get, но при желании соответствующие типы могут переопределить реализацию.

```
extension GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void) {

        let date = NSDate(
        timeIntervalSinceNow: NSDate().timeIntervalSince1970
            + 5000)

        // get data from url
        let urlData: [String: AnyObject] = [
            "title": "Red Camaro",
            "desc": "A fast red car.",
            "thumb":"http://cars.images.com/red-camaro.png",
            "created": NSDate(), "updated": date]

        // in this example forced unwrapping is used
        // forced unwrapping should never be used in practice
        // instead conditional unwrapping should be used (guard or if/let)
        let item = Item(
            title: urlData["title"] as! String,
            description: urlData["desc"] as! String,
            thumbnailURL: NSURL(string: urlData["thumb"] as! String)!,
            created: urlData["created"] as! NSDate,
            updated: urlData["updated"] as! NSDate)

        completed([item])

    }
}

struct ItemOperation: GetAPIItemDataOperation { }
```

Тип значения, который соответствует протоколу ItemData, этот тип значения также может соответствовать другим протоколам.

```
struct Item: ItemData {

    let title: String
    let description: String
    let thumbnailURL: NSURL
```

```
    let created: NSDate
    let updated: NSDate

}
```

Здесь элемент struct расширяется, чтобы соответствовать элементу отображения.

```
extension Item: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updated.timeIntervalSince1970 >
            created.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return description.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}
```

Пример сайта вызова для использования метода статического get.

```
ItemOperation.get(NSURL()) { (itemData) in

    // perhaps inform a view of new data
    // or parse the data for user requested info, etc.
    dispatch_async(dispatch_get_main_queue(), {

        // self.items = itemData
    })

}
```

Различные варианты использования потребуют разных реализаций. Основная идея здесь заключается в том, чтобы показать соответствие от разных типов, где протокол является основной точкой фокусировки в дизайне. В этом примере, возможно, данные API условно сохраняются в объекте Core Data.

```
// the default core data created classes + extension
class LocalItem: NSManagedObject { }

extension LocalItem {

    @NSManaged var title: String
    @NSManaged var itemDescription: String
    @NSManaged var thumbnailURLStr: String
    @NSManaged var createdAt: NSDate
    @NSManaged var updatedAt: NSDate
}
```

Здесь класс поддержки Core Data также может соответствовать протоколу DisplayItem.

```
extension LocalItem: DisplayItem {

```

```
    func hasBeenUpdated() -> Bool {
        return updatedAt.timeIntervalSince1970 >
            createdAt.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return itemDescription.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}

// In use, the core data results can be
// conditionally casts as a protocol
class MyController: UIViewController {

    override func viewDidLoad() {

        let fr: NSFetchRequest = NSFetchRequest(
        entityName: "Items")

        let context = NSManagedObjectContext(
        concurrencyType: .MainQueueConcurrencyType)

        do {

            let items: AnyObject = try context.executeFetchRequest(fr)
            if let displayItems = items as? [DisplayItem] {

                print(displayItems)
            }

        } catch let error as NSError {
            print(error.localizedDescription)
        }

    }
}
```

Прочитайте Начало работы с программно-ориентированным программированием онлайн: https://riptutorial.com/ru/swift/topic/2502/начало-работы-с-программно-ориентированным-программированием

глава 32: Обработка ошибок

замечания

Дополнительные сведения об ошибках см. В разделе Быстрый язык программирования .

Examples

Основы обработки ошибок

Функции в Swift могут возвращать значения, **бросать ошибки** или и то, и другое:

```
func reticulateSplines()                // no return value and no error
func reticulateSplines() -> Int         // always returns a value
func reticulateSplines() throws         // no return value, but may throw an error
func reticulateSplines() throws -> Int  // may either return a value or throw an error
```

Любое значение, которое соответствует протоколу ErrorType (включая объекты NSError), может быть выбрано как ошибка. Перечисления обеспечивают удобный способ определения пользовательских ошибок:

2,0 2,2

```
enum NetworkError: ErrorType {
    case Offline
    case ServerError(String)
}
```

3.0

```
enum NetworkError: Error {
    // Swift 3 dictates that enum cases should be `lowerCamelCase`
    case offline
    case serverError(String)
}
```

Ошибка указывает на нефатальный сбой во время выполнения программы и обрабатывается специализированными конструкциями потока управления do / catch , throw и try .

```
func fetchResource(resource: NSURL) throws -> String {
    if let (statusCode, responseString) = /* ...from elsewhere...*/ {
        if case 500..<600 = statusCode {
            throw NetworkError.serverError(responseString)
        } else {
            return responseString
        }
    } else {
        throw NetworkError.offline
    }
}
```

Ошибки могут быть обнаружены с помощью do / catch :

```
do {
    let response = try fetchResource(resURL)
    // If fetchResource() didn't throw an error, execution continues here:
    print("Got response: \(response)")
```

```
    ...
} catch {
    // If an error is thrown, we can handle it here.
    print("Whoops, couldn't fetch resource: \(error)")
}
```

Любая функция, которая может вызвать ошибку, ***должна*** вызываться с помощью try , try? , или try! :

```
// error: call can throw but is not marked with 'try'
let response = fetchResource(resURL)

// "try" works within do/catch, or within another throwing function:
do {
    let response = try fetchResource(resURL)
} catch {
    // Handle the error
}

func foo() throws {
    // If an error is thrown, continue passing it up to the caller.
    let response = try fetchResource(resURL)
}

// "try?" wraps the function's return value in an Optional (nil if an error was thrown).
if let response = try? fetchResource(resURL) {
    // no error was thrown
}

// "try!" crashes the program at runtime if an error occurs.
let response = try! fetchResource(resURL)
```

Поиск различных типов ошибок

Давайте создадим наш собственный тип ошибки для этого примера.

2,2

```
enum CustomError: ErrorType {
    case SomeError
    case AnotherError
}

func throwing() throws {
    throw CustomError.SomeError
}
```

3.0

```
enum CustomError: Error {
    case someError
    case anotherError
}

func throwing() throws {
    throw CustomError.someError
}
```

Синтаксис Do-Catch позволяет поймать заброшенную ошибку и *автоматически* создает постоянную именованную error доступную в блоке catch :

```
do {
    try throwing()
} catch {
    print(error)
}
```

Вы также можете объявить переменную самостоятельно:

```
do {
    try throwing()
} catch let oops {
    print(oops)
}
```

Это также возможно цепь различных catch отчетности. Это удобно, если в блоке Do можно выбросить несколько типов ошибок.

Здесь Do-Catch сначала попытается передать ошибку как CustomError , а затем как NSError если пользовательский тип не был сопоставлен.

2,2

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

3.0

В Swift 3 нет необходимости явно отключать NSError.

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

Схема улова и переключения для явной обработки ошибок

```
class Plane {

    enum Emergency: ErrorType {
        case NoFuel
        case EngineFailure(reason: String)
        case DamagedWing
    }

    var fuelInKilograms: Int

    //... init and other methods not shown

    func fly() throws {
        // ...
        if fuelInKilograms <= 0 {
```

```
            // uh oh...
            throw Emergency.NoFuel
        }
    }

}
```

В классе клиента:

```
let airforceOne = Plane()
do {
    try airforceOne.fly()
} catch let emergency as Plane.Emergency {
    switch emergency {
    case .NoFuel:
        // call nearest airport for emergency landing
    case .EngineFailure(let reason):
        print(reason) // let the mechanic know the reason
    case .DamagedWing:
        // Assess the damage and determine if the president can make it
    }
}
```

Отключение распространения ошибок

Создатели Swift уделяют много внимания тому, чтобы сделать язык выразительным, а обработка ошибок - это именно то, что выразительно. Если вы попытаетесь вызвать функцию, которая может выдать ошибку, вызов функции должен предшествовать ключевому слову try. Ключевое слово try не является волшебным. Все, что он делает, делает разработчик осведомленным о способности бросать функцию.

Например, следующий код использует функцию loadImage (atPath :), которая загружает ресурс изображения по заданному пути или выдает ошибку, если изображение не может быть загружено. В этом случае, поскольку изображение поставляется вместе с приложением, во время выполнения не будет выдаваться ошибка, поэтому целесообразно отключить распространение ошибок.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Создание пользовательской ошибки с локализованным описанием

Создать enum пользовательских ошибок

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

Создайте extension RegistrationError для обработки описания Localized.

```
extension RegistrationError: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .invalidEmail:
            return NSLocalizedString("Description of invalid email address", comment: "Invalid
Email")
        case .invalidPassword:
            return NSLocalizedString("Description of invalid password", comment: "Invalid
```

```
Password")
        case .invalidPhoneNumber:
            return NSLocalizedString("Description of invalid phoneNumber", comment: "Invalid
Phone Number")
        }
    }
}
```

Ошибка обработки:

```
let error: Error = RegistrationError.invalidEmail
print(error.localizedDescription)
```

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/swift/topic/283/обработка-ошибок

глава 33: Обработчик завершения

Вступление

Практически все приложения используют асинхронные функции, чтобы код не блокировал основной поток.

Examples

Обработчик завершения без входных аргументов

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//Call the function
sampleWithCompletion {
    print("after one second")
}
```

Обработчик завершения с входным аргументом

```
enum ReadResult{
    case Successful
    case Failed
    case Pending
}

struct OutpuData {
    var data = Data()
    var result: ReadResult
    var error: Error?
}

func readData(from url: String, completion: @escaping (OutpuData) ->  Void) {
    var _data = OutpuData(data: Data(), result: .Pending, error: nil)
    DispatchQueue.global().async {
        let url=URL(string: url)
        do {
            let rawData = try Data(contentsOf: url!)
            _data.result = .Successful
            _data.data = rawData
            completion(_data)
        }
        catch let error {
            _data.result = .Failed
            _data.error = error
            completion(_data)
        }

    }
}
```

```
readData(from: "https://raw.githubusercontent.com/trev/bearcal/master/sample-data-large.json")
{ (output) in
    switch output.result {
    case .Successful:
        break
    case .Failed:
        break
    case .Pending:
        break

    }
}
```

Прочитайте Обработчик завершения онлайн: https://riptutorial.com/ru/swift/topic/9378/обработчик-завершения

глава 34: отражение

Синтаксис

- Зеркало (отражающее: экземпляр) // Инициализирует зеркало с объектом для отражения
- mirror.displayStyle // Стиль отображения, используемый для игровых площадок Xcode
- mirror.description // Текстовое представление этого экземпляра, см. CustomStringConvertible
- mirror.subjectType // Возвращает тип отраженного объекта
- mirror.superclassMirror // Возвращает зеркало суперкласса объекта, который отражается

замечания

1. ***Основные пометки:***

Mirror - это struct используемая при интроспекции объекта в Свифте. Его наиболее заметным свойством является массив детей. Одним из возможных вариантов использования является сериализация структуры для Core Data . Это делается путем преобразования struct в объект NSManagedObject .

2. ***Основное использование для зеркальных заметок:***

children свойство Mirror представляет собой массив дочерних объектов из объекта экземпляр Зеркала отражает. У child объекта есть две label и value свойств. Например, ребенок может быть свойством с названием title и значением Game of Thrones: A Song of Ice and Fire .

Examples

Основное использование для зеркала

Создание класса для объекта Mirror

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

Создание экземпляра, который на самом деле будет предметом зеркала. Также здесь вы можете добавить значения к свойствам класса Project.

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "test app for Reflection"
```

В приведенном ниже коде показано создание экземпляра Mirror. Свойство AnyForwardCollection<Child> зеркала - AnyForwardCollection<Child> где Child является кортежем typealias для свойства и значения объекта. Child была label: String и value: Any .

```
let projectMirror = Mirror(reflecting: sampleProject)
let properties = projectMirror.children

print(properties.count)        //5
```

```
print(properties.first?.label) //Optional("title")
print(properties.first!.value) //MirrorMirror
print()

for property in properties {
    print("\(property.label!):\(property.value)")
}
```

Вывод на игровой площадке или консоли в Xcode для цикла for выше.

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

Протестировано на игровой площадке на Xcode 8 beta 2

Получение типа и имен свойств для класса без необходимости его экземпляра

Использование класса Swift Mirror работает, если вы хотите извлечь *имя* , *значение* и *тип* (Swift 3: type(of: value) , Swift 2: value.dynamicType) свойств для **экземпляра** определенного класса.

Если вы наследуете класс из NSObject , вы можете использовать метод class_copyPropertyList вместе с property_getAttributes чтобы узнать *имя* и *типы* свойств для класса – **без его экземпляра** . Для этого я создал проект для Github , но вот код:

```
func getTypesOfProperties(in clazz: NSObject.Type) -> Dictionary<String, Any>? {
    var count = UInt32()
    guard let properties = class_copyPropertyList(clazz, &count) else { return nil }
    var types: Dictionary<String, Any> = [:]
    for i in 0..<Int(count) {
        guard let property: objc_property_t = properties[i], let name = getNameOf(property:
property) else { continue }
        let type = getTypeOf(property: property)
        types[name] = type
    }
    free(properties)
    return types
}

func getTypeOf(property: objc_property_t) -> Any {
    guard let attributesAsNSString: NSString = NSString(utf8String:
property_getAttributes(property)) else { return Any.self }
    let attributes = attributesAsNSString as String
    let slices = attributes.components(separatedBy: "\"")
    guard slices.count > 1 else { return getPrimitiveDataType(withAttributes: attributes) }
    let objectClassName = slices[1]
    let objectClass = NSClassFromString(objectClassName) as! NSObject.Type
    return objectClass
}

   func getPrimitiveDataType(withAttributes attributes: String) -> Any {
        guard let letter = attributes.substring(from: 1, to: 2), let type =
primitiveDataTypes[letter] else { return Any.self }
        return type
    }
```

Где primitiveDataTypes – это словарь, сопоставляющий букву в строке атрибута с типом значения:

```
let primitiveDataTypes: Dictionary<String, Any> = [
    "c" : Int8.self,
    "s" : Int16.self,
    "i" : Int32.self,
    "q" : Int.self, //also: Int64, NSInteger, only true on 64 bit platforms
    "S" : UInt16.self,
    "I" : UInt32.self,
    "Q" : UInt.self, //also UInt64, only true on 64 bit platforms
    "B" : Bool.self,
    "d" : Double.self,
    "f" : Float.self,
    "{" : Decimal.self
]

   func getNameOf(property: objc_property_t) -> String? {
        guard let name: NSString = NSString(utf8String: property_getName(property)) else {
return nil }
        return name as String
    }
```

Он может извлечь NSObject.Type всех свойств, которые наследует тип класса из NSObject таких как NSDate (Swift3: Date), NSString (Swift3: String ?) И NSNumber , однако он является хранилищем в типе Any (как вы можете видеть как тип значения словаря, возвращаемого методом). Это связано с ограничениями value types такими как Int, Int32, Bool. Поскольку эти типы не наследуют от NSObject, вызывая .self на например, Int - Int.self не возвращает NSObject.Type, а тип Any . Таким образом, метод возвращает Dictionary<String, Any>? а не Dictionary<String, NSObject.Type>? ,

Вы можете использовать этот метод следующим образом:

```
class Book: NSObject {
    let title: String
    let author: String?
    let numberOfPages: Int
    let released: Date
    let isPocket: Bool

    init(title: String, author: String?, numberOfPages: Int, released: Date, isPocket: Bool) {
        self.title = title
        self.author = author
        self.numberOfPages = numberOfPages
        self.released = released
        self.isPocket = isPocket
    }
}

guard let types = getTypesOfProperties(in: Book.self) else { return }
for (name, type) in types {
    print("'\(name)' has type '\(type)'")
}
// Prints:
// 'title' has type 'NSString'
// 'numberOfPages' has type 'Int'
// 'author' has type 'NSString'
// 'released' has type 'NSDate'
// 'isPocket' has type 'Bool'
```

Вы также можете попробовать забрасывать Any в NSObject.Type , который сменит для всех свойств , наследующих от NSObject , то вы можете проверить тип , используя стандарт == оператор:

```
func checkPropertiesOfBook() {
```

```
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if let objectType = type as? NSObject.Type {
            if objectType == NSDate.self {
                print("Property named '\(name)' has type 'NSDate'")
            } else if objectType == NSString.self {
                print("Property named '\(name)' has type 'NSString'")
            }
        }
    }
}
```

Если вы объявите этот пользовательский == operator:

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

Затем вы можете проверить тип value types следующим образом:

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if type == Int.self {
            print("Property named '\(name)' has type 'Int'")
        } else if type == Bool.self {
            print("Property named '\(name)' has type 'Bool'")
        }
    }
}
```

ОГРАНИЧЕНИЯ Это решение не работает, если value types являются опциями. Если вы объявили свойство в вашем подклассе var myOptionalInt: Int? следующим образом: var myOptionalInt: Int? , код выше не найдет это свойство, потому что метод class_copyPropertyList не содержит необязательные типы значений.

Прочитайте отражение онлайн: https://riptutorial.com/ru/swift/topic/1201/отражение

глава 35: переключатель

параметры

параметр	подробности
Значение для тестирования	Переменная, которая сравнивается с

замечания

Поставьте случай для каждого возможного значения вашего ввода. Используйте default case по default case для покрытия оставшихся значений ввода, которые вы не хотите указывать. Случай по умолчанию должен быть последним.

По умолчанию Переключатели в Swift не будут продолжать проверять другие случаи после согласования случая.

Examples

Основное использование

```
let number = 3
switch number {
case 1:
    print("One!")
case 2:
    print("Two!")
case 3:
    print("Three!")
default:
    print("Not One, Two or Three")
}
```

Операторы switch также работают с типами данных, отличными от целых. Они работают с любым типом данных. Вот пример включения строки:

```
let string = "Dog"
switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

Это напечатает следующее:

```
Animal is a house pet.
```

Согласование нескольких значений

Один случай в инструкции switch может совпадать по нескольким значениям.

```
let number = 3
switch number {
```

```
case 1, 2:
    print("One or Two!")
case 3:
    print("Three!")
case 4, 5, 6:
    print("Four, Five or Six!")
default:
    print("Not One, Two, Three, Four, Five or Six")
}
```

Соответствие диапазону

Один случай в инструкции switch может соответствовать диапазону значений.

```
let number = 20
switch number {
case 0:
    print("Zero")
case 1..<10:
    print("Between One and Ten")
case 10..<20:
    print("Between Ten and Twenty")
case 20..<30:
    print("Between Twenty and Thirty")
default:
    print("Greater than Thirty or less than Zero")
}
```

Использование оператора where в коммутаторе

Оператор where может использоваться в совпадении с ключом, чтобы добавить дополнительные критерии, необходимые для положительного совпадения. Следующий пример проверяет не только диапазон, но также, если число нечетное или четное:

```
switch (temperature) {
      case 0...49 where temperature % 2 == 0:
        print("Cold and even")

      case 50...79 where temperature % 2 == 0:
        print("Warm and even")

      case 80...110 where temperature % 2 == 0:
        print("Hot and even")

      default:
        print("Temperature out of range or odd")
}
```

Удовлетворите одно из нескольких ограничений, используя переключатель

Вы можете создать кортеж и использовать такой переключатель:

```
var str: String? = "hi"
var x: Int? = 5

switch (str, x) {
case (.Some,.Some):
```

```
    print("Both have values")
case (.Some, nil):
    print("String has a value")
case (nil, .Some):
    print("Int has a value")
case (nil, nil):
    print("Neither have values")
}
```

Частичное совпадение

Оператор Switch использует частичное совпадение.

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0): // 1
  print("Origin")
case (_, 0, 0): // 2
  print("On the x-axis.")
case (0, _, 0): // 3
  print("On the y-axis.")
case (0, 0, _): // 4
  print("On the z-axis.")
default:        // 5
  print("Somewhere in space")
}
```

1. Соответствует точно случаю, когда значение (0,0,0). Это источник 3D-пространства.
2. Соответствует y = 0, z = 0 и любому значению x. Это означает, что координата находится на оси x.
3. Соответствует x = 0, z = 0 и любому значению y. Это означает, что координата находится на оси.
4. Соответствует x = 0, y = 0 и любому значению z. Это означает, что координата находится на оси z.
5. Соответствует остальной части координат.

Примечание: использование подчеркивания означает, что вы не заботитесь о значении.

Если вы не хотите игнорировать значение, вы можете использовать его в своем операторе switch, например:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0):
  print("Origin")
case (let x, 0, 0):
  print("On the x-axis at x = \(x)")
case (0, let y, 0):
  print("On the y-axis at y = \(y)")
case (0, 0, let z):
  print("On the z-axis at z = \(z)")
case (let x, let y, let z):
  print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Здесь в примерах оси используется синтаксис let, чтобы вытащить соответствующие значения. Затем код печатает значения с использованием строковой интерполяции для построения строки.

Примечание: в этом операторе switch вам не требуется значение по умолчанию. Это связано с тем, что окончательный случай по существу является значением по умолчанию - он соответствует чему-либо, потому что ограничений на какую-либо часть кортежа нет. Если оператор switch исчерпывает все возможные значения со своими случаями, то по умолчанию не требуется.

Мы также можем использовать синтаксис let-where для соответствия более сложным случаям. Например:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (let x, let y, _) where y == x:
  print("Along the y = x line.")
case (let x, let y, _) where y == x * x:
  print("Along the y = x^2 line.")
default:
break
}
```

Здесь мы сопоставляем «y равно x» и «y равно x квадрату».

Переключительные провалы

Стоит отметить, что в быстрых, в отличие от других языков, с которыми знакомы люди, в конце каждого случая есть неявный перерыв. Чтобы перейти к следующему случаю (т. fallthrough несколько случаев), вам нужно использовать инструкцию fallthrough .

```
switch(value) {
case 'one':
    // do operation one
    fallthrough
case 'two':
    // do this either independant, or in conjunction with first case
default:
    // default operation
}
```

это полезно для таких вещей, как потоки.

Переключатель и перечисление

Оператор Switch очень хорошо работает с значениями Enum

```
enum CarModel {
    case Standard, Fast, VeryFast
}

let car = CarModel.Standard

switch car {
case .Standard: print("Standard")
case .Fast: print("Fast")
case .VeryFast: print("VeryFast")
}
```

> Поскольку мы предоставили случай для каждого возможного значения автомобиля, мы опускаем случай по default .

Переключатель и опции

Некоторые примеры случаев, когда результат является необязательным.

```
var result: AnyObject? = someMethod()

switch result {
case nil:
    print("result is nothing")
case is String:
    print("result is a String")
case _ as Double:
    print("result is not nil, any value that is a Double")
case let myInt as Int where myInt > 0:
    print("\(myInt) value is not nil but an int and greater than 0")
case let a?:
    print("\(a) - value is unwrapped")
}
```

Переключатели и кортежи

Коммутаторы могут включать кортежи:

```
public typealias mdyTuple = (month: Int, day: Int, year: Int)

let fredsBirthday =   (month: 4, day: 3, year: 1973)




switch theMDY
{
//You can match on a literal tuple:
case (fredsBirthday):
  message = "\(date) \(prefix) the day Fred was born"

//You can match on some of the terms, and ignore others:
case (3, 15, _):
  message = "Beware the Ides of March"

//You can match on parts of a literal tuple, and copy other elements
//into a constant that you use in the body of the case:
case (bobsBirthday.month, bobsBirthday.day, let year) where year > bobsBirthday.year:
  message = "\(date) \(prefix) Bob's \(possessiveNumber(year - bobsBirthday.year))" +
    "birthday"

//You can copy one or more elements of the tuple into a constant and then
//add a where clause that further qualifies the case:
case (susansBirthday.month, susansBirthday.day, let year)
  where year > susansBirthday.year:
  message = "\(date) \(prefix) Susan's " +
    "\(possessiveNumber(year - susansBirthday.year)) birthday"

//You can match some elements to ranges:.
case (5, 1...15, let year):
  message = "\(date) \(prefix) in the first half of May, \(year)"
}
```

Соответствие по классу - отлично подходит для подготовки к обучению

Вы также можете сделать переключатель оператора switch на основе **класса** вещи, которую вы

включаете.

Пример, где это полезно, - в prepareForSegue . Я использовал для переключения на основе идентификатора segue, но это хрупкое. если позже вы измените раскадровку и переименуете идентификатор segue, он сломает ваш код. Или, если вы используете segues для нескольких экземпляров того же класса контроллера вида (но разные сценарии раскадровки), вы не можете использовать идентификатор segue для определения класса адресата.

Операции Swift для спасения.

Используйте Swift case let var as Class синтаксис case let var as Class , например:

3.0

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

3.0

В Swift 3 ситакс слегка изменился:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

Прочитайте переключатель онлайн: https://riptutorial.com/ru/swift/topic/207/переключатель

глава 36: Переменные и свойства

замечания

Свойства : Связан с типом

Переменные : не связаны с типом

Дополнительную информацию см. На языке быстрого программирования iBook .

Examples

Создание переменной

Объявите новую переменную с var , за которой следует имя, тип и значение:

```
var num: Int = 10
```

Переменные могут иметь свои значения:

```
num = 20 // num now equals 20
```

Если они не определены с let :

```
let num: Int = 10 // num cannot change
```

Swift указывает тип переменной, поэтому вам не всегда нужно объявлять тип переменной:

```
let ten = 10 // num is an Int
let pi = 3.14 // pi is a Double
let floatPi: Float = 3.14 // floatPi is a Float
```

Имена переменных не ограничены буквами и цифрами - они могут также содержать большинство других символов в Юникоде, хотя существуют некоторые ограничения

> Имена констант и переменных не могут содержать пробельные символы, математические символы, стрелки, частные (или недействительные) кодовые точки Юникода или символы рисунка строки и окна. Они также не могут начинаться с числа
>
> **Источник** developer.apple.com

```
var π: Double = 3.14159
var □□: String = "Apples"
```

Основы собственности

Свойства могут быть добавлены к классу или структуре (технически перечислены также, см. Пример «Вычислимые свойства»). Они добавляют значения, которые ассоциируются с экземплярами классов / структур:

```
class Dog {
    var name = ""
}
```

В приведенном выше случае экземпляры Dog имеют свойство с именем name type String . Свойство может быть доступно и изменено на экземплярах Dog :

```
let myDog = Dog()
myDog.name = "Doggy" // myDog's name is now "Doggy"
```

Эти типы свойств считаются **хранимыми свойствами** , поскольку они хранят что-то на объекте и влияют на его память.

Леновые сохраненные свойства

Lazy хранимые свойства имеют значения, которые не вычисляются до первого доступа. Это полезно для экономии памяти, когда вычисление переменной вычисляется дорого. Вы объявляете ленивое имущество lazy :

```
lazy var veryExpensiveVariable = expensiveMethod()
```

Часто ему присваивается возвращаемое значение замыкания:

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

Ленивые хранимые свойства должны быть объявлены с помощью var .

Вычислимые свойства

В отличие от хранимых свойств, **вычисляемые свойства** создаются с помощью геттера и сеттера, выполняя необходимый код при доступе и настройке. Вычисленные свойства должны определять тип:

```
var pi = 3.14

class Circle {
    var radius = 0.0
    var circumference: Double {
        get {
            return pi * radius * 2
        }
        set {
            radius = newValue / pi / 2
        }
    }
}

let circle = Circle()
circle.radius = 1
print(circle.circumference) // Prints "6.28"
circle.circumference = 14
print(circle.radius) // Prints "2.229..."
```

Вычислимое свойство только для чтения все еще объявляется с помощью var :

```
var circumference: Double {
    get {
        return pi * radius * 2
    }
}
```

Выделенные свойства только для чтения можно сократить, чтобы исключить get :

```
var circumference: Double {
    return pi * radius * 2
}
```

Локальные и глобальные переменные

Локальные переменные определяются внутри функции, метода или закрытия:

```
func printSomething() {
    let localString = "I'm local!"
    print(localString)
}

func printSomethingAgain() {
    print(localString) // error
}
```

Глобальные переменные определяются вне функции, метода или закрытия и не определены внутри типа (думайте вне всех скобок). Они могут использоваться везде:

```
let globalString = "I'm global!"
print(globalString)

func useGlobalString() {
    print(globalString) // works!
}

for i in 0..<2 {
    print(globalString) // works!
}

class GlobalStringUser {
    var computeGlobalString {
        return globalString // works!
    }
}
```

Глобальные переменные определяются лениво (см. Пример «Lazy Properties»).

Свойства типа

Свойства типа - это свойства самого типа, а не экземпляра. Они могут быть как сохраненными, так и вычисленными свойствами. Вы объявляете свойство type со static :

```
struct Dog {
    static var noise = "Bark!"
}

print(Dog.noise) // Prints "Bark!"
```

В классе вы можете использовать ключевое слово class вместо static чтобы сделать его переопределяемым. Однако вы можете применить это только к вычисленным свойствам:

```
class Animal {
    class var noise: String {
        return "Animal noise!"
    }
}
```

```
class Pig: Animal {
    override class var noise: String {
        return "Oink oink!"
    }
}
```

Это часто используется с шаблоном singleton .

Наблюдатели за недвижимостью

Наблюдатели собственности реагируют на изменения стоимости объекта.

```
var myProperty = 5 {
    willSet {
        print("Will set to \(newValue). It was previously \(myProperty)")
    }
    didSet {
        print("Did set to \(myProperty). It was previously \(oldValue)")
    }
}
myProperty = 6
// prints: Will set to 6, It was previously 5
// prints: Did set to 6. It was previously 5
```

- willSet вызывается **до** myProperty будет установлен myProperty . Новое значение доступно как newValue , и старое значение по-прежнему доступно как myProperty .
- didSet вызывается **после установки** myProperty . Старое значение доступно как oldValue , и новое значение теперь доступно как myProperty .

  Примечание: didSet и willSet не будут вызываться в следующих случаях:

  - Присвоение начального значения
  - Изменение переменной в ее собственном didSet или willSet

- Имена параметров для oldValue и newValue of didSet и willSet также могут быть объявлены для повышения удобочитаемости:

```
var myFontSize = 10 {
    willSet(newFontSize) {
        print("Will set font to \(newFontSize), it was \(myFontSize)")
    }
    didSet(oldFontSize) {
        print("Did set font to \(myFontSize), it was \(oldFontSize)")
    }
}
```

Внимание. Хотя при объявлении имен параметров сеттера необходимо соблюдать осторожность, чтобы не смешивать имена:

- willSet(oldValue) и didSet(newValue) полностью легальны, но значительно запутают читателей вашего кода.

Прочитайте Переменные и свойства онлайн: https://riptutorial.com/ru/swift/topic/536/переменные-и-свойства

глава 37: Перечисления

замечания

Подобно структурам и в отличие от классов, перечисления являются типами значений и копируются вместо ссылки, когда они передаются.

Дополнительные сведения о перечислениях см. В разделе «Быстрый язык программирования» .

Examples

Основные перечисления

Перечисление предоставляет набор связанных значений:

```
enum Direction {
    case up
    case down
    case left
    case right
}

enum Direction { case up, down, left, right }
```

Значения Enum могут использоваться их полностью квалифицированным именем, но вы можете опустить имя типа, когда это можно сделать:

```
let dir = Direction.up
let dir: Direction = Direction.up
let dir: Direction = .up

// func move(dir: Direction)...
move(Direction.up)
move(.up)

obj.dir = Direction.up
obj.dir = .up
```

Самый фундаментальный способ сравнения / извлечения значений enum - с помощью оператора switch :

```
switch dir {
case .up:
    // handle the up case
case .down:
    // handle the down case
case .left:
    // handle the left case
case .right:
    // handle the right case
}
```

Простые перечисления автоматически Hashable , Equatable и имеют преобразования строк:

```
if dir == .down { ... }

let dirs: Set<Direction> = [.right, .left]

print(Direction.up)  // prints "up"
```

```
debugPrint(Direction.up)  // prints "Direction.up"
```

Перечисления со связанными значениями

События Enum могут содержать одну или несколько **полезных нагрузок** (**связанных значений**):

```
enum Action {
    case jump
    case kick
    case move(distance: Float)  // The "move" case has an associated distance
}
```

Полезная нагрузка должна быть предоставлена при создании экземпляра значения перечисления:

```
performAction(.jump)
performAction(.kick)
performAction(.move(distance: 3.3))
performAction(.move(distance: 0.5))
```

Оператор switch может извлечь связанное значение:

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let distance):  // or case let .move(distance):
    print("Moving: \(distance)")
}
```

Выделение одного случая может быть выполнено с использованием if case :

```
if case .move(let distance) = action {
    print("Moving: \(distance)")
}
```

Синтаксис guard case можно использовать для последующего использования:

```
guard case .move(let distance) = action else {
    print("Action is not move")
    return
}
```

Перечисления со связанными значениями по умолчанию не Equatable . Реализация оператора == должна выполняться вручную:

```
extension Action: Equatable { }

func ==(lhs: Action, rhs: Action) -> Bool {
    switch lhs {
    case .jump: if case .jump = rhs { return true }
    case .kick: if case .kick = rhs { return true }
    case .move(let lhsDistance): if case .move (let rhsDistance) = rhs { return lhsDistance ==
rhsDistance }
    }
    return false
}
```

Косвенная полезная нагрузка

Обычно перечисления не могут быть рекурсивными (потому что они потребуют бесконечного хранения):

```
enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)  // error: recursive enum 'Tree<T>' is not marked 'indirect'
}
```

indirect ключевое слово заставляет enum хранить свою полезную нагрузку со слоем косвенности, а не хранить его в строке. Вы можете использовать это ключевое слово в одном случае:

```
enum Tree<T> {
    case leaf(T)
    indirect case branch(Tree<T>, Tree<T>)
}

let tree = Tree.branch(.leaf(1), .branch(.leaf(2), .leaf(3)))
```

indirect также работает на всей переписке, делая при необходимости косвенным косвенным образом:

```
indirect enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)
}
```

Значения Raw и Hash

Перечисления без полезных нагрузок могут иметь *необработанные значения* любого литерального типа:

```
enum Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

Перечисления без определенного типа не выставляют свойство rawValue

```
enum Rotation {
    case up
    case right
    case down
    case left
}

let foo = Rotation.up
foo.rawValue //error
```

Предполагается, что значения целых сырых значений начинаются с 0 и монотонно возрастают:

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue is automatically 0
    case bar  // rawValue is automatically 1
    case baz = 7
    case quux  // rawValue is automatically 8
}
```

Строковые исходные значения могут быть синтезированы автоматически:

```
enum MarsMoon: String {
    case phobos  // rawValue is automatically "phobos"
    case deimos  // rawValue is automatically "deimos"
}
```

Необработанное перечисление автоматически соответствует RawRepresentable . Вы можете получить соответствующее исходное значение значения .rawValue с помощью .rawValue :

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    ...
}
```

Вы также можете создать перечисление *из* необработанного значения с помощью init?(rawValue:) :

```
let rotation = Rotation(rawValue: 0)  // returns Rotation.Up
let otherRotation = Rotation(rawValue: 45)  // returns nil (there is no Rotation with rawValue
45)

if let moon = MarsMoon(rawValue: str) {
    print("Mars has a moon named \(str)")
} else {
    print("Mars doesn't have a moon named \(str)")
}
```

Если вы хотите получить хеш-значение определенного перечисления, вы можете получить доступ к его hashValue, хеш-значение вернет индекс перечисления, начиная с нуля.

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue is 8
quux?.hashValue //hashValue is 3
```

Инициализаторы

В Enums могут быть настраиваемые методы init, которые могут быть более полезными, чем init?(rawValue:) по умолчанию init?(rawValue:) . Enums также может хранить значения. Это может быть полезно для хранения значений, которые они инициализировали и извлекали это значение позже.

```
enum CompassDirection {
    case north(Int)
    case south(Int)
    case east(Int)
    case west(Int)

    init?(degrees: Int) {
        switch degrees {
        case 0...45:
            self = .north(degrees)
        case 46...135:
            self = .east(degrees)
        case 136...225:
            self = .south(degrees)
        case 226...315:
            self = .west(degrees)
        case 316...360:
            self = .north(degrees)
        default:
            return nil
```

```
        }
    }

    var value: Int = {
        switch self {
            case north(let degrees):
                return degrees
            case south(let degrees):
                return degrees
            case east(let degrees):
                return degrees
            case west(let degrees):
                return degrees
        }
    }
}
```

Используя этот инициализатор, мы можем сделать это:

```
var direction = CompassDirection(degrees: 0) // Returns CompassDirection.north
direction = CompassDirection(degrees: 90) // Returns CompassDirection.east
print(direction.value) //prints 90
direction = CompassDirection(degrees: 500) // Returns nil
```

Перечисления разделяют многие функции с классами и структурами

Перечисления в Swift намного мощнее, чем некоторые из их коллег на других языках, таких как C. Они имеют множество функций с классами и структурами , такими как определение инициализаторов , вычисляемых свойств , методов экземпляров , соответствия протоколов и расширений .

```
protocol ChangesDirection {
    mutating func changeDirection()
}

enum Direction {

    // enumeration cases
    case up, down, left, right

    // initialise the enum instance with a case
    // that's in the opposite direction to another
    init(oppositeTo otherDirection: Direction) {
        self = otherDirection.opposite
    }

    // computed property that returns the opposite direction
    var opposite: Direction {
        switch self {
        case .up:
            return .down
        case .down:
            return .up
        case .left:
            return .right
        case .right:
            return .left
        }
    }
}
```

```
// extension to Direction that adds conformance to the ChangesDirection protocol
extension Direction: ChangesDirection {
    mutating func changeDirection() {
        self = .left
    }
}
```

```
var dir = Direction(oppositeTo: .down) // Direction.up

dir.changeDirection() // Direction.left

let opposite = dir.opposite // Direction.right
```

Вложенные перечисления

Вы можете встраивать перечисления один внутри другого, это позволяет вам структурировать иерархические перечисления, чтобы быть более организованными и понятными.

```
enum Orchestra {
    enum Strings {
        case violin
        case viola
        case cello
        case doubleBasse
    }

    enum Keyboards {
        case piano
        case celesta
        case harp
    }

    enum Woodwinds {
        case flute
        case oboe
        case clarinet
        case bassoon
        case contrabassoon
    }
}
```

И вы можете использовать его так:

```
let instrment1 = Orchestra.Strings.viola
let instrment2 = Orchestra.Keyboards.piano
```

Прочитайте Перечисления онлайн: https://riptutorial.com/ru/swift/topic/224/перечисления

глава 38: протоколы

Вступление

Протоколы - это способ указать, как использовать объект. Они описывают набор свойств и методов, которые должен предоставлять класс, структура или перечисление, хотя протоколы не создают ограничений для реализации.

замечания

Протокол Swift представляет собой набор требований, которые должны выполняться соответствующими типами. Протокол затем может использоваться в большинстве мест, где ожидается тип, например массивы и общие требования.

Члены протокола всегда используют один и тот же квалификатор доступа, что и весь протокол, и не могут указываться отдельно. Хотя протокол может ограничивать доступ с помощью требований геттера или сеттера, как описано выше.

Дополнительные сведения о протоколах см. В разделе Язык быстрого программирования .

Протоколы Objective-C аналогичны протоколам Swift.

Протоколы также сопоставимы с интерфейсами Java .

Examples

Основы протокола

О протоколах

Протокол определяет инициализаторы, свойства, функции, индексы и связанные типы, необходимые для типа объекта Swift (класс, структура или перечисление), соответствующего протоколу. На некоторых языках подобные идеи для спецификаций требований последующих объектов называются «интерфейсами».

Объявленный и определенный Протокол является Типом, сам по себе, с подписями его заявленных требований, несколько похожим на то, как Swift Functions являются типом, основанным на их сигнатуре параметров и возвратов.

Спецификации Swift Protocol могут быть необязательными, явно требуемыми и / или заданными реализациями по умолчанию через средство, известное как расширения протокола. Тип Swift Object (класс, структура или перечисление), желающий соответствовать Протоколу, который связан с расширениями для всех его требований, требует только заявить о своем желании соответствовать полностью соответствовать. Возможности реализации по умолчанию для расширений протокола могут быть достаточными для выполнения всех обязательств, связанных с Протоколом.

Протоколы могут быть унаследованы другими протоколами. Это, в сочетании с расширением протокола, означает, что протоколы могут и должны рассматриваться как важная функция Swift.

Протоколы и расширения важны для реализации более широких целей и подходов Swift к гибкости разработки и развития программ. Первичной целью Swift's Protocol and Extension является упрощение дизайна композиции в архитектуре и разработке программ. Это называется программно-ориентированным программированием. Крепкие старые таймеры считают, что это превосходит фокус на дизайне ООП.

Протоколы определяют интерфейсы, которые могут быть реализованы любой структурой , классом или перечислением :

```
protocol MyProtocol {
    init(value: Int)                    // required initializer
    func doSomething() -> Bool          // instance method
    var message: String { get }         // instance read-only property
```

```
    var value: Int { get set }            // read-write instance property
    subscript(index: Int) -> Int { get }  // instance subscript
    static func instructions() -> String  // static method
    static var max: Int { get }           // static read-only property
    static var total: Int { get set }     // read-write static property
}
```

Свойства, определенные в протоколах, должны быть аннотированы как { get } или { get set } . { get } означает, что свойство должно быть gettable, и поэтому оно может быть реализовано как *любое* свойство. { get set } означает, что свойство должно быть настраиваемым, а также gettable.

Структура, класс или перечисление могут **соответствовать** протоколу:

```
struct MyStruct : MyProtocol {
    // Implement the protocol's requirements here
}
class MyClass : MyProtocol {
    // Implement the protocol's requirements here
}
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // Implement the protocol's requirements here
}
```

Протокол также может определять **реализацию** по **умолчанию** для любого из своих требований через расширение :

```
extension MyProtocol {

    // default implementation of doSomething() -> Bool
    // conforming types will use this implementation if they don't define their own
    func doSomething() -> Bool {
        print("do something!")
        return true
    }
}
```

Протокол может **использоваться как тип** , при условии, что он не имеет associatedtype **типов** требований :

```
func doStuff(object: MyProtocol) {
    // All of MyProtocol's requirements are available on the object
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Вы также можете определить абстрактный тип, который соответствует **нескольким** протоколам:

3.0

С Swift 3 или выше это делается путем разделения списка протоколов с амперсандом (&):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

3.0

В старых версиях есть protocol<...> синтаксиса protocol<...> где протоколы представляют собой список, разделенный запятыми между угловыми скобками <> .

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // All of MyProtocol & AnotherProtocol's requirements are available on the object
    print(object.message)
    object.doSomethingElse()
}

// MyStruct, MyClass & MyEnum must now conform to both MyProtocol & AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Существующие типы могут быть **расширены**, чтобы соответствовать протоколу:

```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Требования к связанным типам

Протоколы могут определять **связанные требования типа** с использованием ключевого слова associatedtype :

```
protocol Container {
    associatedtype Element
    var count: Int { get }
    subscript(index: Int) -> Element { get set }
}
```

Протоколы со связанными требованиями типа **могут использоваться только как общие ограничения** :

```
// These are NOT allowed, because Container has associated type requirements:
func displayValues(container: Container) { ... }
class MyClass { let container: Container }
// > error: protocol 'Container' can only be used as a generic constraint
// > because it has Self or associated type requirements

// These are allowed:
func displayValues<T: Container>(container: T) { ... }
class MyClass<T: Container> { let container: T }
```

Тип, который соответствует протоколу, неявно может удовлетворять требованию associatedtype , предоставляя заданный тип, в котором протокол ожидает появления associatedtype :

```
struct ContainerOfOne<T>: Container {
    let count = 1          // satisfy the count requirement
    var value: T

    // satisfy the subscript associatedtype requirement implicitly,
    // by defining the subscript assignment/return type as T
    // therefore Swift will infer that T == Element
    subscript(index: Int) -> T {
        get {
            precondition(index == 0)
```

```
            return value
        }
        set {
            precondition(index == 0)
            value = newValue
        }
    }
}

let container = ContainerOfOne(value: "Hello")
```

(Обратите внимание, что для того, чтобы добавить ясность в этот пример, общий тип заполнителя называется `T` - более подходящим именем будет `Element` , который будет затенять `associatedtype Element` с `associatedtype Element` протокола. Компилятор все равно выведет, что общий `Element` заполнитель используется для удовлетворения `associatedtype Element` Требование `associatedtype Element` .)

associatedtype typealias также может быть удовлетворен явно посредством использования typealias :

```
struct ContainerOfOne<T>: Container {

    typealias Element = T
    subscript(index: Int) -> Element { ... }

    // ...
}
```

То же самое касается расширений:

```
// Expose an 8-bit integer as a collection of boolean values (one for each bit).
extension UInt8: Container {

    // as noted above, this typealias can be inferred
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

Если соответствующий тип уже удовлетворяет требованию, реализация не требуется:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

Модель делегата

Делегат - это общий шаблон проектирования, используемый в рамках Cocoa и CocoaTouch, где один

класс делегирует ответственность за реализацию некоторых функций другому. Это следует принципу разделения проблем, когда класс framework реализует общие функции, в то время как отдельный экземпляр делегата реализует конкретный прецедент.

Еще один способ взглянуть на шаблон делегата - с точки зрения связи объектов. Objects часто приходится разговаривать друг с другом, и для этого объект должен соответствовать protocol , чтобы стать делегатом другого объекта. Как только эта настройка будет выполнена, другой объект вернется к своим делегатам, когда произойдет интересное.

Например, представление в пользовательском интерфейсе для отображения списка данных должно отвечать только за логику отображения данных, а не за определение того, какие данные должны отображаться.

Давайте перейдем к более конкретному примеру. если у вас есть два класса: родительский и дочерний:

```
class Parent { }
class Child { }
```

И вы хотите уведомить родителя об изменении из ребенка.

В Swift делегаты реализуются с использованием объявления protocol поэтому мы объявим protocol который будет реализован delegate . Здесь делегат является parent объектом.

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

Ребенок должен объявить свойство для хранения ссылки на делегат:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Обратите внимание, что delegate переменной является факультативным, а протокол ChildDelegate отмечен как реализуемый только типом класса (без этого переменная delegate не может быть объявлена как weak ссылка, избегая любого цикла сохранения. Это означает, что если переменная delegate больше не является упоминается где-нибудь еще, он будет выпущен). Это значит, что родительский класс регистрирует делегат только тогда, когда он необходим и доступен.

Также для того, чтобы отметить, что наш делегат weak мы должны ограничить наш протокол ChildDelegate ссылкой на типы, добавив ключевое слово class в объявление протокола.

В этом примере, когда ребенок что-то делает и ему нужно уведомить своего родителя, ребенок вызовет:

```
delegate?.childDidSomething()
```

Если делегат был определен, делегат будет уведомлен о том, что ребенок что-то сделал.

Родительский класс должен будет продлить протокол ChildDelegate , чтобы иметь возможность реагировать на его действия. Это можно сделать непосредственно в родительском классе:

```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("Yay!")
    }
}
```

Или используя расширение:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

Родитель также должен сообщить ребенку, что он является делегатом ребенка:

```
// In the parent
let child = Child()
child.delegate = self
```

По умолчанию protocol Swift не позволяет реализовать дополнительную функцию. Они могут быть указаны только в том случае, если ваш протокол отмечен атрибутом @objc и optional модификатором.

Например, UITableView реализует общее поведение представления таблицы в iOS, но пользователь должен реализовать два класса делегатов, называемые UITableViewDelegate и UITableViewDataSource которые реализуют способы отображения конкретных ячеек и поведения.

```
@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate {

        // Display customization
        optional public func tableView(tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        optional public func tableView(tableView: UITableView, willDisplayHeaderView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, willDisplayFooterView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, didEndDisplayingCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        ...
}
```

Вы можете реализовать этот протокол, изменив определение класса, например:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Любые методы, не отмеченные как optional в определении протокола (UITableViewDelegate в этом случае), должны быть реализованы.

Расширение протокола для определенного класса соответствия

Вы можете написать **реализацию протокола** по **умолчанию** для определенного класса.

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController default protocol implementation")
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
```

```
vc.doSomething() // Prints "UIViewController default protocol implementation"
```

Использование протокола RawRepresentable (Extensible Enum)

```
// RawRepresentable has an associatedType RawValue.
// For this struct, we will make the compiler infer the type
// by implementing the rawValue variable with a type of String
//
// Compiler infers RawValue = String without needing typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

Эта структура может быть расширена в другом месте, чтобы добавить случаи

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

И интерфейс может проектироваться вокруг любого RawRepresentable типа или, в частности, вашей структуры перечисления

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

На сайте вызова вы можете использовать сокращенную синтаксическую разметку для typeafe NotificationName

```
post(notification: .dataFinished)
```

Использование общей функции RawRepresentable

```
// RawRepresentable has an associate type, so the
// method that wants to accept any type conforming to
// RawRepresentable needs to be generic
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

Протоколы только для классов

Протокол может указывать, что только класс может реализовать его с помощью ключевого слова class в своем списке наследования. Это ключевое слово должно отображаться перед любыми другими унаследованными протоколами в этом списке.

```
protocol ClassOnlyProtocol: class, SomeOtherProtocol {
    // Protocol requirements
}
```

Если тип non-class пытается реализовать ClassOnlyProtocol , будет генерироваться ошибка компилятора.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

Другие протоколы могут наследоваться от ClassOnlyProtocol , но они будут иметь одно и то же требование к классу.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Справочная семантика протоколов класса

Использование протокола только для классов позволяет использовать ссылочную семантику, когда соответствующий тип неизвестен.

```
protocol Foo : class {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {

    // this assignment requires reference semantics,
    // as foo is a let constant in this scope.
    foo.bar = "new value"
}
```

В этом примере, поскольку Foo - это протокол только для классов, назначение в bar является допустимым, поскольку компилятор знает, что foo является типом класса и поэтому имеет ссылочную семантику.

Если Foo не был протоколом только для класса, была бы допущена ошибка компилятора - поскольку соответствующий тип может быть типом значения , для которого требуется аннотация var , чтобы быть изменчивым.

```
protocol Foo {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {
    foo.bar = "new value" // error: Cannot assign to property: 'foo' is a 'let' constant
}
```

```
func takesAFoo(foo:Foo) {
    var foo = foo // mutable copy of foo
    foo.bar = "new value" // no error - satisfies both reference and value semantics
}
```

Слабые переменные типа протокола

При применении weak модификатора к переменной типа протокола этот тип протокола должен быть только классом, так как weak может применяться только к ссылочным типам.

```
weak var weakReference : ClassOnlyProtocol?
```

Реализация протокола Hashable

Типы, используемые в Sets и Hashable Dictionaries(key) должны соответствовать протоколу Hashable , который наследуется от Equatable protocol.

Пользовательский тип, соответствующий протоколу Hashable , должен реализовывать

- Вычисленное свойство hashValue
- Определите один из операторов равенства ie == or != .

В следующем примере реализуется протокол Hashable для настраиваемой struct :

```
struct Cell {
    var row: Int
    var col: Int

    init(_ row: Int, _ col: Int) {
        self.row = row
        self.col = col
    }
}

extension Cell: Hashable {

    // Satisfy Hashable requirement
    var hashValue: Int {
        get {
            return row.hashValue^col.hashValue
        }
    }

    // Satisfy Equatable requirement
    static func ==(lhs: Cell, rhs: Cell) -> Bool {
        return lhs.col == rhs.col && lhs.row == rhs.row
    }

}

// Now we can make Cell as key of dictonary
var dict = [Cell : String]()

dict[Cell(0, 0)] = "0, 0"
dict[Cell(1, 0)] = "1, 0"
dict[Cell(0, 1)] = "0, 1"

// Also we can create Set of Cells
var set = Set<Cell>()

set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

Примечание . Нет необходимости, чтобы разные значения в пользовательском типе имели разные значения хэширования, допустимы конфликты. Если значения хэша равны, оператор равенства будет использоваться для определения реального равенства.

Прочитайте протоколы онлайн: https://riptutorial.com/ru/swift/topic/241/протоколы

глава 39: Работа с C и Objective-C

замечания

Для получения дополнительной информации см. Документацию Apple по использованию Swift с Cocoa и Objective-C .

Examples

Использование классов Swift из кода Objective-C

В том же модуле

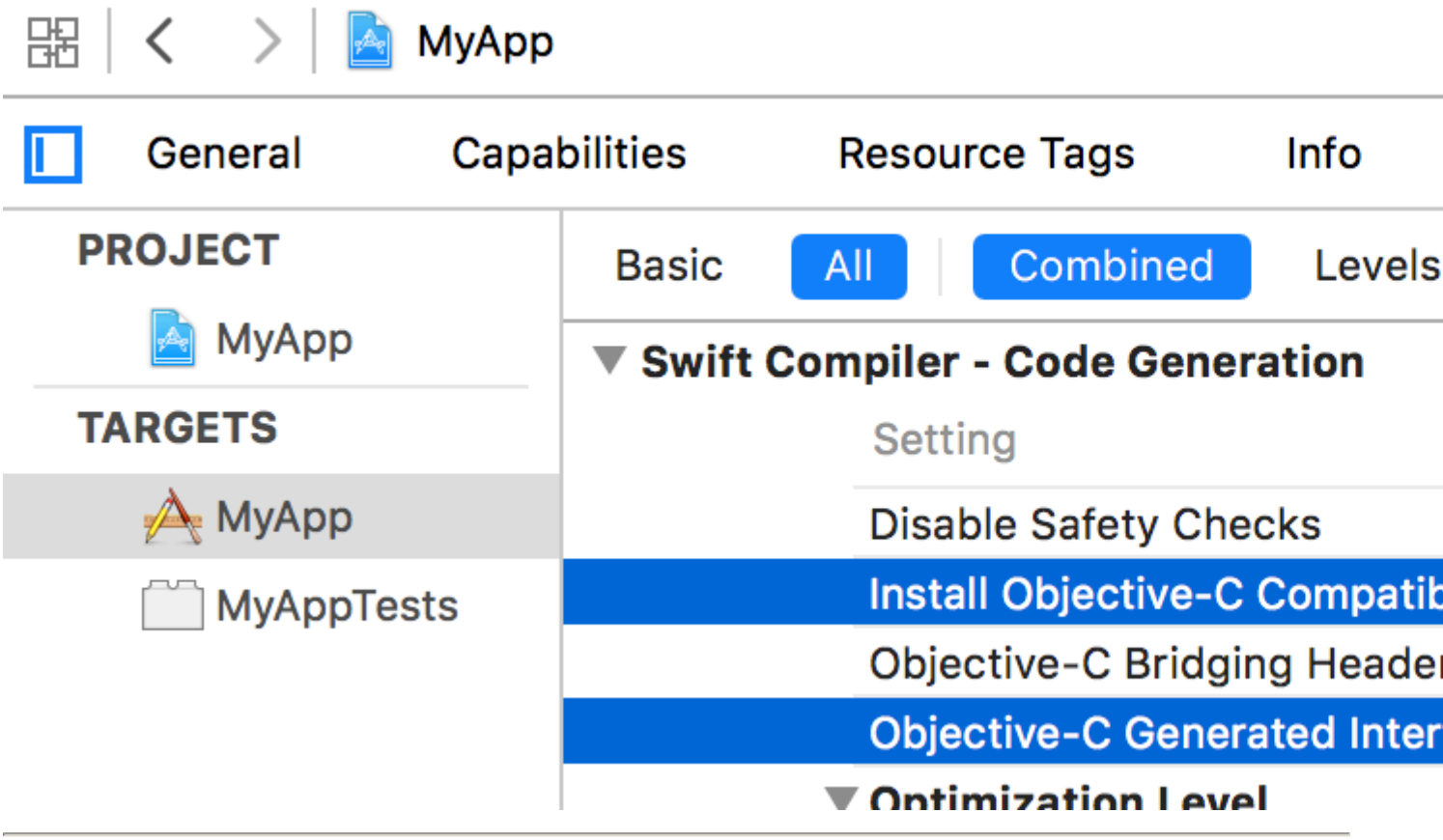
Внутри модуля с именем « **MyModule** » Xcode генерирует заголовок с именем **MyModule-Swift.h** который предоставляет общедоступные классы Swift для Objective-C. Импортируйте этот заголовок, чтобы использовать классы Swift:

```
// MySwiftClass.swift in MyApp
import Foundation

// The class must be `public` to be visible, unless this target also has a bridging header
public class MySwiftClass: NSObject {
    // ...
}
```

```
// MyViewController.m in MyApp

#import "MyViewController.h"
#import "MyApp-Swift.h"                     // import the generated interface
#import <MyFramework/MyFramework-Swift.h>  // or use angle brackets for a framework target

@implementation MyViewController
- (void)demo {
    [[MySwiftClass alloc] init];  // use the Swift class
}
@end
```

Соответствующие настройки сборки:

- **Objective-C Generated Interface Header Name** : управляет именем сгенерированного заголовка Obj-C.
- **Установите заголовок совместимости Objective-C** : должен ли заголовок -Swift.h быть общедоступным заголовком (для целей инфраструктуры).



В другом модуле

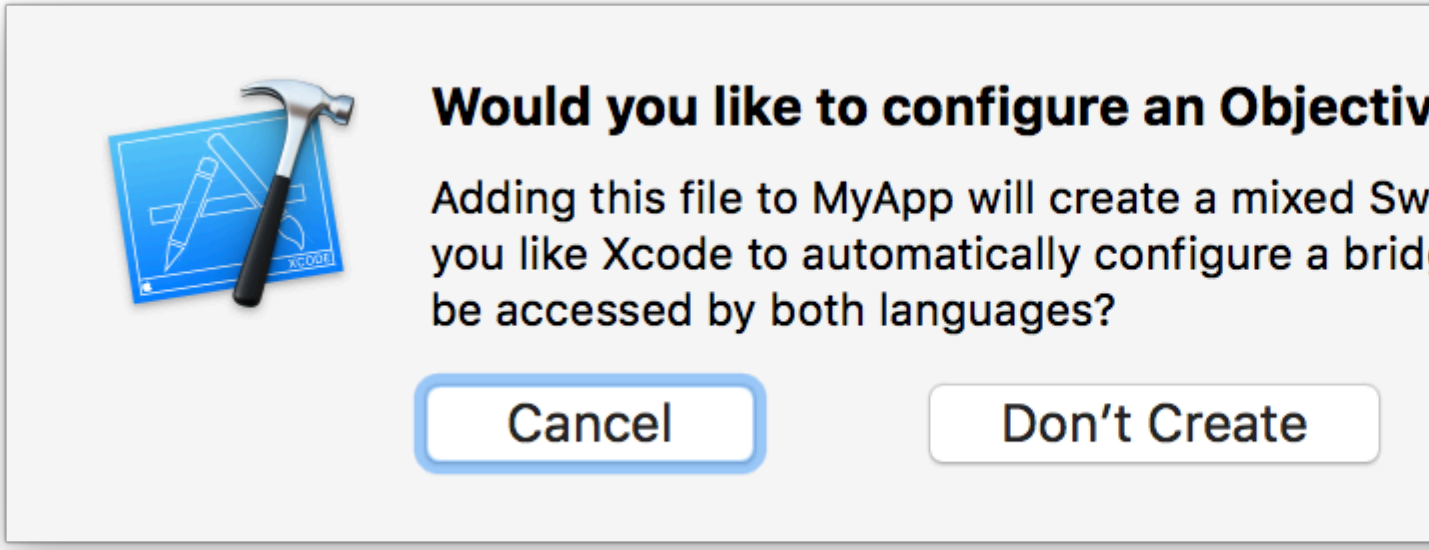
Использование **@import MyFramework;** импортирует весь модуль, включая интерфейсы Obj-C, к классам Swift (если указан вышеупомянутый параметр сборки).

Использование классов Objective-C из кода Swift

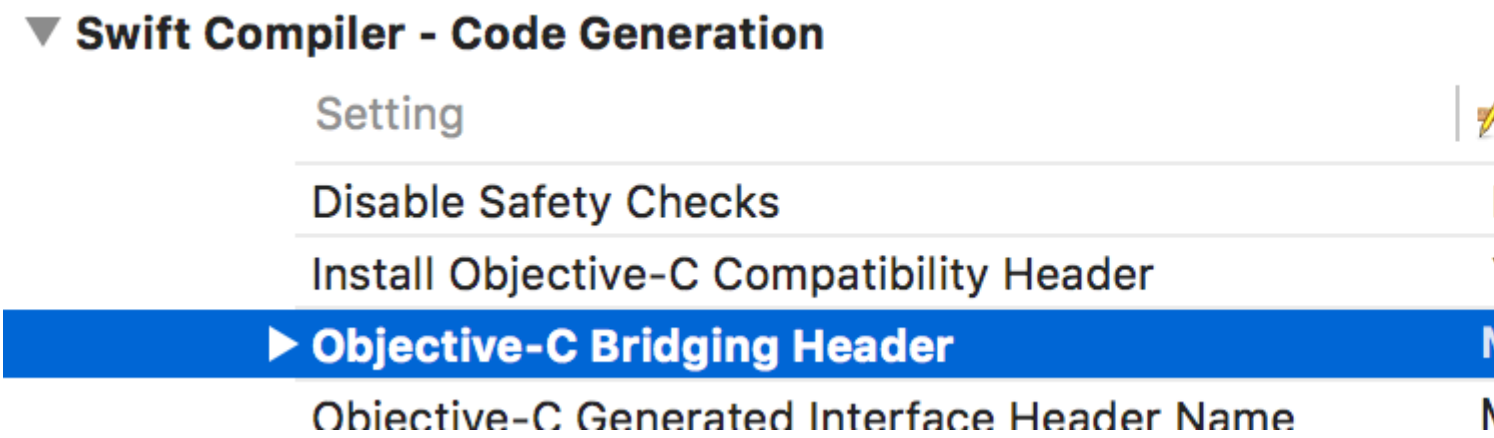
Если MyFramework содержит классы Objective-C в своих публичных заголовках (и заголовок зонтика), тогда **import MyFramework** - это все, что необходимо для их использования из Swift.

Мостовые заголовки

Мостовой заголовок делает дополнительные объявления Objective-C и C видимыми для кода Swift. При добавлении файлов проекта Xcode может предлагать автоматически создавать заголовок моста:



Чтобы создать его вручную, измените настройку сборки **заголовка Objective-C Bridging Header** :

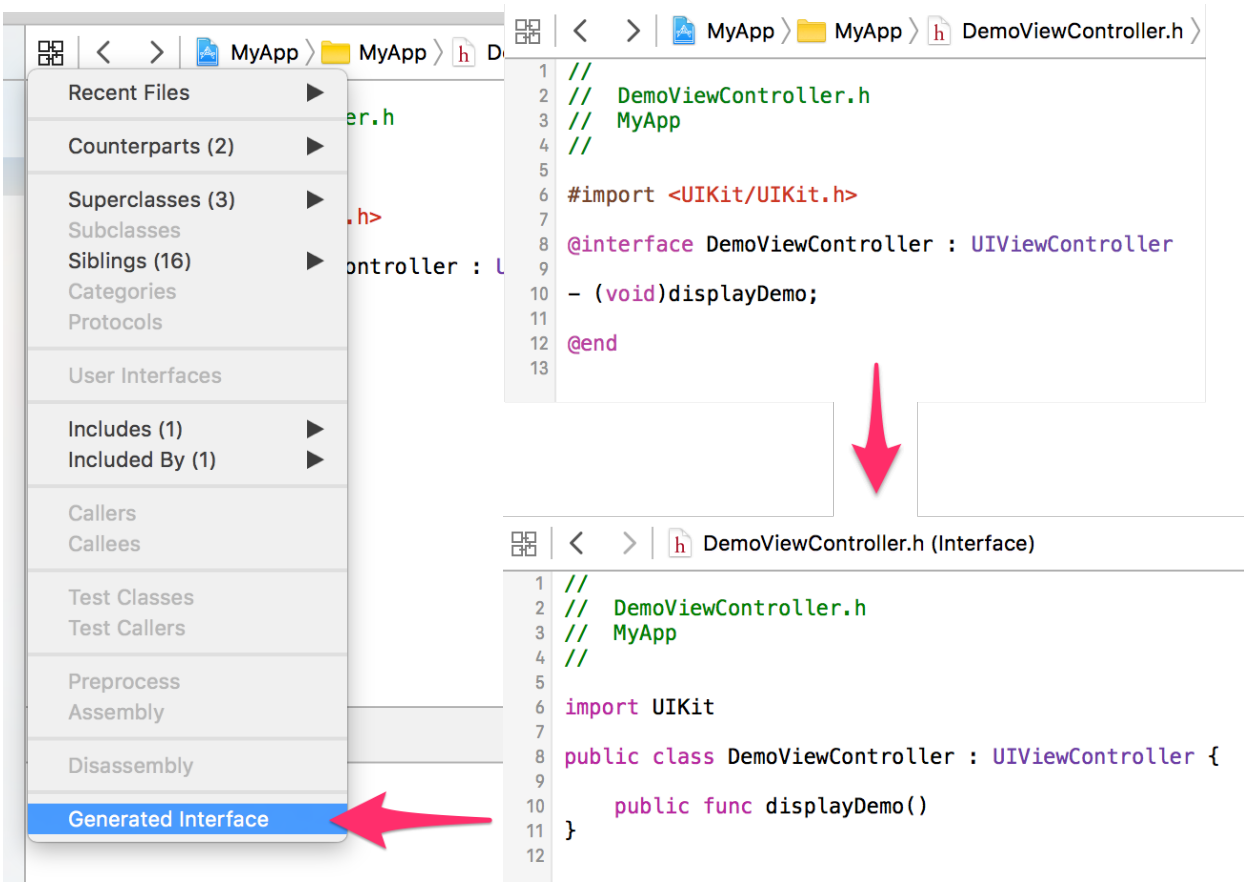


Внутри соединительного заголовка импортируйте файлы, которые необходимо использовать из кода:

```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // allows code in this module to use MyClass
```

Сгенерированный интерфейс

Нажмите кнопку Связанные элементы (или нажмите ^ 1), затем выберите « **Сгенерированный интерфейс»**, чтобы увидеть интерфейс Swift, который будет создан из заголовка Objective-C.



Укажите мостиковый заголовок для swiftc

-import-objc-header указывает заголовок для импорта swiftc :

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

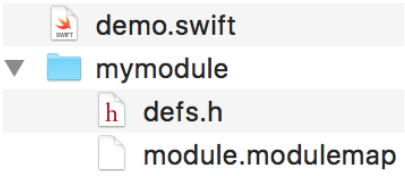
```
// demo.swift
extension Color: CustomStringConvertible {  // extension on a C struct
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C macro becomes a constant
let color = Color(red: 0xCA, green: 0xCA, blue: 0xD0)  // C struct initializer
print("The color is \(color)")
```

```
$ swiftc demo.swift -import-objc-header defs.h && ./demo
MAX_VALUE is: 255
The color is Color(red: 202, green: 202, blue: 208)
```

Используйте карту модуля для импорта заголовков C

Карта модуля может просто import mymodule , настроив его на чтение файлов заголовков C и превращая их в функции Swift.

Поместите файл с именем module.modulemap в каталог с именем mymodule :



Внутри файла карты модуля:

```
// mymodule/module.modulemap
module mymodule {
    header "defs.h"
}
```

Затем import модуль:

```
// demo.swift
import mymodule
print("Empty color: \(Color())")
```

Используйте флаг -I *directory* чтобы сообщить swiftc где найти модуль:

```
swiftc -I . demo.swift   # "-I ." means "search for modules in the current directory"
```

Дополнительные сведения о синтаксисе карты модуля см. В документации Clang о картах модулей .

Мелкозернистая взаимосвязь между Objective-C и Swift

Когда API помечается NS_REFINED_FOR_SWIFT , при импорте в Swift он будет иметь префикс с двумя NS_REFINED_FOR_SWIFT подчеркивания (__):

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj NS_REFINED_FOR_SWIFT;
@end
```

Сгенерированный интерфейс выглядит следующим образом:

```
public class MyClass : NSObject {
    public func __indexOfObject(obj: AnyObject) -> Int
}
```

Теперь вы можете **заменить API** на более «Swifty» расширение. В этом случае мы можем использовать необязательное возвращаемое значение, отфильтровывая NSNotFound :

```
extension MyClass {
    // Rather than returning NSNotFound if the object doesn't exist,
    // this "refined" API returns nil.
    func indexOfObject(obj: AnyObject) -> Int? {
        let idx = __indexOfObject(obj)
        if idx == NSNotFound { return nil }
        return idx
    }
}

// Swift code, using "if let" as it should be:
let myobj = MyClass()
if let idx = myobj.indexOfObject(something) {
```

```
    // do something with idx
}
```

В большинстве случаев вы можете ограничить, может ли аргумент функции Objective-C быть nil . Это делается с использованием _Nonnull слова _Nonnull , которое квалифицирует любую ссылку указателя или блока:

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // complex asynchronous code
}
```

С учетом этого компилятор должен испускать ошибку всякий раз, когда мы пытаемся передать nil этой функции из нашего кода Swift:

```
doStuff(
    nil,  // error: nil is not compatible with expected argument type 'UnsafeRawPointer'
    nil)  // error: nil is not compatible with expected argument type '() -> Void'
```

Противоположность _Nonnull _Nullable , что означает, что допустимо пропускать nil в этом аргументе. _Nullable также является значением по умолчанию; однако указание этого явно позволяет использовать более самодокументированный и будущий код.

Чтобы еще больше помочь компилятору с оптимизацией кода, вы также можете указать, будет ли блок экранироваться:

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f is not stored anywhere
}
```

С помощью этого атрибута мы обещаем не сохранять ссылку на блок, а не вызывать блок после завершения функции.

Используйте стандартную библиотеку C

Совместимость Swift C позволяет использовать функции и типы из стандартной библиотеки C.

В Linux стандартная библиотека C отображается через модуль Glibc ; на платформах Apple это называется Darwin .

```
#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

// use open(), read(), and other libc features
```

Прочитайте Работа с C и Objective-C онлайн: https://riptutorial.com/ru/swift/topic/421/работа-с-c-и-objective-c

глава 40: Разметка документации

Examples

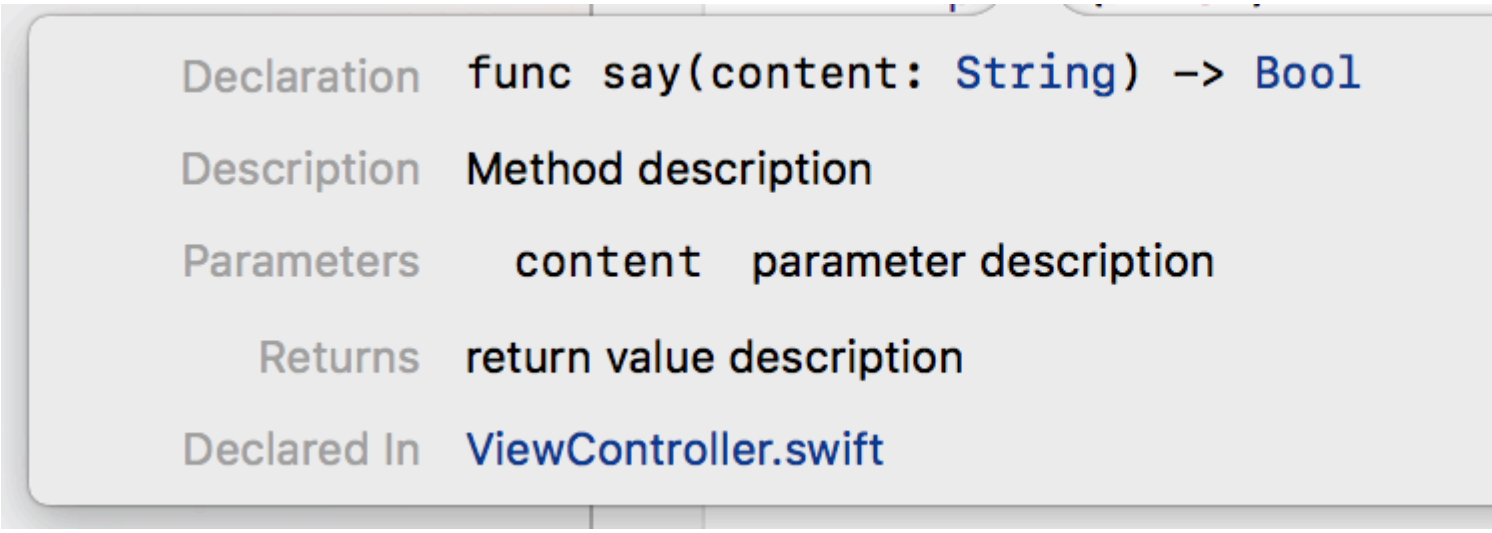
Документация класса

Вот пример документации по базовому классу:

```
/// Class description
class Student {

    // Member description
    var name: String

    /// Method description
    ///
    /// - parameter content:   parameter description
    ///
    /// - returns: return value description
    func say(content: String) -> Bool {
        print("\(self.name) say \(content)")
        return true
    }
}
```

Обратите внимание, что с помощью *Xcode 8* вы можете сгенерировать фрагмент документации с помощью команды + option + / .

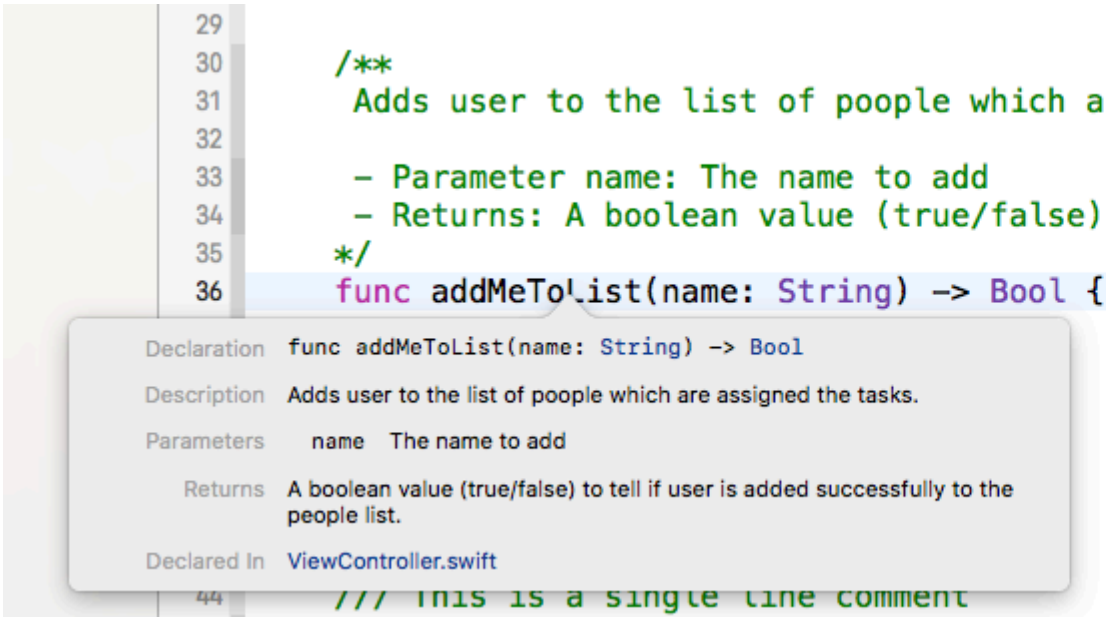
Это вернет:



Стили документации

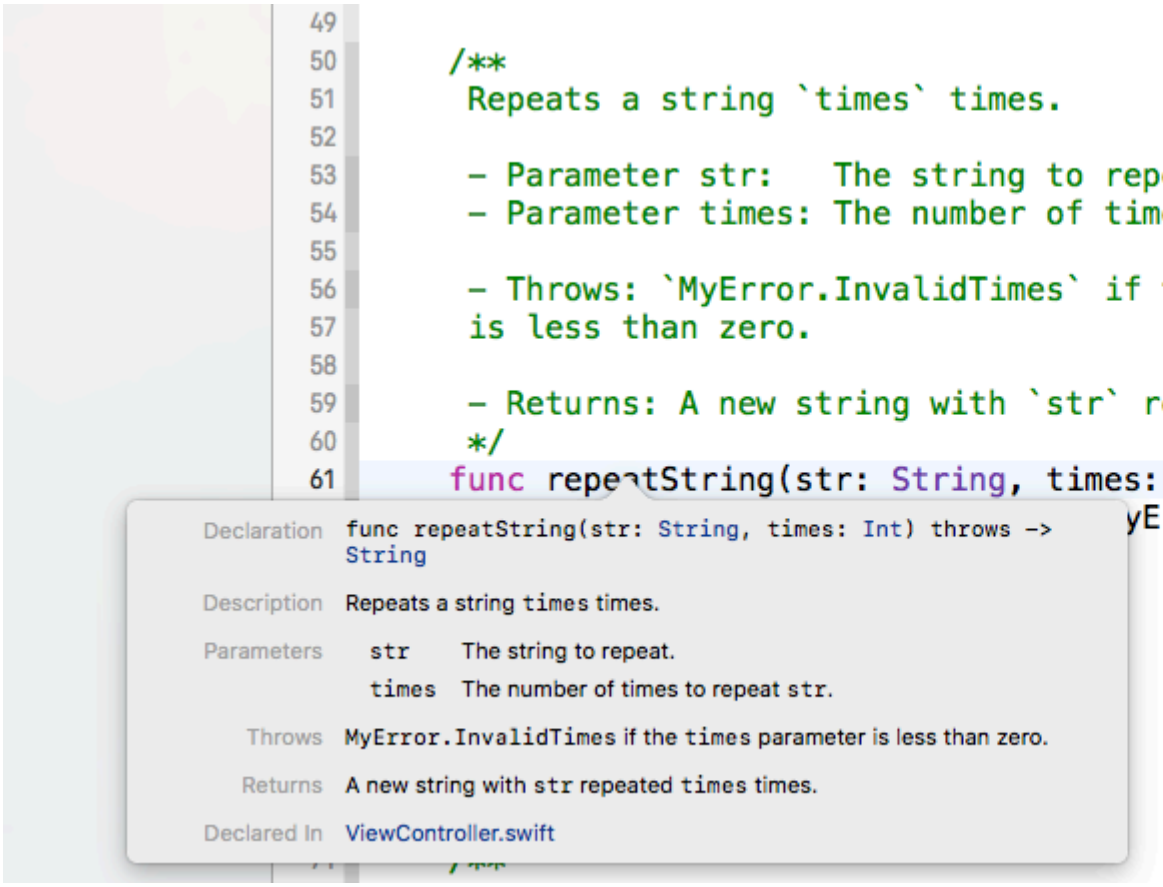
```
/**
 Adds user to the list of poople which are assigned the tasks.

 - Parameter name: The name to add
 - Returns: A boolean value (true/false) to tell if user is added successfully to the people
list.
*/
func addMeToList(name: String) -> Bool {

    // Do something....
```

```
    return true
}
```

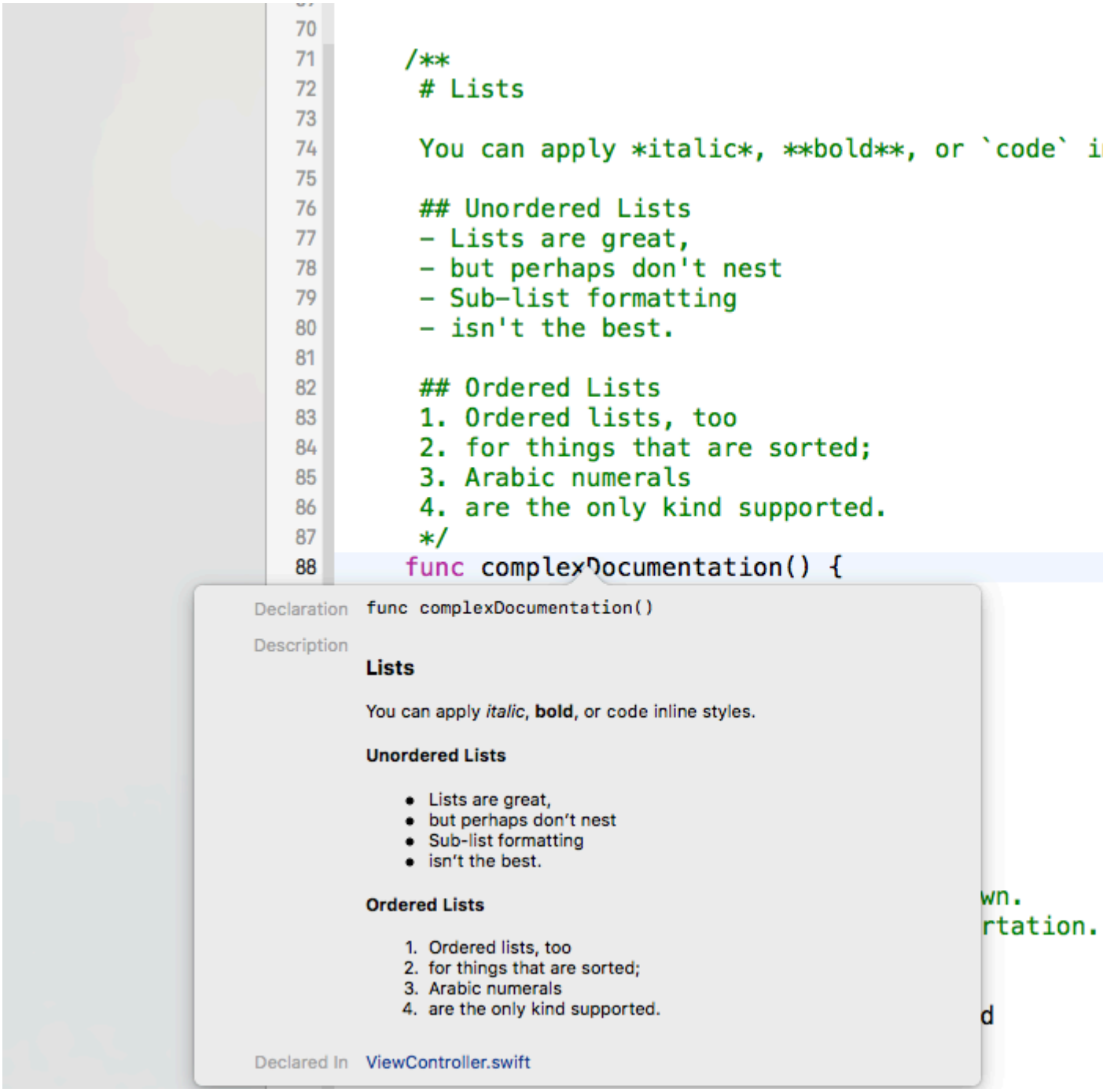



```
/// This is a single line comment
func singleLineComment() {

}
```

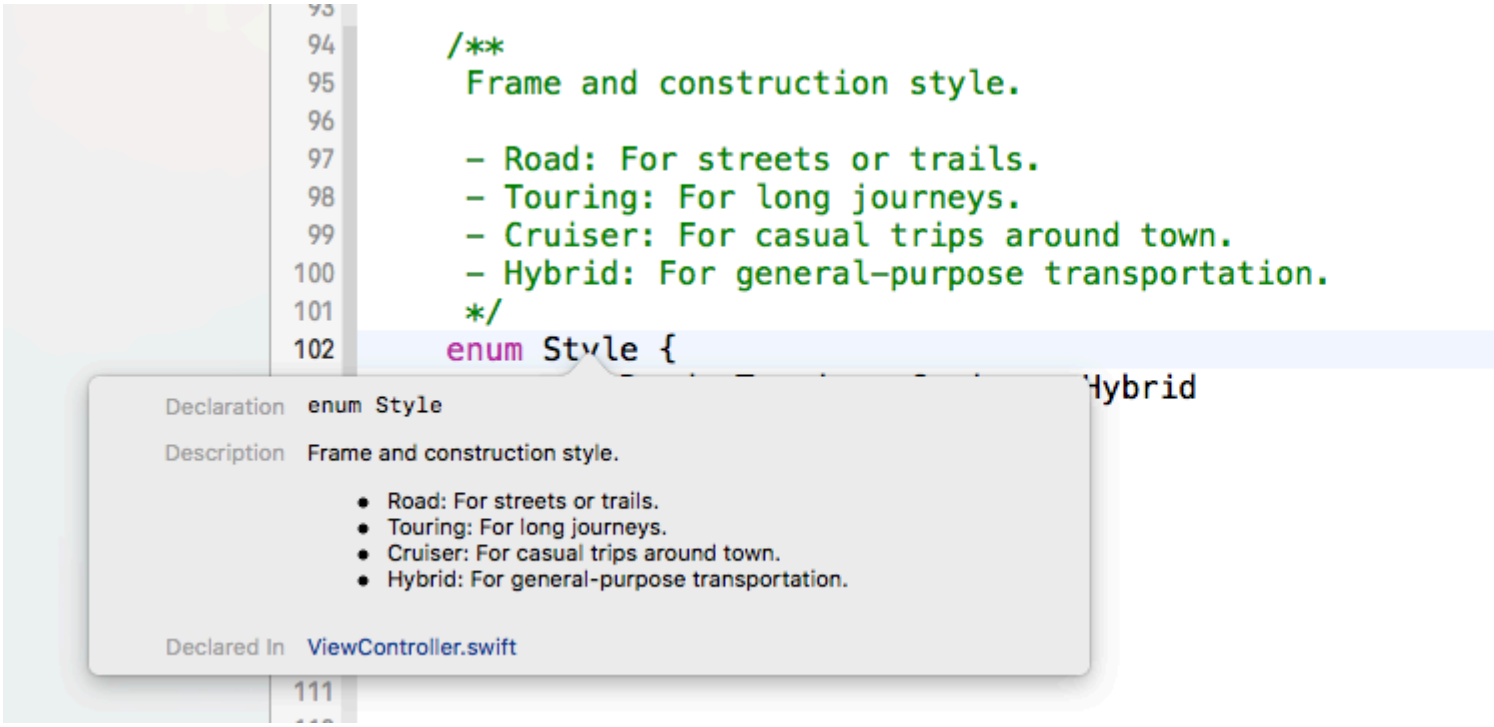

```
/**
 Repeats a string `times` times.

 - Parameter str:   The string to repeat.
 - Parameter times: The number of times to repeat `str`.

 - Throws: `MyError.InvalidTimes` if the `times` parameter
 is less than zero.

 - Returns: A new string with `str` repeated `times` times.
*/
func repeatString(str: String, times: Int) throws -> String {
    guard times >= 0 else { throw MyError.invalidTimes }
    return "Hello, world"
}
```




```
/**
 # Lists

 You can apply *italic*, **bold**, or `code` inline styles.

 ## Unordered Lists
 - Lists are great,
 - but perhaps don't nest
 - Sub-list formatting
 - isn't the best.

 ## Ordered Lists
 1. Ordered lists, too
 2. for things that are sorted;
 3. Arabic numerals
 4. are the only kind supported.
*/
func complexDocumentation() {

}
```




```
/**
 Frame and construction style.

 - Road: For streets or trails.
 - Touring: For long journeys.
 - Cruiser: For casual trips around town.
 - Hybrid: For general-purpose transportation.
*/
enum Style {
    case Road, Touring, Cruiser, Hybrid
}
```



Прочитайте Разметка документации онлайн: https://riptutorial.com/ru/swift/topic/6937/разметка-документации

глава 41: расширения

замечания

Вы можете узнать больше о расширениях на языке Swift Programming .

Examples

Переменные и функции

Расширения могут содержать функции и вычисленные / константные переменные get. Они в формате

```
extension ExtensionOf {
    //new functions and get-variables
}
```

Чтобы ссылаться на экземпляр расширенного объекта, можно использовать self , так же, как его можно использовать

Чтобы создать расширение String которое добавляет функцию .length() которая возвращает длину строки, например

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

Расширения также могут содержать переменные get . Например, добавив переменную .length в строку, которая возвращает длину строки

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

Инициализаторы в расширениях

Расширения могут содержать инициализаторы удобства. Например, неудачный инициализатор для Int который принимает NSString :

```
extension Int {
    init?(_ string: NSString) {
        self.init(string as String)  // delegate to the existing Int.init(String) initializer
    }
}

let str1: NSString = "42"
Int(str1) // 42
```

```
let str2: NSString = "abc"
Int(str2) // nil
```

Что такое расширения?

Расширения используются для расширения функциональности существующих типов в Swift. Расширения могут добавлять индексы, функции, инициализаторы и вычисленные свойства. Они также могут создавать типы, соответствующие протоколам .

Предположим, вы хотите иметь возможность вычислить факториал Int . Вы можете добавить вычисленное свойство в расширение:

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

Затем вы можете получить доступ к объекту так же, как если бы он был включен в оригинальный Int API.

```
let val1: Int = 10

val1.factorial  // returns 3628800
```

Расширения протокола

Очень полезной особенностью Swift 2.2 является возможность расширения протоколов.

Он работает в значительной степени подобно абстрактным классам при рассмотрении функциональности, которую вы хотите получить во всех классах, реализующих некоторый протокол (без наследования от базового общего класса).

```
protocol FooProtocol {
    func doSomething()
}

extension FooProtocol {
    func doSomething() {
        print("Hi")
    }
}

class Foo: FooProtocol {
    func myMethod() {
        doSomething() // By just implementing the protocol this method is available
    }
}
```

Это также возможно с помощью дженериков.

ограничения

Можно написать метод на родовом типе, который более ограничительный, используя предложение.

```
extension Array where Element: StringLiteralConvertible {
  func toUpperCase() -> [String] {
```

```
      var result = [String]()
      for value in self {
          result.append(String(value).uppercaseString)
      }
      return result
    }
}
```

Пример использования

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

Что такое расширения и когда их использовать

Расширения добавляют новые функции к существующему классу, структуре, перечислению или типу протокола. Это включает в себя возможность расширения типов, для которых у вас нет доступа к исходному исходному коду.

Расширения в Swift могут:

- Добавить вычисленные свойства и свойства вычисленного типа
- Определение методов экземпляра и методов типа
- Предоставить новые инициализаторы
- Определение индексов
- Определить и использовать новые вложенные типы
- Сделать существующий тип соответствующим протоколу

Когда использовать Swift Extensions:

- Дополнительная функциональность для Swift
- Дополнительные возможности для UIKit / Foundation
- Дополнительная функциональность без возиться с кодом других лиц
- Классы разбивки на: Данные / Функциональность / Делегат

Когда не использовать:

- Расширьте свои собственные классы из другого файла

Простой пример:

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

Источник

Нижние индексы

Расширения могут добавлять новые индексы к существующему типу.

Этот пример получает символ внутри строки, используя данный индекс:

2,2

```
extension String {
    subscript(index: Int) -> Character {
        let newIndex = startIndex.advancedBy(index)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

3.0

```
extension String {
    subscript(offset: Int) -> Character {
        let newIndex = self.index(self.startIndex, offsetBy: offset)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

Прочитайте расширения онлайн: https://riptutorial.com/ru/swift/topic/324/расширения

глава 42: Расширенные операторы

Examples

Пользовательские операторы

Swift поддерживает создание пользовательских операторов. Новые операторы объявляются на глобальном уровне с использованием ключевого слова operator .

Структура оператора определяется тремя частями: размещение операндов, приоритет и ассоциативность.

1. Модификаторы prefix , infix и postfix используются для запуска пользовательской декларации оператора. Модификаторы prefix и postfix объявляют, должен ли оператор быть до или после, соответственно, значением, на котором он действует. Такие операторы норны, как 8 и 3++ ** , поскольку они могут действовать только на одну цель. infix объявляет двоичный оператор, который действует на два значения между ними, например 2+3 .

2. Сначала вычисляются операторы с более высоким **приоритетом** . По умолчанию приоритет оператора выше ? ... : (значение 100 в Swift 2.x). Очередность стандартных операторов Swift можно найти здесь .

3. **Ассоциативность** определяет порядок операций между операторами с одинаковым приоритетом. Левые ассоциативные операторы вычисляются слева направо (порядок чтения, как и большинство операторов), а правые ассоциативные операторы вычисляют справа налево.

3.0

Начиная с Swift 3.0, можно определить приоритет и ассоциативность в **группе приоритетов** вместо самого оператора, так что несколько операторов могут легко использовать одинаковый приоритет, не обращаясь к загадочным номерам. Список стандартных групп приоритетов показан ниже .

Операторы возвращают значения на основе кода расчета. Этот код действует как нормальная функция с параметрами, определяющими тип ввода и ключевое слово return определяющее вычисленное значение, возвращаемое оператором.

Вот определение простого экспоненциального оператора, так как стандартный Swift не имеет экспоненциального оператора.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

infix говорит, что оператор ** работает между двумя значениями, такими как 9**2 . Поскольку функция оставила ассоциативность, 3**3**2 вычисляется как (3**3)**2 . Приоритет 170 выше всех стандартных операций Swift, что означает, что 3+2**4 вычисляет до 19 , несмотря на левую ассоциативность ** .

3.0

```
import Foundation

infix operator **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}
```

Вместо того, чтобы явно указывать приоритет и ассоциативность, в Swift 3.0 мы могли бы использовать встроенную группу приоритетов BitwiseShiftPrecedence, которая дает правильные значения (такие же, как << , >>).

**: Приращение и декремент устарели и будут удалены в Swift 3.

Перегрузка + для словарей

Поскольку в настоящее время нет простого способа комбинирования словарей в Swift, может быть полезно перегрузить операторы + и += , чтобы добавить эту функциональность с помощью дженериков .

```
// Combines two dictionaries together. If both dictionaries contain
// the same key, the value of the right hand side dictionary is used.
func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
        combined[key] = value
    }
    return combined
}

// The mutable variant of the + overload, allowing a dictionary
// to be appended to 'in-place'.
func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
        lhs[key] = value
    }
}
```

3.0

Начиная с Swift 3, inout следует помещать перед типом аргумента.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Пример использования:

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

Коммутативные операторы

Давайте добавим пользовательский оператор для умножения CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Теперь это работает

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

Но если мы попытаемся сделать операцию в обратном порядке, мы получим ошибку

```
let sizeC = sizeB * 20          // ERROR
```

Но достаточно просто добавить:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
}
```

Теперь оператор коммутативен.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Побитовые операторы

Быстрые операторы Swift позволяют выполнять операции над двоичной формой чисел. Вы можете указать бинарный литерал, предварительно 0b110 номер 0b , поэтому, например, 0b110 эквивалентен двоичному номеру 110 (десятичное число 6). Каждый 1 или 0 бит в числе.

Побитовое NOT ~ :

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Здесь каждый бит изменяется на противоположное. Объявление числа как явно UInt8 гарантирует, что число положительно (так что нам не нужно иметь дело с негативами в примере) и что это всего 8 бит. Если 0b01101100 был большим UInt, то были бы ведущие 0s, которые были бы преобразованы в 1s и стали значительными при инверсии:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 -> 1
- 1 -> 0

Побитовое И & :

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Здесь данный бит будет равен 1, если и только если двоичные числа с обеих сторон оператора & содержали 1 в этом месте бит.

- 0 & 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

Побитовое ИЛИ | :

```
var number = 0b0110
let newNumber = number | 0b1000
```

```
// newNumber is equal to 0b1110
```

Здесь данный бит будет равен 1, если и только если двоичное число, по крайней мере, на одной стороне | оператор содержал 1 в этом месте бит.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

Побитовое XOR (Исключительное ИЛИ) ^ :

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Здесь данный бит будет равен 1, если и только если бит в этом положении двух операндов отличается.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

Для всех двоичных операций порядок операндов не влияет на результат.

Операторы переполнения

Переполнение относится к тому, что происходит, когда операция приведет к числу, которое больше или меньше, чем указанное количество бит для этого числа может удерживаться.

Из-за того, как работает двоичная арифметика, после того, как число становится слишком большим для своих битов, число переполняется до наименьшего возможного числа (для размера бита), а затем продолжает подсчитывать оттуда. Точно так же, когда число становится слишком маленьким, оно переполняется до максимально возможного числа (для его размера бит) и продолжает отсчет оттуда.

Поскольку это поведение часто не требуется и может привести к серьезным проблемам безопасности, операторы арифметики Swift + , - и * будут вызывать ошибки, когда операция приведет к переполнению или потоку. Чтобы явно разрешить переполнение и недоиспользование, вместо этого используйте &+ , &- и &* .

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Приоритет стандартных операторов Swift

В первую очередь перечисляются операторы, которые связаны более строгим (более высокий приоритет).

операторы	Группа приоритетов (≥ 3,0)	старшинство	Ассоциативность
.		∞	оставил
? , ! , ++ , -- , [] , () , {}	(Постфиксный)		
! , ~ , + , - , ++ , --	(префикс)		
~> (быстрый ≤2.3)		255	оставил

операторы	Группа приоритетов (≥ 3,0)	старшинство	Ассоциативность
`<<` , `>>`	BitwiseShiftPrecedence	160	никто
`*` , `/` , `%` , `&` , `&*`	MultiplicationPrecedence	150	оставил
`+` , `-` , `|` , `^` , `&+` , `&-`	AdditionPrecedence	140	оставил
`...` , `..<`	RangeFormationPrecedence	135	никто
`is` , `as` , `as?` , `as!`	CastingPrecedence	132	оставил
`??`	NilCoalescingPrecedence	131	право
`<` , `<=` , `>` , `>=` , `==` `!=` , `===` `!==` , `~=`	ComparisonPrecedence	130	никто
`&&`	LogicalConjunctionPrecedence	120	оставил
`||`	LogicalDisjunctionPrecedence	110	оставил
	DefaultPrecedence *		никто
`?` ... `:`	TernaryPrecedence	100	право
`=` , `+=` , `-=` , `*=` , `/=` , `%=` , `<<=` , `>>=` , `&=` , `|=` , `^=`	AssignmentPrecedence	90	право, назначение
`->`	FunctionArrowPrecedence		право

3.0

- Группа приоритетов DefaultPrecedence выше, чем TernaryPrecedence , но не упорядочена с остальными операторами. Помимо этой группы, остальные приоритеты линейны.

- Эту таблицу можно также найти в справочной системе Apple API
- Фактическое определение групп приоритетов можно найти в исходном коде GitHub

Прочитайте Расширенные операторы онлайн: https://riptutorial.com/ru/swift/topic/1048/расширенные-операторы

глава 43: Связанные объекты

Examples

Свойство в расширении протокола достигается с помощью связанного объекта.

В Swift расширения протокола не могут иметь истинных свойств.

Однако на практике вы можете использовать технику «связанного объекта». Результат почти точно как «реальное» свойство.

Вот точная методика добавления «связанного объекта» к расширению протокола:

По сути, вы используете объектно-с-вызовы objc_getAssociatedObject и _set.

Основные вызовы:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Вот полный пример. Две критические точки:

1. В протоколе вы должны использовать «: class», чтобы избежать проблемы с мутацией.
2. В расширении вы должны использовать «where Self: UIViewController» (или любой подходящий класс), чтобы указать подтверждающий тип.

Итак, для примера property "p":

```
import Foundation
import UIKit
import ObjectiveC          // don't forget this

var _Handle: UInt8 = 42    // it can be any value

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // note that you >> do not << declare p here
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType might be, say, an Enum
      get {
          return objc_getAssociatedObject(self, & _Handle) as! YourType
          // HOWEVER, SEE BELOW
          }
      set {
          objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
          // often, you'll want to run some sort of "setter" here...
          __setter()
          }
    }

    func __setter() { something = p.blah() }
```

```
    func someOtherExtensionFunction() { p.blah() }
    // it's ok to use "p" inside other extension functions,
    // and you can use p anywhere in the conforming class
}
```

В любом соответствующем классе вы теперь «добавили» свойство «p»:

Вы можете использовать «p» так же, как вы бы использовали любое обычное свойство в соответствующем классе. Пример:

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
      u = ...
      ... = u
      // use "p" as you would any normal property
      p = ...
      ... = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
      pm = .none // "p" MUST be "initialized" somewhere in Clock
    }
}
```

Заметка. Вы ДОЛЖНЫ инициализировать псевдо-свойство.

Xcode ***не*** **будет принудительно выполнять** инициализацию «p» в соответствующем классе.

Очень важно, чтобы вы инициализировали «p», возможно, в viewDidLoad подтверждающего класса.

Стоит помнить, что p на самом деле просто ***вычисленное свойство*** . p на самом деле просто две функции с синтаксическим сахаром. В любом месте нет «переменной»: компилятор не «назначает некоторую память для p» в каком-либо смысле. По этой причине бессмысленно ожидать, что Xcode обеспечит «инициализацию p».

Действительно, чтобы говорить более точно, вы должны помнить, что «используйте p в первый раз, как если бы вы его инициализировали». (Опять же, это, скорее всего, будет в вашем коде viewDidLoad.)

Что касается геттера как такового.

Обратите внимание, что он **будет сбой**, если вызывается геттер до того, как будет установлено значение для «p».

Чтобы избежать этого, рассмотрите код, например:

```
    get {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
            objc_setAssociatedObject(self, &_Handle, _default initial value_,
.OBJC_ASSOCIATION)
            return _default initial value_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

Повторить. Xcode ***не*** **будет принудительно выполнять** инициализацию p в соответствующем классе. Очень важно, чтобы вы инициализировали p, например, в viewDidLoad соответствующего класса.

Сделать код проще ...

Вы можете использовать эти две глобальные функции:

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
        objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

Обратите внимание, что они ничего не делают, кроме сохранения ввода и делают код более удобочитаемым. (Они по сути являются макросами или встроенными функциями.)

Тогда ваш код будет выглядеть следующим образом:

```
protocol PMable: class {
    var click:UILabel? { get set } // ordinary properties here
}

var _pHandle: UInt8 = 321

extension PMable where Self:UIViewController {

    var p:P {
        get {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
            _aoSet(self, &_pHandle, newValue)
            __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
        p.blah()
    }
}
```

(В примере в _aoGet, P является initalizable: вместо P () вы можете использовать «», 0 или любое значение по умолчанию.)

Прочитайте Связанные объекты онлайн: https://riptutorial.com/ru/swift/topic/1085/связанные-объекты

глава 44: Словари

замечания

Некоторые примеры в этом разделе могут иметь другой порядок при использовании, потому что порядок словаря не гарантируется.

Examples

Объявление словарей

Словари - это неупорядоченный набор ключей и значений. Значения относятся к уникальным ключам и должны быть одного типа.

При инициализации словаря полный синтаксис выглядит следующим образом:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Хотя более сжатый способ инициализации:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
```

Объявите словарь с ключами и значениями, указав их в списке, разделенном запятыми. Типы могут быть выведены из типов ключей и значений.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]
var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

Изменение словарей

Добавить ключ и значение в словарь

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue возвращает исходное значение, если оно существует, или nil.

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

Удалить значение и их ключи с похожим синтаксисом

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
```

```
//books = [:]
//removedValue = "Deleting from Dictionaries"
```

Доступ к значениям

Значение в Dictionary можно получить с помощью его ключа:

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

Значения словаря можно повторить с помощью свойства values :

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

Аналогично, ключи словаря могут быть повторены с использованием его свойств keys :

```
for bookNumbers in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
// Book number: 1
// Book number: 2
```

Чтобы получить все пары key и value соответствующие друг другу (вы не будете в порядке, так как это словарь)

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
// outputs:
// 2  Book 2
// 1  Book 1
```

Обратите внимание, что Dictionary , в отличие от Array , по своей сути неупорядочен, то есть во время итерации нет гарантии на порядок.

Если вы хотите получить доступ к нескольким уровням словаря, используйте повторяющийся синтаксис индекса.

```
// Create a multilevel dictionary.
var myDictionary: [String:[Int:String]]! =
["Toys":[1:"Car",2:"Truck"],"Interests":[1:"Science",2:"Math"]]

print(myDictionary["Toys"][2]) // Outputs "Truck"
print(myDictionary["Interests"][1]) // Outputs "Science"
```

Изменение значения словаря с помощью клавиши

```
var dict = ["name": "John", "surname": "Doe"]
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"
print(dict)
```

Получить все ключи в словаре

```
let myAllKeys = ["name" : "Kirit" , "surname" : "Modi"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

Слияние двух словарей

```
extension Dictionary {
    func merge(dict: Dictionary<Key,Value>) -> Dictionary<Key,Value> {
        var mutableCopy = self
        for (key, value) in dict {
            // If both dictionaries have a value for same key, the value of the other
dictionary is used.
            mutableCopy[key] = value
        }
        return mutableCopy
    }
}
```

Прочитайте Словари онлайн: https://riptutorial.com/ru/swift/topic/310/словари

глава 45: совпадение

Синтаксис

- Swift 3.0
- DispatchQueue.main // Получить основную очередь
- DispatchQueue (метка: «my-serial-queue», атрибуты: [.serial, .qosBackground]) // Создайте свою собственную приватную последовательную очередь
- DispatchQueue.global (атрибуты: [.qosDefault]) // Доступ к одной из глобальных параллельных очередей
- DispatchQueue.main.async {...} // Отправлять задачу асинхронно в основной поток
- DispatchQueue.main.sync {...} // Отправлять задачу синхронно с основным потоком
- DispatchQueue.main.asyncAfter (крайний срок: .now () + 3) {...} // Отправлять задачу асинхронно в основной поток, который будет выполняться через x секунд
- Swift <3.0
- dispatch_get_main_queue () // Получить основную очередь, запущенную в основном потоке
- dispatch_get_global_queue (dispatch_queue_priority_t, 0) // Получить глобальную очередь с указанным приоритетом dispatch_queue_priority_t
- dispatch_async (dispatch_queue_t) {() -> Void in ...} // Отправлять задачу асинхронно по указанному dispatch_queue_t
- dispatch_sync (dispatch_queue_t) {() -> Void in ...} // Отправлять задачу синхронно на указанный dispatch_queue_t
- dispatch_after (dispatch_time (DISPATCH_TIME_NOW, Int64 (наносекунды)), dispatch_queue_t, {...}); // Отправляем задачу на указанный dispatch_queue_t после наносекунд

Examples

Получение очереди Grand Central Dispatch (GCD)

Grand Central Dispatch работает над концепцией «Dispatch Queues». Очередь отправки выполняет задачи, которые вы назначаете в том порядке, в котором они передаются. Существует три типа диспетчерских очередей:

- **Последовательные** диспетчерские очереди (например, частные очереди отправки) выполняют одну задачу за раз, по порядку. Они часто используются для синхронизации доступа к ресурсу.
- **Параллельные** диспетчерские очереди (например, глобальные очереди отправки) выполняют одну или несколько задач одновременно.
- **Основная очередь диспетчера** выполняет задачи в основном потоке.

Для доступа к основной очереди:

3.0

```
let mainQueue = DispatchQueue.main
```

3.0

```
let mainQueue = dispatch_get_main_queue()
```

Система предоставляет *параллельные* глобальные очереди отправки (глобальные для вашего приложения) с различными приоритетами. Вы можете получить доступ к этим очередям, используя класс DispatchQueue в Swift 3:

3.0

```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

эквивалентно

```
let globalConcurrentQueue = DispatchQueue.global()
```

3.0

```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

В iOS 8 или более поздних версиях возможные значения качества обслуживания, которые могут быть переданы, являются .userInteractive , .userInitiated , .default , .utility и .background . Они заменяют константы DISPATCH_QUEUE_PRIORITY_ .

Вы также можете создавать свои собственные очереди с различными приоритетами:

3.0

```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```

3.0

```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue",
DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

В Swift 3 очереди, созданные с помощью этого инициализатора, по умолчанию являются серийными, а передача .workItem для частоты автоопределения обеспечивает создание и удаление пула автозапуска для каждого рабочего элемента. Существует также .never , что означает, что вы сами будете управлять собственными пулами автозапуска, или .inherit который наследует настройку из среды. В большинстве случаев вы, вероятно, не будете использовать. .never кроме случаев экстремальной настройки.

Выполнение задач в очереди Grand Central Dispatch (GCD)

3.0

Чтобы запускать задачи в очереди отправки, используйте методы sync , async и after .

Чтобы отправить задачу в очередь асинхронно:

```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //do something

    DispatchQueue.main.async {
        //this will be called in main thread
        //any UI updates should be placed here
    }
}
```

```
// ... code here will execute immediately, before the task finished
```

Чтобы отправить задачу в очередь синхронно:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

Чтобы отправить задачу в очередь через определенное количество секунд:

```
queue.asyncAfter(deadline: .now() + 3) {
    //this will be executed in a background-thread after 3 seconds
}
// ... code here will execute immediately, before the task finished
```

> **ПРИМЕЧАНИЕ.** Любые обновления пользовательского интерфейса должны быть вызваны в основной поток! Убедитесь, что вы разместили код для обновлений пользовательского интерфейса внутри DispatchQueue.main.async { ... }

2,0

Типы очереди:

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

Чтобы отправить задачу в очередь асинхронно:

```
dispatch_async(queue) {
    // Your code run run asynchronously. Code is queued and executed
    // at some point in the future.
}
// Code after the async block will execute immediately
```

Чтобы отправить задачу в очередь синхронно:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

Чтобы отправить задание через промежуток времени (используйте NSEC_PER_SEC для преобразования секунд в наносекунды):

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
dispatch_get_main_queue()) {
    // Code to be performed in 2.5 seconds here
}
```

Чтобы выполнить задачу асинхронно и обновить пользовательский интерфейс:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
```

```
    }
}
```

> **ПРИМЕЧАНИЕ.** Любые обновления пользовательского интерфейса должны быть вызваны в основной поток! Убедитесь, что вы разместили код для обновлений пользовательского интерфейса внутри dispatch_async(dispatch_get_main_queue()) { ... }

Контурные петли

GCD обеспечивает механизм для выполнения цикла, посредством чего петли происходят одновременно друг с другом. Это очень полезно при выполнении серии дорогостоящих вычислений.

Рассмотрим этот цикл:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

Вы можете выполнять эти вычисления одновременно с помощью concurrentPerform (в Swift 3) или dispatch_apply (в Swift 2):

3.0

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // Do something computationally expensive here
}
```

3.0

```
dispatch_apply(iterations, queue) { index in
    // Do something computationally expensive here
}
```

Закрытие цикла будет вызываться для каждого index от 0 до, но не включая, iterations . Эти итерации будут выполняться одновременно друг с другом, и, следовательно, порядок, в котором они выполняются, не гарантируется. Фактическое количество итераций, которые происходят одновременно в любой момент времени, обычно определяется возможностями соответствующего устройства (например, сколько ядер имеет устройство).

Несколько особых соображений:

- Параметр concurrentPerform / dispatch_apply может запускать циклы одновременно друг относительно друга, но все это происходит синхронно по отношению к потоку, из которого вы его вызываете. Поэтому не называйте это из основного потока, так как это блокирует этот поток до тех пор, пока цикл не будет выполнен.
- Поскольку эти петли происходят одновременно друг с другом, вы несете ответственность за обеспечение безопасности потоков результатов. Например, если обновить какой-либо словарь с результатами этих дорогостоящих вычислений, убедитесь, что вы сами синхронизировали эти обновления.
- Обратите внимание: в запуске параллельных циклов есть некоторые служебные данные. Таким образом, если вычисления, выполняемые внутри цикла, недостаточно интенсивно вычислительны, вы можете обнаружить, что любая производительность, получаемая при использовании параллельных циклов, может быть уменьшена, если не быть полностью компенсированной, служебными данными, связанными с синхронизацией всех этих параллельных потоков.

  Таким образом, вы отвечаете за определение правильного количества работы, которое должно выполняться на каждой итерации цикла. Если вычисления слишком просты, вы можете использовать «шагание», чтобы включить больше работы за цикл. Например, вместо выполнения

параллельного цикла с 1 миллионом тривиальных вычислений вы можете выполнить 100 итераций в своем цикле, выполняя 10 000 вычислений на цикл. Таким образом, в каждом потоке выполняется достаточная работа, поэтому накладные расходы, связанные с управлением этими параллельными циклами, становятся менее значительными.

Выполнение задач в OperationQueue

Вы можете думать о OperationQueue как о линии задач, ожидающих исполнения. В отличие от диспетчерских очередей в GCD, очереди операций не являются FIFO (first-in-first-out). Вместо этого они выполняют задачи, как только они готовы к выполнению, если для этого достаточно системных ресурсов.

Получить главный OperationQueue :

3.0

```
let mainQueue = OperationQueue.main
```

Создайте пользовательский режим OperationQueue :

3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

Качество обслуживания определяет важность работы или насколько пользователь, вероятно, рассчитывает на немедленные результаты этой задачи.

Добавить Operation в OperationQueue :

3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Добавьте блок к OperationQueue :

3.0

```
myQueue.addOperation {
    // some task
}
```

Добавьте несколько Operation s в OperationQueue :

3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Отрегулируйте, сколько Operation s может выполняться одновременно в очереди:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

Приостановка очереди предотвратит запуск каких-либо существующих, нераспределенных операций или любых новых операций, добавленных в очередь. Способ возобновления этой очереди заключается в том, isSuspended вернуть isSuspended в значение false :

3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

Приостановка OperationQueue не останавливает или отменяет выполняемые операции. Нужно только попытаться приостановить создавшуюся очередь, а не глобальные очереди или основную очередь.

Создание высокоуровневых операций

Структура Foundation предоставляет тип Operation , который представляет объект высокого уровня, который инкапсулирует часть работы, которая может выполняться в очереди. Координация не только координирует работу этих операций, но также позволяет устанавливать зависимости между операциями, создавать отмененные операции, ограничивать степень параллелизма, применяемую в очереди операций, и т. Д.

Operation готова к выполнению, когда все ее зависимости завершены. Свойство isReady затем изменяется на true .

Создайте простой неавтоматический подкласс Operation :

3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

2,3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

Добавьте операцию в OperationQueue :

1,0

```
myQueue.addOperation(operation)
```

Это будет выполнять операцию одновременно в очереди.

Управление зависимостями от Operation .

Зависимости определяют другие Operation s, которые должны выполняться в очереди до того, как Operation считается готовой к выполнению.

1,0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Запуск Operation без очереди:

1,0

```
   operation.start()
```

Зависимости будут проигнорированы. Если это параллельная операция, задача может выполняться одновременно, если ее метод start работает в фоновых очередях.

Параллельные операции.

Если задача, которую выполняет Operation , сама, асинхронная (например, URLSession данных URLSession), вы должны реализовать Operation как параллельную операцию. В этом случае ваша isAsynchronous реализация должна возвращать значение true , обычно у вас есть метод start который выполняет некоторую настройку, а затем вызывает его main метод, который фактически выполняет задачу.

При isExecuting асинхронной Operation вы должны реализовать isExecuting , isFinished methods и isFinished . Таким образом, при запуске isExecuting свойство isExecuting изменяется на true . Когда Operation завершает свою задачу, isExecuting устанавливается в false , а isFinished - true . Если операция отменена, оба isCancelled и isFinished изменяются на true . Все эти свойства являются наблюдаемыми по ключевым значениям.

Отмените Operation .

Вызов cancel просто изменяет свойство isCancelled на true . Чтобы отреагировать на отмену в пределах вашего собственного подкласса Operation , вы должны проверить значение isCancelled по крайней мере, периодически в пределах main и соответствующим образом реагировать.

1,0

```
operation.cancel()
```

Прочитайте совпадение онлайн: https://riptutorial.com/ru/swift/topic/1649/совпадение

глава 46: Соглашения стилей

замечания

У Свифта есть официальный стиль руководства: Swift.org API Design Guidelines . Еще один популярный справочник - официальный путеводитель Swift Style от Raywenderlich.com.

Examples

Очистить использование

Избегайте неоднозначности

Название классов, структур, функций и переменных должно избегать двусмысленности.

Пример:

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // implementation
    }
}
```

Вызов функции этой функции будет выглядеть следующим образом:

```
list.remove(at: 42)
```

Таким образом, избегается двусмысленность. Если вызов функции был бы просто list.remove(42) было бы неясно, будет ли удален элемент, равный 42, или если элемент в индексе 42 будет удален.

Избегайте избыточности

Название функций не должно содержать избыточной информации.

Плохой пример:

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // implementation
    }
}
```

Вызов функции может выглядеть как list.removeElement(someObject) . Переменная someObject уже указывает, что элемент удален. Было бы лучше, чтобы подпись функции выглядела так:

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // implementation
    }
}
```

Вызов этой функции выглядит так: list.remove(someObject) .

Именование переменных в соответствии с их ролью

Переменные должны быть названы их ролью (например, поставщиком, приветствием) вместо их типа

(например, фабрика, строка и т. Д.).

Высокая связь между именем протокола и именами переменных

Если имя типа описывает его роль в большинстве случаев (например, Iterator), тип должен быть назван суффиксом `Type`. (например, IteratorType)

Предоставьте дополнительные сведения при использовании слабо типизированных параметров

Если тип объекта явно не указывает на его использование в вызове функции, функция должна быть названа с предшествующим существительным для каждого слабо типизированного параметра, описывающего его использование.
Пример:

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

к которому вызов будет выглядеть как `object.addObserver (self, forKeyPath: path)

вместо

```
func add(_ observer: NSObject, for keyPath: String)
```

к которому вызов будет выглядеть как object.add(self, for: path)

Свободное использование

Использование естественного языка

Вызов функций должен быть близок к естественному английскому языку.

Пример:

```
list.insert(element, at: index)
```

вместо

```
list.insert(element, position: index)
```

Именование методов фабрики

Фабричные методы должны начинаться с префикса `make`.

Пример:

```
factory.makeObject()
```

Именование параметров в инициализаторах и заводских методах

Имя первого аргумента не должно быть связано с присвоением имени фабричному методу или инициализатору.

Пример:

```
factory.makeObject(key: value)
```

Вместо:

```
factory.makeObject(havingProperty: value)
```

Именование по побочным эффектам

- Функции с побочными эффектами (mutating functions) следует называть с помощью глаголов или существительных с префиксом form- .
- Функции без побочных эффектов (nonmutating functions) должны быть названы с использованием существительных или глаголов с суффикс -ing или -ed .

Пример: Мутирующие функции:

```
print(value)
array.sort()                  // in place sorting
list.add(value)               // mutates list
set.formUnion(anotherSet)     // set is now the union of set and anotherSet
```

Немутативные функции:

```
let sortedArray = array.sorted()     // out of place sorting
let union = set.union(anotherSet)    // union is now the union of set and another set
```

Булевы функции или переменные

Заявления, связанные с булевыми, должны читаться как утверждения.

Пример:

```
set.isEmpty
line.intersects(anotherLine)
```

Протоколы именования

- Протоколы, описывающие, что что-то нужно называть, используют существительные.
- Протоколы, описывающие возможности, должны иметь -able , -ible или -ing как суффикс.

Пример:

```
Collection        // describes that something is a collection
ProgressReporting // describes that something has the capability of reporting progress
Equatable         // describes that something has the capability of being equal to something
```

Типы и свойства

Типы, переменные и свойства должны читаться как существительные.

Пример:

```
let factory = ...
let list = [1, 2, 3, 4]
```

капитализация

Типы и протоколы

Имена типов и протоколов должны начинаться с буквы верхнего регистра.

Пример:

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

Все остальное...

Переменные, константы, функции и случаи перечисления должны начинаться с строчной буквы.

Пример:

```
let greeting = "Hello"
let height = 42.0

enum Color {
    case red
    case green
    case blue
}

func print(_ string: String) {
    ...
}
```

Случай верблюда:

Все имена должны использовать соответствующий верблюд. Верхний верблюжковый футляр для имен типов / протоколов и нижний корпус верблюда для всего остального.

Верхний верблюд:

```
protocol IteratorType { ... }
```

Нижняя часть верблюда:

```
let inputView = ...
```

Сокращения

Сокращения следует избегать, если они не используются (например, URL, ID). Если используется аббревиатура, все буквы должны иметь один и тот же случай.

Пример:

```
let userID: UserID = ...
let urlString: URLString = ...
```

Прочитайте Соглашения стилей онлайн: https://riptutorial.com/ru/swift/topic/3031/соглашения-стилей

глава 47: Создать UIImage из инициалов из строки

Вступление

Это класс, который будет генерировать UIImage инициалов человека. Гарри Поттер создаст образ HP.

Examples

InitialsImageFactory

```
class InitialsImageFactory: NSObject {

class func imageWith(name: String?) -> UIImage? {

let frame = CGRect(x: 0, y: 0, width: 50, height: 50)
let nameLabel = UILabel(frame: frame)
nameLabel.textAlignment = .center
nameLabel.backgroundColor = .lightGray
nameLabel.textColor = .white
nameLabel.font = UIFont.boldSystemFont(ofSize: 20)
var initials = ""

if let initialsArray = name?.components(separatedBy: " ") {

  if let firstWord = initialsArray.first {
    if let firstLetter = firstWord.characters.first {
      initials += String(firstLetter).capitalized
    }

  }
  if initialsArray.count > 1, let lastWord = initialsArray.last {
    if let lastLetter = lastWord.characters.first {
      initials += String(lastLetter).capitalized
    }

  }
} else {
  return nil
}

nameLabel.text = initials
UIGraphicsBeginImageContext(frame.size)
if let currentContext = UIGraphicsGetCurrentContext() {
  nameLabel.layer.render(in: currentContext)
  let nameImage = UIGraphicsGetImageFromCurrentImageContext()
  return nameImage
}
return nil
}

}
```

Прочитайте Создать UIImage из инициалов из строки онлайн:
https://riptutorial.com/ru/swift/topic/10915/создать-uiimage-из-инициалов-из-строки

глава 48: Спектакль

Examples

Производительность распределения

В Swift управление памятью выполняется автоматически для автоматического подсчета ссылок. (См. « Управление памятью»). Распределение - это процесс резервирования места в памяти для объекта, а в Swift понимание производительности такого требует некоторого понимания **кучи** и **стека** . Куча - это место памяти, где размещается большинство объектов, и вы можете думать об этом как о хранилище. С другой стороны, стек представляет собой стек вызовов функций, которые привели к текущему исполнению. (Следовательно, трассировка стека - это своего рода распечатка функций в стеке вызовов).

Выделение и освобождение из стека является очень эффективной операцией, однако в сравнении распределение кучи является дорогостоящим. При проектировании производительности вы должны помнить об этом.

Классы:

```
class MyClass {

    let myProperty: String

}
```

Классы в Swift являются ссылочными типами, и поэтому происходит несколько вещей. Во-первых, фактический объект будет выделен в кучу. Затем любые ссылки на этот объект должны быть добавлены в стек. Это делает классы более дорогостоящим объектом для распределения.

Структуры:

```
struct MyStruct {

    let myProperty: Int

}
```

Поскольку structs являются типами значений и поэтому копируются при передаче, они выделяются в стеке. Это делает структуры более эффективными, чем классы, однако, если вам нужно понятие идентичности и / или эталонная семантика, структура не может предоставить вам эти вещи.

Предупреждение о структурах со строками и свойствами, которые являются классами

Хотя структуры, как правило, более чистые, чем классы, вы должны быть осторожны с структурами со свойствами, которые являются классами:

```
struct MyStruct {

    let myProperty: MyClass

}
```

Здесь, из-за подсчета ссылок и других факторов, производительность теперь больше похожа на класс. Кроме того, если более чем одно свойство в структуре является классом, влияние производительности может быть еще более отрицательным, чем если бы структура была классом.

Кроме того, хотя Strings являются структурами, они внутренне хранят свои символы в куче, поэтому стоят дороже, чем большинство структур.

Прочитайте Спектакль онлайн: https://riptutorial.com/ru/swift/topic/4067/спектакль

Синтаксис

- String.characters // Возвращает массив символов в строке
- String.characters.count // Возвращает количество символов
- String.utf8 // A String.UTF8View возвращает символы символа UTF-8 в строке
- String.utf16 // A String.UTF16View возвращает символы символа UTF-16 в строке
- String.unicodeScalars // String.UnicodeScalarView возвращает символы символа UTF-32 в строке
- String.isEmpty // Возвращает true, если строка не содержит текста
- String.hasPrefix (String) // Возвращает true, если строка имеет префикс с аргументом
- String.hasSuffix (String) // Возвращает true, если String суффикс с аргументом
- String.startIndex // Возвращает индекс, соответствующий первому символу в строке
- String.endIndex // Возвращает индекс, соответствующий пятну после последнего символа в строке
- String.components (separateBy: String) // Возвращает массив, содержащий подстроки, разделенные данной разделительной строкой
- String.append (Character) // Добавляет символ (заданный как аргумент) в String

замечания

String в Swift - это набор символов, а также набор скаляров Unicode. Поскольку Swift Strings основаны на Unicode, они могут быть любым сканируемым значением Unicode, включая языки, отличные от английского, и emojis.

Поскольку два скаляра могут объединяться для формирования одного символа, количество скаляров в String не обязательно всегда совпадает с количеством символов.

Дополнительные сведения о строках см. В разделе Быстрый язык программирования и ссылка на String Structure .

Сведения о реализации см. В разделе «Swift String Design»

Examples

Строковые и символьные литералы

Строковые литералы в Swift ограничены двойными кавычками ("):

```
let greeting = "Hello!"  // greeting's type is String
```

Символы могут быть инициализированы из строковых литералов, пока литерал содержит только один кластер графем:

```
let chr: Character = "H" // valid
let chr2: Character = "" // valid
let chr3: Character = "abc" // invalid - multiple grapheme clusters
```

Интерполяция строк

Строковая интерполяция позволяет вводить выражение непосредственно в строковый литерал. Это можно сделать со всеми типами значений, включая строки, целые числа, числа с плавающей запятой и многое другое.

Синтаксис - это обратная косая черта, за которой следуют скобки, обертывающие значение: \(value) . Любое допустимое выражение может появляться в круглых скобках, включая вызовы функций.

```
let number = 5
let interpolatedNumber = "\(number)"  // string is "5"
let fortyTwo = "\(6 * 7)"             // string is "42"

let example = "This post has \(number) view\(number == 1 ? "" : "s")"
// It will output "This post has 5 views" for the above example.
// If the variable number had the value 1, it would output "This post has 1 view" instead.
```

Для пользовательских типов поведение по умолчанию для интерполяции строк заключается в том, что "\(myobj)" эквивалентно String(myobj) , тому же представлению, используемому print(myobj) . Вы можете настроить это поведение, CustomStringConvertible протокол CustomStringConvertible для своего типа.

3.0

Для Swift 3, в соответствии с SE-0089 , String.init<T>(_:) был переименован в String.init<T>(describing:) .

Интерполяция строк "\(myobj)" предпочтет новый String.init<T: LosslessStringConvertible>(_:) , но вернется к init<T>(describing:) если значение не LosslessStringConvertible .

Специальные символы

Некоторым символам требуется специальная **escape-последовательность**, чтобы использовать их в строковых литералах:

символ	Имея в виду
\0	нулевой символ
\\	простой обратный слэш, \
\t	символ табуляции
\v	вертикальная вкладка
\r	возвращение каретки
\n	строка («новая строка»)
\"	двойная кавычка "
\'	одна цитата, '
\u{n}	кодовая точка Unicode *n* (в шестнадцатеричной форме)

Пример:

```
let message = "Then he said, \"I \u{1F496} you!\""

print(message) // Then he said, "I 💖 you!"
```

Конкатенатные строки

Объединить строки с оператором + для создания новой строки:

```
let name = "John"
let surname = "Appleseed"
let fullName = name + " " + surname  // fullName is "John Appleseed"
```

Добавляем к **изменяемой** строке, используя оператор присваивания += или используя метод:

```
let str2 = "there"
var instruction = "look over"
instruction += " " + str2  // instruction is now "look over there"

var instruction = "look over"
instruction.append(" " + str2)  // instruction is now "look over there"
```

Добавить один символ в изменяемую строку:

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// produces a modified String (greeting) = "Hello!"
```

Добавить несколько символов в изменяемую строку

```
var alphabet:String = "my ABCs: "
alphabet.append(contentsOf: (0x61...0x7A).map(UnicodeScalar.init)
                                         .map(Character.init) )
// produces a modified string (alphabet) = "my ABCs: abcdefghijklmnopqrstuvwxyz"
```

3.0

appendContentsOf(_:) был переименован в append(_:) .

Присоедините последовательность строк, чтобы сформировать новую строку, используя joinWithSeparator(_:) :

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

3.0

joinWithSeparator(_:) был переименован в joinWithSeparator(_:) joined(separator:) .

По умолчанию separator является пустая строка, поэтому ["a", "b", "c"].joined() == "abc" .

Изучить и сравнить строки

Проверьте, нет ли строки:

```
if str.isEmpty {
    // do something if the string is empty
}

// If the string is empty, replace it with a fallback:
let result = str.isEmpty ? "fallback string" : str
```

Проверьте, равны ли две строки (в смысле канонической эквивалентности Unicode):

```
"abc" == "def"          // false
"abc" == "ABC"          // false
"abc" == "abc"          // true

// "LATIN SMALL LETTER A WITH ACUTE" == "LATIN SMALL LETTER A" + "COMBINING ACUTE ACCENT"
```

```
"\u{e1}" == "a\u{301}"  // true
```

Убедитесь, что строка начинается / заканчивается другой строкой:

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

Кодирование строк и их разложение

Строка Swift состоит из кодовых точек Unicode . Его можно разложить и закодировать несколькими способами.

```
let str = "ที่①!"
```

Разделение строк

Строка , в characters являются Unicode расширенной графемы кластеры :

```
Array(str.characters)  // ["ที่", "①", "!"]
```

unicodeScalars - это коды кода Unicode, которые составляют строку (обратите внимание, что ที่ является одним кластером графем, но 3 кодовых пункта - 3607, 3637, 3656), поэтому длина результирующего массива не совпадает с длиной characters):

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

Вы можете кодировать и разлагать строки как UTF-8 (последовательность UInt8 s) или UTF-16 (последовательность UInt16 s):

```
Array(str.utf8)   // [224, 184, 151, 224, 184, 181, 224, 185, 136, 240, 159, 145, 140, 226,
145, 160, 33]
Array(str.utf16)  // [3607, 3637, 3656, 55357, 56396, 9312, 33]
```

Длина строки и итерация

characters строки, unicodeScalars , utf8 и utf16 - это все коллекции s, поэтому вы можете получить их count и перебрать их:

```
// NOTE: These operations are NOT necessarily fast/cheap!

str.characters.count     // 4
str.unicodeScalars.count // 6
str.utf8.count           // 17
str.utf16.count          // 7
```

```
for c in str.characters { // ...
for u in str.unicodeScalars { // ...
for byte in str.utf8 { // ...
for byte in str.utf16 { // ...
```

Unicode

Установка значений

Использование Unicode напрямую

```
var str: String = "I want to visit 🇨🇦, Москва, मुंबई, القاهرة, and 🇨🇳. 🐱"
var character: Character = "🐱"
```

Использование шестнадцатеричных значений

```
var str: String = "\u{61}\u{5927}\u{1F34E}\u{3C0}" // a大🍎π
var character: Character = "\u{65}\u{301}" // é = "e" + accent mark
```

Обратите внимание, что Swift Character может состоять из нескольких кодовых точек Unicode, но представляется одиночным символом. Это называется расширенным кластером графем.

Конверсии

String -> Hex

```
// Accesses views of different Unicode encodings of `str`
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

Hex -> String

```
let value0: UInt8 = 0x61
let value1: UInt16 = 0x5927
let value2: UInt32 = 0x1F34E

let string0 = String(UnicodeScalar(value0)) // a
let string1 = String(UnicodeScalar(value1)) // 大
let string2 = String(UnicodeScalar(value2)) // 🍎

// convert hex array to String
let myHexArray = [0x43, 0x61, 0x74, 0x203C, 0x1F431] // an Int array
var myString = ""
for hexValue in myHexArray {
    myString.append(UnicodeScalar(hexValue))
}
print(myString) // Cat!!🐱
```

Обратите внимание, что для UTF-8 и UTF-16 преобразование не всегда легко, потому что такие вещи, как emoji, не могут быть закодированы с одним значением UTF-16. Требуется суррогатная пара.

Реверсивные струны

2,2

```
let aString = "This is a test string."

// first, reverse the String's characters
let reversedCharacters = aString.characters.reverse()

// then convert back to a String with the String() initializer
let reversedString = String(reversedCharacters)

print(reversedString) // ".gnirts tset a si sihT"
```

3.0

```
let reversedCharacters = aString.characters.reversed()
let reversedString = String(reversedCharacters)
```

Строки верхнего и нижнего регистров

Чтобы сделать все символы в строковом или нижнем регистре:

2,2

```
let text = "AaBbCc"
let uppercase = text.uppercaseString // "AABBCC"
let lowercase = text.lowercaseString // "aabbcc"
```

3.0

```
let text = "AaBbCc"
let uppercase = text.uppercased() // "AABBCC"
let lowercase = text.lowercased() // "aabbcc"
```

Проверьте, содержит ли String символы из заданного набора

Буквы

3.0

```
let letters = CharacterSet.letters

let phrase = "Test case"
let range = phrase.rangeOfCharacter(from: letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
   print("letters not found")
}
```

2,2

```
let letters = NSCharacterSet.letterCharacterSet()

let phrase = "Test case"
let range = phrase.rangeOfCharacterFromSet(letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
  print("letters not found")
}
```

Новая структура CharacterSet которая также соединяется с классом Objective-C NSCharacterSet определяет несколько предопределенных множеств:

- decimalDigits
  - capitalizedLetters

        - alphanumerics
    - controlCharacters
    - illegalCharacters
    - и больше вы можете найти в ссылке NSCharacterSet .

Вы также можете определить свой собственный набор символов:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive) {
   print("yes")
}
else {
   print("no")
}
```

2,2

```
let charset = NSCharacterSet(charactersInString: "t")

if let _ = phrase.rangeOfCharacterFromSet(charset, options: .CaseInsensitiveSearch, range:
nil) {
   print("yes")
}
else {
    print("no")
}
```

Вы также можете включить диапазон:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive, range:
phrase.startIndex..<phrase.endIndex)) {
   print("yes")
}
else {
   print("no")
}
```

Количество вхождений символа в строку

Учитывая String и Character

```
let text = "Hello World"
let char: Character = "o"
```

Мы можем подсчитать количество раз, когда Character появляется в String используя

```
let sensitiveCount = text.characters.filter { $0 == char }.count // case-sensitive
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // case-insensitive
```

Удалить символы из строки, не определенной в Set

2,2

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
   return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text, set: chars) // "SwiftComeOut"
```

3.0

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
  return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text: text, set: chars)
```

Форматирование строк

Ведущие нули

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

Числа после десятичного разряда

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416 (rounded)
```

Десятичное число до шестнадцатеричного

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b (notice the lowercase b)
```

В качестве альтернативы можно использовать специализированный инициализатор, который делает то же самое:

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

Десятичное число с произвольным числом оснований

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

Radix - Int в [2, 36] .

Преобразование строки Swift в числовой тип

```
Int("123") // Returns 123 of Int type
Int("abcd") // Returns nil
Int("10") // Returns 10 of Int type
Int("10", radix: 2) // Returns 2 of Int type
Double("1.5") // Returns 1.5 of Double type
Double("abcd") // Returns nil
```

Обратите внимание, что при выполнении этого возвращается Optional значение, которое должно быть разворачиваться соответствующим образом перед использованием.

Итерация строк

3.0

```
let string = "My fantastic string"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
    index = index.successor()
}
```

Примечание: endIndex после конца строки (т.е. string[string.endIndex] является ошибкой, но string[string.startIndex] в порядке). Кроме того, в пустой строке ("") string.startIndex == string.endIndex true . Не забудьте проверить пустые строки, так как вы не можете вызвать startIndex.successor() в пустой строке.

3.0

В Swift 3 индексы String больше не имеют successor() , predecessor() , advancedBy(_:) , advancedBy(_:limit:) или distanceTo(_:) .

Вместо этого эти операции перемещаются в коллекцию, которая теперь отвечает за увеличение и уменьшение своих индексов.

Доступными методами являются .index(after:) , .index(before:) и .index(_:, offsetBy:) .

```
let string = "My fantastic string"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
    currentIndex = string.index(after: currentIndex)
}
```

Примечание. Мы используем currentIndex в качестве имени переменной, чтобы избежать путаницы с методом .index .

И, например, если вы хотите пойти другим путем:

3.0

```
var index:String.Index? = string.endIndex.predecessor()

while index != nil {
    print(string[index!])
```

```
    if index != string.startIndex {
        index = index.predecessor()
    }
    else {
        index = nil
    }
}
```

(Или вы можете просто перенести строку сначала, но если вам не нужно проходить весь путь через строку, вы, вероятно, предпочли бы такой метод)

3.0

```
var currentIndex: String.Index? = string.index(before: string.endIndex)

while currentIndex != nil {
    print(string[currentIndex!])
    if currentIndex != string.startIndex {
        currentIndex = string.index(before: currentIndex!)
    }
    else {
        currentIndex = nil
    }
}
```

Примечание. Index - это тип объекта, а не Int . Вы не можете получить доступ к символу строки следующим образом:

```
let string = "My string"
string[2] // can't do this
string.characters[2] // and also can't do this
```

Но вы можете получить конкретный индекс следующим образом:

3.0

```
index = string.startIndex.advanceBy(2)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

И может вернуться назад следующим образом:

3.0

```
index = string.endIndex.advancedBy(-2)
```

3.0

```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

Если вы можете превысить границы строки или хотите указать лимит, вы можете использовать:

3.0

```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

В качестве альтернативы можно просто перебирать символы в строке, но это может быть менее полезно в зависимости от контекста:

```
for c in string.characters {
    print(c)
}
```

Удалить ведущие и задние WhiteSpace и NewLine

3.0

```
let someString = "  Swift Language  \n"
let trimmedString =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift Language"
```

Метод stringByTrimmingCharactersInSet возвращает новую строку, выполненную путем удаления с обоих концов символов String, содержащихся в заданном наборе символов.

Мы также можем просто удалить только пробел или новую строку.

Удаление только пробелов:

```
let trimmedWhiteSpace =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())
// "Swift Language  \n"
```

Удаление только новой строки:

```
let trimmedNewLine =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.newlineCharacterSet())
// "  Swift Language  "
```

3.0

```
let someString = "  Swift Language  \n"

let trimmedString = someString.trimmingCharacters(in: .whitespacesAndNewlines)
// "Swift Language"

let trimmedWhiteSpace = someString.trimmingCharacters(in: .whitespaces)
// "Swift Language  \n"

let trimmedNewLine = someString.trimmingCharacters(in: .newlines)
// "  Swift Language  "
```

Примечание: все эти методы принадлежат Foundation . Используйте import Foundation если Foundation еще не импортирован через другие библиотеки, такие как Cocoa или UIKit.

Преобразование строки в и из данных / NSData

Чтобы преобразовать String в и из Data / NSData, нам нужно закодировать эту строку с определенной кодировкой. Самым известным является UTF-8 который представляет собой 8-битное представление символов Unicode, пригодных для передачи или хранения на основе ASCII-систем. Вот список всех доступных String Encodings

String к Data / NSData

3.0

```
let data = string.data(using: .utf8)
```

2,2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

Data / NSData для String

3.0

```
let string = String(data: data, encoding: .utf8)
```

2,2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

Разделение строки в массив

В Swift вы можете легко разделить String на массив, отрезав его определенным символом:

3.0

```
let startDate = "23:51"

let startDateAsArray = startDate.components(separatedBy: ":") // ["23", "51"]`
```

2,2

```
let startDate = "23:51"

let startArray = startDate.componentsSeparatedByString(":") // ["23", "51"]`
```

Или когда разделителя нет:

3.0

```
let myText = "MyText"

let myTextArray = myText.components(separatedBy: " ") // myTextArray is ["MyText"]
```

2,2

```
let myText = "MyText"

let myTextArray = myText.componentsSeparatedByString(" ") // myTextArray is ["MyText"]
```

Прочитайте Строки и символы онлайн: https://riptutorial.com/ru/swift/topic/320/строки-и-символы

глава 50: Структуры

Examples

Основы структур

```
struct Repository {
    let identifier: Int
    let name: String
    var description: String?
}
```

Это определяет структуру Repository с тремя хранимыми свойствами, целочисленным identifier , строковым name и необязательным description строки. identifier и name являются константами, так как они были объявлены с помощью let -keyword. После установки во время инициализации они не могут быть изменены. Описание - это переменная. Изменение этого параметра изменяет значение структуры.

Типы структуры автоматически получают членский инициализатор, если они не определяют никаких собственных собственных инициализаторов. Структура получает член-инициализатор, даже если он сохранил свойства, которые не имеют значений по умолчанию.

Repository содержит три хранимых свойства, для которых только description имеет значение по умолчанию (nil). Далее он не определяет никаких собственных инициализаторов, поэтому он бесплатно получает инициализатор по порядку:

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New
Repository")
```

Структуры являются типами значений

В отличие от классов, которые передаются по ссылке, структуры передаются путем копирования:

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

Строка - это структура, поэтому она копируется при назначении.

Структуры также не могут сравниваться с использованием оператора идентичности:

```
window0 === window1 // works because a window is a class instance
"hello" === "hello" // error: binary operator '===' cannot be applied to two 'String' operands
```

Каждые два экземпляра структуры считаются идентичными, если сравнивать их равными.

В совокупности эти черты, которые различают структуры из классов, являются тем, что делает типы ценностей структур.

Мутирование структуры

Метод структуры, который меняет значение самой структуры, должен иметь префикс ключевого слова mutating

```
struct Counter {
```

```
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

Когда вы можете использовать мутирующие методы

Методы mutating доступны только для значений структуры внутри переменных.

```
var counter = Counter()
counter.next()
```

Если вы не можете использовать мутирующие методы

С другой стороны, mutating методы НЕ доступны по значениям структуры внутри констант

```
let counter = Counter()
counter.next()
//  error: cannot use mutating member on immutable value: 'counter' is a 'let' constant
```

Структуры не могут наследовать

В отличие от классов, структуры не могут наследовать:

```
class MyView: NSView { }  // works

struct MyInt: Int { } // error: inheritance from non-protocol type 'Int'
```

Однако структуры могут принимать протоколы:

```
struct Vector: Hashable { ... }  // works
```

Доступ к членам структуры

В Swift структуры используют простой «точечный синтаксис» для доступа к своим членам.

Например:

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

Вы можете получить доступ (распечатать) диапазон следующим образом:

```
    print(pizzaRange.range) // 200
```

Вы даже можете получить доступ к членам членов, используя точечный синтаксис:

```
 print(pizzaRange.center.latitude) // 44.9871
```

Подобно тому, как вы можете читать значения с синтаксисом точек, вы также можете назначить их.

```
pizzaRange.range = 250
```

Прочитайте Структуры онлайн: https://riptutorial.com/ru/swift/topic/255/структуры

глава 51: Управление памятью

Вступление

В этом разделе описывается, как и когда среда выполнения Swift должна выделять память для структур данных приложений и когда эта память должна быть восстановлена. По умолчанию экземпляры класса поддержки памяти управляются посредством подсчета ссылок. Структуры всегда передаются путем копирования. Чтобы отказаться от схемы управления встроенной памятью, можно использовать структуру [Unmanaged] [1]. [1]: https://developer.apple.com/reference/swift/unmanaged

замечания

Когда использовать слабое ключевое слово:

Следует использовать weak ключевое слово, если ссылочный объект может быть освобожден за время жизни объекта, содержащего ссылку.

Когда использовать ключевое слово unowned:

unowned –ключевое слово следует использовать, если объект ссылки не ожидается освобождаться в течение всего срока службы объекта , держащий ссылку.

Ловушки

Частая ошибка заключается в том, чтобы забыть создавать ссылки на объекты, которые должны жить после завершения функции, например, менеджеров местоположения, менеджеров движения и т. Д.

Пример:

```
class A : CLLocationManagerDelegate
{
    init()
    {
        let locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

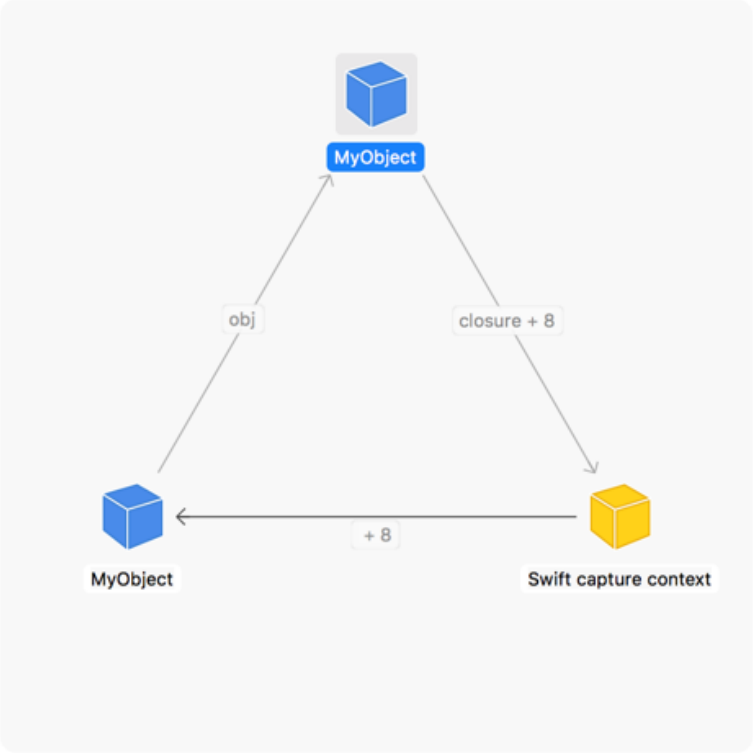
Этот пример не будет работать должным образом, так как менеджер местоположений освобождается после возвращения инициализатора. Правильное решение – создать сильную ссылку в качестве переменной экземпляра:

```
class A : CLLocationManagerDelegate
{
    let locationManager:CLLocationManager

    init()
    {
        locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

Examples

Справочные циклы и слабые ссылки

Базовый цикл (или *цикл сохранения*) назван так потому, что он указывает цикл в графе объектов :



Каждая стрелка указывает, что один объект сохраняет другую (сильная ссылка). Если цикл не сломан, память для этих объектов **никогда не будет освобождена** .

Цикл сохранения создается, когда два экземпляра классов ссылаются друг на друга:

```
class A { var b: B? = nil }
class B { var a: A? = nil }

let a = A()
let b = B()

a.b = b  // a retains b
b.a = a  // b retains a -- a reference cycle
```

Оба экземпляра они будут жить до тех пор, пока программа не завершится. Это цикл сохранения.

Слабые ссылки

Чтобы избежать циклов сохранения, используйте ключевое слово weak или unowned при создании ссылок для прерывания циклов сохранения.

```
class B { weak var a: A? = nil }
```

Слабые или неопубликованные ссылки не будут увеличивать количество ссылок экземпляра. Эти ссылки не способствуют сохранению циклов. Слабая ссылка **становится nil** когда ссылающийся на нее объект освобождается.

```
a.b = b  // a retains b
b.a = a  // b holds a weak reference to a -- not a reference cycle
```

При работе с затворами вы также можете использовать weak и unowned списки захвата .

Управление памятью вручную

При взаимодействии с API-интерфейсом C можно отменить контрольный счетчик Swift. Это достигается с помощью неуправляемых объектов.

Если вам нужно указать указатель типа на функцию C, используйте метод toOpaque Unmanaged структуры для получения необработанного указателя и fromOpaque для восстановления исходного экземпляра:

```
setupDisplayLink() {
  let pointerToSelf: UnsafeRawPointer = Unmanaged.passUnretained(self).toOpaque()
  CVDisplayLinkSetOutputCallback(self.displayLink, self.redraw, pointerToSelf)
}

func redraw(pointerToSelf: UnsafeRawPointer, /* args omitted */) {
  let recoveredSelf = Unmanaged<Self>.fromOpaque(pointerToSelf).takeUnretainedValue()
  recoveredSelf.doRedraw()
}
```

Обратите внимание: если вы используете passUnretained и аналоги, необходимо принять все меры предосторожности, как в случае с unowned ссылками.

Чтобы взаимодействовать с устаревшими API Objective-C, можно было бы вручную повлиять на подсчет ссылок на определенный объект. Для этого Unmanaged имеет соответствующие методы retain и release . Тем не менее, более желательно использовать passRetained и takeRetainedValue , которые выполняют сохранение перед возвратом результата:

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

Эти решения всегда должны быть в последнюю очередь, и языковые API-интерфейсы всегда будут предпочтительнее.

Прочитайте Управление памятью онлайн: https://riptutorial.com/ru/swift/topic/745/управление-памятью

глава 52: Утверждение Отсрочки

Examples

Когда использовать заявление о переносе

Оператор defer состоит из блока кода, который будет выполняться при возврате функции и должен использоваться для очистки.

Как Свифт guard заявления поощряют стиль раннего возвращения, много возможных путей для возвращения могут существовать. Оператор defer предоставляет код очистки, который затем не нужно повторять каждый раз.

Он также может сэкономить время при отладке и профилировании, поскольку можно избежать утечек памяти и неиспользуемых открытых ресурсов из-за забытой очистки.

Его можно использовать для освобождения буфера в конце функции:

```
func doSomething() {
    let data = UnsafeMutablePointer<UInt8>(allocatingCapacity: 42)
    // this pointer would not be released when the function returns
    // so we add a defer-statement
    defer {
        data.deallocateCapacity(42)
    }
    // it will be executed when the function returns.

    guard condition else {
        return /* will execute defer-block */
    }

}   // The defer-block will also be executed on the end of the function.
```

Он также может использоваться для закрытия ресурсов в конце функции:

```
func write(data: UnsafePointer<UInt8>, dataLength: Int) throws {
    var stream:NSOutputStream = getOutputStream()
    defer {
        stream.close()
    }

    let written = stream.write(data, maxLength: dataLength)
    guard written >= 0 else {
        throw stream.streamError! /* will execute defer-block */
    }

}    // the defer-block will also be executed on the end of the function
```

Когда НЕ использовать оператор отсрочки

При использовании инструкции defer-оператора убедитесь, что код остается читаемым, и порядок выполнения остается ясным. Например, следующее использование оператора defer-statement делает порядок выполнения и функцию кода трудно понять.

```
postfix func ++ (inout value: Int) -> Int {
    defer { value += 1 } // do NOT do this!
    return value
}
```

Прочитайте Утверждение Отсрочки онлайн: https://riptutorial.com/ru/swift/topic/4932/утверждение-отсрочки

глава 53: функции

Examples

Основное использование

Функции могут быть объявлены без параметров или возвращаемого значения. Единственной необходимой информацией является имя (hello в этом случае).

```
func hello()
{
    print("Hello World")
}
```

Вызовите функцию без параметров, записав ее имя, а затем пустую пару скобок.

```
hello()
//output: "Hello World"
```

Функции с параметрами

Функции могут принимать параметры, чтобы их функциональность могла быть изменена. Параметры задаются как список, разделенный запятыми, с указанием их типов и имен.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Примечание. Синтаксис \(number1) является базовой интерполяцией строк и используется для вставки целого в String.

Функции с параметрами вызываются путем указания функции по имени и подачи входного значения типа, используемого в объявлении функции.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Любое значение типа Int могло быть использовано.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

Когда функция использует несколько параметров, имя первого параметра не требуется для первого, а имеет следующие параметры.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//output: "15 Is the magic number"
```

Используйте внешние имена параметров, чтобы сделать вызовы функций более читабельными.

```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

Установка значения по умолчанию в объявлении функции позволяет вызывать функцию без ввода каких-либо входных значений.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Возвращаемые значения

Функции могут возвращать значения, задавая тип после списка параметров.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Функции также могут возвращать несколько значений с помощью кортежей.

```
func maths(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = maths(5)
//resultTuple = (10, 15)
```

Ошибки

Если вы хотите, чтобы функция , чтобы быть в состоянии бросить ошибки, вам нужно добавить throws ключевого слова после скобок , которые содержат аргументы:

```
func errorThrower()throws -> String {}
```

Когда вы хотите выбросить ошибку, используйте ключевое слово throw :

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // Throwing an error
```

```
    throw Error.error
  }
}
```

Если вы хотите вызвать функцию, которая может вызвать ошибку, вам нужно использовать ключевое слово try в блоке do :

```
do {
  try errorThrower()
}
```

Подробнее о ошибках Swift: Ошибки

методы

Методы экземпляра - это функции, относящиеся к экземплярам типа в Swift (класс , структура , перечисление или протокол). **Методы** типа вызываются по самому типу.

Методы экземпляра

Методы экземпляров определяются с объявлением func внутри определения типа или в расширении .

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

Метод экземпляра increment() вызывается в экземпляре класса Counter :

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Методы типа

Методы типов определяются со static func ключевыми словами static func . (Для классов class func определяет метод типа, который может быть переопределен подклассами.)

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Параметры Inout

Функции могут изменять переданные им параметры, если они отмечены ключевым словом inout . При передаче параметра inout функции, вызывающий должен добавить & к передаваемой переменной.

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
}
```

```
var apples = 30 // Prints "There's 30 apples"
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("There's now \(apples) apples") // Prints "There's 29 apples".
```

Это позволяет применять ссылочную семантику к типам, которые обычно имеют семантику значений.

Синтаксис закрытия трейлинга

Когда последний параметр функции является замыканием

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

функция может быть вызвана с помощью синтаксиса Trailing Closure

```
loadData("123") { result in
    print(result)
}
```

Операторы - это функции

Операторы, такие как + , - , ?? являются своего рода функцией с использованием символов, а не букв. Они вызываются иначе, чем функции:

- Префикс: **-** x
- Infix: x **+** y
- Postfix: x **++**

Вы можете больше узнать об основных операторах и расширенных операторах на языке Swift Programming.

Вариадические параметры

Иногда невозможно указать количество параметров, которые могут понадобиться функции. Рассмотрим функцию sum :

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

Это отлично подходит для нахождения суммы двух чисел, но для нахождения суммы трех мы должны были бы написать другую функцию:

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

и один с четырьмя параметрами понадобится другой, и так далее. Swift позволяет определить функцию с переменным числом параметров, используя последовательность из трех периодов: ... Например,

```
func sum(_ numbers: Int...) -> Int {
```

```
    return numbers.reduce(0, combine: +)
}
```

Обратите внимание, что параметр numbers , который является переменным, объединяется в один Array типа [Int] . Это в общем случае, вариационные параметры типа T... доступны как [T] .

Теперь эту функцию можно вызвать так:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

Параметр Variable в Swift не должен появляться в конце списка параметров, но в каждой сигнатуре функции может быть только один.

Иногда удобно указывать минимальный размер числа параметров. Например, на самом деле нет смысла брать sum значений. Простой способ обеспечить это - это добавить некоторые невариантные требуемые параметры, а затем добавить переменный параметр после. Чтобы убедиться, что sum может быть вызвана хотя бы двумя параметрами, мы можем написать

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // ok
sum(3, 4, 5, 6, 7) // ok
sum(1) // not ok
sum() // not ok
```

Нижние индексы

Классы, структуры и перечисления могут определять индексы, которые являются ярлыками для доступа к элементам-членам коллекции, списка или последовательности.

пример

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
}
```

Использование подзаголовков

```
var week = DaysOfWeek()
//you access an element of an array at index by array[index].
debugPrint(week[1])
debugPrint(week[0])
week[0] = "Sunday"
debugPrint(week[0])
```

Параметры подписок:

Подписчики могут принимать любое количество входных параметров, и эти входные параметры могут быть любого типа. Подписчики могут также возвращать любой тип. В подзаголовках могут использоваться переменные параметры и переменные параметры, но они не могут использовать параметры вывода или задавать значения параметров по умолчанию.

Пример:

```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

использование

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Функции с закрытием

Использование функций, выполняющих и выполняющих закрытие, может быть чрезвычайно полезно для отправки блока кода, который будет выполнен в другом месте. Мы можем начать с того, что позволили нашей функции взять необязательное закрытие, которое (в этом случае) вернет Void .

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

    if let block = block {
        block()
    }
}
```

Теперь, когда наша функция определена, назовем ее и передадим в код:

```
closedFunc() { Void in
    print("Over already")
}
```

Используя **трейлинг-закрытие** с помощью нашего вызова функции, мы можем передать код (в данном случае, print), который будет выполнен в какой-то момент в нашей функции closedFunc() .

Журнал должен печатать:

Только начало

Уже

Более конкретный вариант использования этого может включать выполнение кода между двумя классами:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // Notice how this is executed before the  closure
        print("1")
        // Make sure closure isn't nil
        self.closure = closure?()
    }
}
```

Журнал должен печатать:

1

2

Функции передачи и возврата

Следующая функция возвращает в качестве результата другую функцию, которая впоследствии может быть назначена переменной и называется:

```
func jediTrainer () -> ((String, Int) -> String) {
  func train(name: String, times: Int) -> (String) {
    return "\(name) has been trained in the Force \(times) times"
  }
  return train
}

let train = jediTrainer()
train("Obi Wan", 3)
```

Типы функций

Каждая функция имеет свой собственный тип функции, состоящий из типов параметров и типа возврата самой функции. Например, следующая функция:

```
func sum(x: Int, y: Int) -> (result: Int) { return x + y }
```

имеет тип функции:

```
(Int, Int) -> (Int)
```

Таким образом, типы функций могут использоваться как типы параметров или типы возврата для функций вложенности.

Прочитайте функции онлайн: https://riptutorial.com/ru/swift/topic/432/функции

Examples

Извлечение списка имен из списка Person (s)

Учитывая структуру Person

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

и Массив Person(s)

```
let persons = [
    Person(name: "Walter White", birthYear: 1959),
    Person(name: "Jesse Pinkman", birthYear: 1984),
    Person(name: "Skyler White", birthYear: 1970),
    Person(name: "Saul Goodman", birthYear: nil)
]
```

мы можем получить массив String содержащий свойство name для каждого Person.

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

Пересекая

```
let numbers = [3, 1, 4, 1, 5]
// non-functional
for (index, element) in numbers.enumerate() {
    print(index, element)
}

// functional
numbers.enumerate().map { (index, element) in
    print((index, element))
}
```

выступающий

Применение функции к коллекции / потоку и создание новой коллекции / потока называется **проекцией** .

```
/// Projection
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
```

```
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]


var videoAndTitlePairs = [[String: AnyObject]]()
newReleases.map { e in
    videoAndTitlePairs.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videoAndTitlePairs)
```

фильтрация

Создание потока путем выбора элементов из потока, который передает определенное условие, называется **фильтрацией**

```
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
```

```
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]

var videos1 = [[String: AnyObject]]()
/**
 *  Filtering using map
 */
newReleases.map { e in
    if e["rating"] as! Float == 5.0 {
        videos1.append(["id": e["id"] as! Int, "title": e["title"] as! String])
    }
}

print(videos1)

var videos2 = [[String: AnyObject]]()
/**
 *  Filtering using filter and chaining
 */
newReleases
    .filter{ e in
        e["rating"] as! Float == 5.0
    }
    .map { e in
    videos2.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videos2)
```

Использование фильтра с помощью структур

Часто вы можете фильтровать структуры и другие сложные типы данных. Поиск массива структур для записей, содержащих конкретное значение, является очень общей задачей и легко достигается в Swift с использованием функций функционального программирования. Более того, код чрезвычайно краток.

```
struct Painter {
    enum Type { case Impressionist, Expressionist, Surrealist, Abstract, Pop }
    var firstName: String
    var lastName: String
    var type: Type
}

let painters = [
    Painter(firstName: "Claude", lastName: "Monet", type: .Impressionist),
    Painter(firstName: "Edgar", lastName: "Degas", type: .Impressionist),
    Painter(firstName: "Egon", lastName: "Schiele", type: .Expressionist),
    Painter(firstName: "George", lastName: "Grosz", type: .Expressionist),
    Painter(firstName: "Mark", lastName: "Rothko", type: .Abstract),
```

```
    Painter(firstName: "Jackson", lastName: "Pollock", type: .Abstract),
    Painter(firstName: "Pablo", lastName: "Picasso", type: .Surrealist),
    Painter(firstName: "Andy", lastName: "Warhol", type: .Pop)
]

// list the expressionists
dump(painters.filter({$0.type == .Expressionist}))

// count the expressionists
dump(painters.filter({$0.type == .Expressionist}).count)
// prints "2"

// combine filter and map for more complex operations, for example listing all
// non-impressionist and non-expressionists by surname
dump(painters.filter({$0.type != .Impressionist && $0.type != .Expressionist})
    .map({$0.lastName}).joinWithSeparator(", "))
// prints "Rothko, Pollock, Picasso, Warhol"
```

Прочитайте Функциональное программирование в Swift онлайн:
https://riptutorial.com/ru/swift/topic/2948/функциональное-программирование-в-swift

глава 55: Функционируйте как граждане первого класса в Свифте

Вступление

Функции как первоклассные члены означают, что они могут пользоваться привилегиями, как это делают объекты. Он может быть назначен переменной, переданной функции в качестве параметра или может использоваться как возвращаемый тип.

Examples

Назначение функции переменной

```
struct Mathematics
{
    internal func performOperation(inputArray: [Int], operation: (Int)-> Int)-> [Int]
    {
        var processedArray = [Int]()

        for item in inputArray
        {
            processedArray.append(operation(item))
        }

        return processedArray
    }


    internal func performComplexOperation(valueOne: Int)-> ((Int)-> Int)
    {
        return
            ({
                 return valueOne + $0
            })
    }

}


let arrayToBeProcessed = [1,3,5,7,9,11,8,6,4,2,100]

let math = Mathematics()

func add2(item: Int)-> Int
{
    return (item + 2)
}

// assigning the function to a variable and then passing it to a function as param
let add2ToMe = add2
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2ToMe))
```

Выход:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Аналогичным образом вышеуказанное может быть достигнуто с использованием `closure`

```
// assigning the closure to a variable and then passing it to a function as param
```

```
let add2 = {(item: Int)-> Int in return item + 2}
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2))
```

Передача функции в качестве аргумента другой функции, тем самым создавая функцию более высокого порядка

```
func multiply2(item: Int)-> Int
{
    return (item + 2)
}


let multiply2ToMe = multiply2

// passing the function directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: multiply2ToMe))
```

Выход:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Аналогичным образом вышеуказанное может быть достигнуто с использованием closure

```
// passing the closure directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

Функция типа возврата из другой функции

```
// function as return type
print(math.performComplexOperation(valueOne: 4)(5))
```

Выход:

```
9
```

Прочитайте Функционируйте как граждане первого класса в Свифте онлайн:
https://riptutorial.com/ru/swift/topic/8618/функционируйте-как-граждане-первого-класса-в-свифте

глава 56: чисел

Examples

Числовые типы и литералы

Встроенные числовые типы Swift:

- Word-размер (зависит от архитектуры), подписанный `Int` и unsigned `UInt` .
- Фиксированного размера , подписанные целые `INT8` , `Int16` , `Int32` , `Int64` и беззнаковых целых чисел `Uint8` , `UInt16` , `UInt32` , `UInt64` .
- Плавающие точки `Float32` / `Float` , `Float64` / `Double` и `Float80` (только для x86).

литералы

Тип числового литерала выводится из контекста:

```
let x = 42    // x is Int by default
let y = 42.0  // y is Double by default

let z: UInt = 42     // z is UInt
let w: Float = -1    // w is Float
let q = 100 as Int8  // q is Int8
```

Подчеркивания (_) могут использоваться для разделения цифр в числовых литералах. Ведущие нули игнорируются.

Литералы с плавающей запятой могут быть указаны с использованием значимых и экспоненциальных частей (*«significand»* **e** *«exponent»* для десятичного числа, **0x** *«significand»* **p** *«exponent»* для шестнадцатеричных чисел).

Целочисленный литеральный синтаксис

```
let decimal = 10               // ten
let decimal = -1000            // negative one thousand
let decimal = -1_000           // equivalent to -1000
let decimal = 42_42_42         // equivalent to 424242
let decimal = 0755             // equivalent to 755, NOT 493 as in some other languages
let decimal = 0123456789

let hexadecimal = 0x10         // equivalent to 16
let hexadecimal = 0x7FFFFFFF
let hexadecimal = 0xBadFace
let hexadecimal = 0x0123_4567_89ab_cdef

let octal = 0o10               // equivalent to 8
let octal = 0o755              // equivalent to 493
let octal = -0o0123_4567

let binary = -0b101010         // equivalent to -42
let binary = 0b111_101_101     // equivalent to 0o755
let binary = 0b1011_1010_1101  // equivalent to 0xB_A_D
```

Синтаксис буквенного обозначения с плавающей запятой

```
let decimal = 0.0
let decimal = -42.0123456789
let decimal = 1_000.234_567_89
```

```
let decimal = 4.567e5               // equivalent to 4.567×10⁵, or 456_700.0
let decimal = -2E-4                 // equivalent to -2×10⁻⁴, or -0.0002
let decimal = 1e+0                  // equivalent to 1×10⁰, or 1.0

let hexadecimal = 0x1p0             // equivalent to 1×2⁰, or 1.0
let hexadecimal = 0x1p-2            // equivalent to 1×2⁻², or 0.25
let hexadecimal = 0xFEEDp+3         // equivalent to 65261×2³, or 522088.0
let hexadecimal = 0x1234.5P4        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x123.45P8        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x12.345P12       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x1.2345P16       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x0.12345P20      // equivalent to 0x12345, or 74565.0
```

Преобразование одного числового типа в другой

```
func doSomething1(value: Double) { /* ... */ }
func doSomething2(value: UInt) { /* ... */ }

let x = 42               // x is an Int
doSomething1(Double(x))  // convert x to a Double
doSomething2(UInt(x))    // convert x to a UInt
```

Инициализаторы целых чисел создают **ошибку времени выполнения**, если значение переполняется или переполняется:

```
Int8(-129.0) // fatal error: floating point value cannot be converted to Int8 because it is
less than Int8.min
Int8(-129)   // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(-128)   // ok
Int8(-2)     // ok
Int8(17)     // ok
Int8(127)    // ok
Int8(128)    // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(128.0)  // fatal error: floating point value cannot be converted to Int8 because it is
greater than Int8.max
```

Поля чисел округления до целого числа **равны нулю** :

```
Int(-2.2)  // -2
Int(-1.9)  // -1
Int(-0.1)  //  0
Int(1.0)   //  1
Int(1.2)   //  1
Int(1.9)   //  1
Int(2.0)   //  2
```

Преобразование Integer-to-float может быть **потерянным** :

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

Преобразование чисел в / из строк

Используйте инициализаторы строк для преобразования чисел в строки:

```
String(1635999)                             // returns "1635999"
String(1635999, radix: 10)                  // returns "1635999"
```

```
String(1635999, radix: 2)                    // returns "110001111011010011111"
String(1635999, radix: 16)                   // returns "18f69f"
String(1635999, radix: 16, uppercase: true)  // returns "18F69F"
String(1635999, radix: 17)                   // returns "129gf4"
String(1635999, radix: 36)                   // returns "z2cf"
```

Или используйте строчную интерполяцию для простых случаев:

```
let x = 42, y = 9001
"Between \(x) and \(y)"  // equivalent to "Between 42 and 9001"
```

Используйте инициализаторы числовых типов для преобразования строк в числа:

```
if let num = Int("42") { /* ... */ }                // num is 42
if let num = Int("Z2cF") { /* ... */ }              // returns nil (not a number)
if let num = Int("z2cf", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int("Z2cF", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int8("Z2cF", radix: 36) { /* ... */ }  // returns nil (too large for Int8)
```

округление

круглый

Раундов значение до ближайшего целого числа с округлением x.5 (но обратите внимание, что -x.5 раундов вниз).

```
round(3.000) // 3
round(3.001) // 3
round(3.499) // 3
round(3.500) // 4
round(3.999) // 4

round(-3.000) // -3
round(-3.001) // -3
round(-3.499) // -3
round(-3.500) // -4  *** careful here ***
round(-3.999) // -4
```

перекрывать

Раунды любого числа с десятичным значением до следующего большего целого числа.

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4

ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

этаж

Раунды любого числа с десятичным значением до следующего меньшего целого числа.

```
floor(3.000) // 3
floor(3.001) // 3
```

```
floor(3.999) // 3

floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

Int

Преобразует Double в Int , отбрасывая любое десятичное значение.

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

Заметки

- round , ceil и floor рукояткой 64 и 32-битной архитектуры.

Генерация случайных чисел

arc4random_uniform(someNumber: UInt32) -> UInt32

Это дает случайные целые числа в диапазоне от 0 до someNumber - 1 .

Максимальное значение для UInt32 составляет 4 294 967 295 (т. UInt32 2^32 - 1).

Примеры:

- Монетный флип

  ```
  let flip = arc4random_uniform(2) // 0 or 1
  ```

- Бросок в кости

  ```
  let roll = arc4random_uniform(6) + 1 // 1...6
  ```

- Случайный день в октябре

  ```
  let day = arc4random_uniform(31) + 1 // 1...31
  ```

- Случайный год в 1990-х

  ```
  let year = 1990 + arc4random_uniform(10)
  ```

Общая форма:

```
let number = min + arc4random_uniform(max - min + 1)
```

где number , max и min - UInt32 .

Заметки

- Существует небольшое смещение по модулю с arc4random так что arc4random_uniform является

предпочтительным.
- Вы можете UInt32 значение UInt32 Int но просто остерегайтесь выхода из диапазона.

Возведение

В Swift мы можем **усилить** Double s со встроенным методом pow() :

```
pow(BASE, EXPONENT)
```

В приведенном ниже коде база (5) установлена на мощность экспоненты (2):

```
let number = pow(5.0, 2.0) // Equals 25
```

Прочитайте чисел онлайн: https://riptutorial.com/ru/swift/topic/454/чисел

глава 57: Чтение и письмо JSON

Синтаксис

- NSJSONSerialization.JSONObjectWithData (jsonData, options: NSJSONReadingOptions) // Возвращает объект из jsonData. Этот метод бросает неудачу.
- NSJSONSerialization.dataWithJSONObject (jsonObject, options: NSJSONWritingOptions) // Возвращает NSData из объекта JSON. Перейдите в NSJSONWritingOptions.PrettyPrinted в опции для вывода, который более читабельен.

Examples

Сериализация JSON, кодирование и декодирование с помощью Apple Foundation и стандартной библиотеки Swift

Класс JSONSerialization встроен в структуру Apple Foundation.

2,2

Читать JSON

Функция JSONObjectWithData принимает NSData и возвращает AnyObject . Вы можете использовать as? для преобразования результата в ожидаемый тип.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else
{
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}
```

Вы можете передавать options: .AllowFragments вместо options: [] чтобы разрешить чтение JSON, когда объект верхнего уровня не является массивом или словарем.

Написать JSON

Вызов dataWithJSONObject преобразует JSON-совместимый объект (вложенные массивы или словари со строками, номерами и NSNull) в необработанные NSData закодированные как UTF-8.

```
do {
    // Convert object to JSON as NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

Вы можете передать options: .PrettyPrinted вместо options: [] для довольно-печатной.

3.0

Такое же поведение в Swift 3, но с другим синтаксисом.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}

do {
    // Convert object to JSON as NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

> Примечание. В настоящее время следующие доступны только в **Swift 4.0** и более поздних версиях.

Начиная с Swift 4.0, стандартная библиотека Swift включает в себя протоколы Encodable и Decodable для определения стандартизованного подхода к кодированию и декодированию данных. Принятие этих протоколов позволит реализации протоколов Encoder и Decoder принимать ваши данные и кодировать или декодировать их в и из внешнего представления, такого как JSON. Соответствие с Codable протоколом сочетает в Encodable и Decodable протоколы. Это теперь рекомендуемое средство для обработки JSON в вашей программе.

Автоматическое кодирование и декодирование

Самый простой способ сделать тип codable - объявить его свойства как типы, которые уже являются Codable . Эти типы включают стандартные типы библиотек, такие как String , Int и Double ; и типы Foundation, такие как Date , Data и URL . Если свойства типа можно кодировать, сам тип автоматически будет соответствовать Codable простым объявлением соответствия.

Рассмотрим следующий пример, в котором структура Book соответствует Codable .

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Обратите внимание, что стандартные коллекции, такие как Array и Dictionary соответствуют Codable если они содержат типы codable.

Приняв Codable , структура Book теперь может быть закодирована и декодирована из JSON с использованием классов Apple Foundation JSONEncoder и JSONDecoder , хотя сама Book не содержит кода для специфической обработки JSON. Пользовательские кодировщики и декодеры также могут быть записаны, соответственно, в соответствии с протоколами Encoder и Decoder .

Кодировать данные JSON

```
// Create an instance of Book called book
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // Do not use try! in production code
print(data)
```

Установите encoder.outputFormatting = .prettyPrinted для упрощения чтения. ## Декодирование данных JSON

Декодирование данных JSON

```
// Retrieve JSON string from some source
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // Do not use try! in production code
print(book)
```

В приведенном выше примере Book.self сообщает декодеру того типа, который должен быть декодирован JSON.

Кодирование или декодирование Исключительно

Иногда вам могут не потребоваться кодирование и декодирование данных, например, когда вам нужно только считывать данные JSON из API или если ваша программа только передает данные JSON в API.

Если вы собираетесь писать только данные JSON, совместите свой тип с Encodable .

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Если вы собираетесь читать только данные JSON, совместите свой тип с Decodable .

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Использование пользовательских имен ключей

API часто используют соглашения об именах, отличные от случая верблюжьего верблюда, такого как случай с змеей. Это может стать проблемой, когда дело доходит до декодирования JSON, поскольку по умолчанию ключи JSON должны точно совпадать с именами свойств вашего типа. Чтобы справиться с этими сценариями, вы можете создавать собственные ключи для своего типа, используя протокол CodingKey .

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
        case authors
```

```
        case publicationDate = "publication_date"
    }
}
```

CodingKeys генерируются автоматически для типов , которые принимают в Codable протокола, но путем создания нашей собственной реализации в приведенном выше примере , мы позволяем нашим декодером , чтобы соответствовать местному верблюд случай publicationDate с змея случае publication_date , как они поступают по API.

SwiftyJSON

SwiftyJSON - это структура Swift, созданная для устранения необходимости дополнительной привязки в обычной сериализации JSON.

Вы можете скачать его здесь: https://github.com/SwiftyJSON/SwiftyJSON

Без SwiftyJSON ваш код будет выглядеть так, чтобы найти имя первой книги в объекте JSON:

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments)
as? [[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //We can now use the book name
}
```

В SwiftyJSON это очень упрощено:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

Это устраняет необходимость проверять каждое поле, так как оно возвращает нуль, если какой-либо из них недействителен.

Чтобы использовать SwiftyJSON, загрузите правильную версию из репозитория Git - есть ветка для Swift 3. Просто перетащите «SwiftyJSON.swift» в свой проект и импортируйте в свой класс:

```
import SwiftyJSON
```

Вы можете создать свой объект JSON, используя следующие два инициализатора:

```
let jsonObject = JSON(data: dataObject)
```

или же

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

Чтобы получить доступ к своим данным, используйте индексы:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

Затем вы можете проанализировать свое значение на определенный тип данных, который вернет необязательное значение:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string

let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

Вы также можете скомпилировать свои пути в быстрый массив:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Такой же как:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON также имеет функциональность для печати собственных ошибок:

```
if let name = json[1337].string {
    //You can use the value - it is valid
} else {
    print(json[1337].error) // "Array[1337] is out of bounds" - You cant use the value
}
```

Если вам нужно написать свой объект JSON, вы можете снова использовать индексы:

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //This changes the age to 25
originalJSON["surname"] = "Smith" //This creates a new field called "surname" and adds the
value to it
```

Если вам нужна оригинальная строка для JSON, например, если вам нужно записать ее в файл, вы можете получить исходное значение:

```
if let string = json.rawString() { //This is a String object
    //Write the string to a file if you like
}

if let data = json.rawData() { //This is an NSData object
    //Send the data to your server if you like
}
```

Фредди

Freddy - это библиотека разбора JSON, которую ведет Big Nerd Ranch . Он имеет три основных преимущества:

1. Тип Безопасность: Помогает вам работать с отправкой и получением JSON таким образом, чтобы предотвратить сбои во время выполнения.
2. Idiomatic: использует преимущества генераторов Swift, перечислений и функциональных возможностей без сложной документации или магических пользовательских операторов.
3. Обработка ошибок. Предоставляет информативную информацию об ошибках для часто встречающихся ошибок JSON.

Пример данных JSON

Давайте определим некоторые примеры данных JSON для использования с этими примерами.

```
{
  "success": true,
  "people": [
    {
      "name": "Matt Mathias",
      "age": 32,
```

```
      "spouse": true
    },
    {
      "name": "Sergeant Pepper",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "teacher",
    "judge"
  ],
  "states": {
    "Georgia": [
      30301,
      30302,
      30303
    ],
    "Wisconsin": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"Matt Mathias\",\"age\":
32,\"spouse\": true},{\"name\": \"Sergeant Pepper\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"teacher\",\"judge\"],\"states\": {\"Georgia\": [30301,30302,30303],\"Wisconsin\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

Удаление десериализации исходных данных

Чтобы десериализовать данные, мы инициализируем объект JSON затем обращаемся к определенному ключу.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

Мы try здесь, потому что доступ к json для ключевого "success" может быть неудачным – он может не существовать, или значение может быть не логическим.

Мы также можем указать путь доступа к элементам, вложенным в структуру JSON. Путь представляет собой список ключей и индексов, разделенных запятыми, которые описывают путь к интересующей стоимости.

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // do something with the error
}
```

Десериализация моделей напрямую

JSON может быть непосредственно разобран на класс модели, который реализует протокол JSONDecodable .

```
public struct Person {
    public let name: String
    public let age: Int
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // do something with the error
}
```

Сериализация исходных данных

Любое значение JSON может быть сериализовано непосредственно в NSData .

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Сериализация моделей напрямую

Любой класс модели, реализующий протокол JSONEncodable , может быть сериализован непосредственно в NSData .

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

Стрела

Arrow - это элегантная библиотека разбора JSON в Swift.

Он позволяет анализировать JSON и сопоставлять его с пользовательскими классами моделей с помощью оператора <-- :

```
identifier <-- json["id"]
name <-- json["name"]
stats <-- json["stats"]
```

Пример:

Модель Swift

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

Файл JSON

```
{
    "id": 15678,
    "name": "John Doe",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                     "label": "house",
                     "number": "9809876545"
                     }, {
                     "label": "cell",
                     "number": "0908070656"
                     }, {
                     "label": "work",
                     "number": "0916570656"
    }]
}
```

картографирование

```
extension Profile: ArrowParsable {
    mutating func deserialize(json: JSON) {
        identifier <-- json["id"]
        link <-- json["link"]
        name <-- json["name"]
        weekday <-- json["weekdayInt"]
        stats <- json["stats"]
        phoneNumbers <-- json["phoneNumbers"]
    }
}
```

использование

```
let profile = Profile()
profile.deserialize(json)
```

Монтаж:

Карфаген

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

Вручную

Просто скопируйте и вставьте Arrow.swift в свой проект Xcode

https://github.com/s4cha/Arrow

Как A Framework

Загрузите Arrow из репозитория GitHub и создайте целевой объект Framework в примере проекта. Затем ссылку на эту структуру.

Простой JSON-анализ в пользовательских объектах

Даже если сторонние библиотеки хороши, простой способ разобрать JSON обеспечивается протоколами. Вы можете себе представить, что у вас есть объект Todo as

```
struct Todo {
    let comment: String
}
```

Всякий раз, когда вы получаете JSON, вы можете обрабатывать простые NSData как показано в другом примере, используя объект NSJSONSerialization .

После этого, используя простой протокол JSONDecodable

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

И превращение вашей структуры Todo соответствие с JSONDecodable делает трюк

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

Вы можете попробовать это с помощью json-кода:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

Когда вы получили его из API, вы можете сериализовать его как предыдущие примеры, показанные в экземпляре AnyObject . После этого вы можете проверить, является ли экземпляр экземпляром JSONDictionary

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

Другое дело, чтобы проверить, конкретный для этого случая, потому что у вас есть массив Todo в JSON, это словарь todos

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Теперь, когда вы получили массив словарей, вы можете преобразовать каждый из них в объект Todo с помощью flatMap (он автоматически удалит значения nil из массива)

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

JSON Parsing Swift 3

Вот файл JSON, который мы будем использовать под названием animals.json

```
{
    "Sea Animals": [
    {
     "name": "Fish",
     "question": "How many species of fish are there?"    },
                {
                "name": "Sharks",
                "question": "How long do sharks live?"
                },
                {
                "name": "Squid",
                "question": "Do squids have brains?"
                },
                {
                "name": "Octopus",
                "question": "How big do octopus get?"
                },
                {
                "name": "Star Fish",
                "question": "How long do star fish live?"
                }
                ],
    "mammals": [
                {
      "name": "Dog",
      "question": "How long do dogs live?"
    },
    {
      "name": "Elephant",
      "question": "How much do baby elephants weigh?"
    },
    {
      "name": "Cats",
      "question": "Do cats really have 9 lives?"
    },
    {
      "name": "Tigers",
      "question": "Where do tigers live?"
    },
    {
      "name": "Pandas",
      "question": "WHat do pandas eat?"
    }   ] }
```

Импортируйте свой JSON-файл в свой проект

Вы можете выполнить эту простую функцию, чтобы распечатать свой файл JSON

```
   func jsonParsingMethod() {
        //get the file
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//Call which part of the file you'd like to pare
        if let results = json["mammals"] as? [[String: AnyObject]] {

            for res in results {
                //this will print out the names of the mammals from out file.
                if let rates = res["name"] as? String {
                    print(rates)
                }
           }
       }
    }
```

Если вы хотите поместить его в табличное представление, я бы сначала создал словарь с NSObject.

Создайте новый быстрый файл под названием ParsingObject и создайте строковые переменные.

Убедитесь, что имя переменной совпадает с именем файла JSON

, Например, в нашем проекте у нас есть name и question поэтому в нашем новом быстром файле мы будем использовать

```
var name: String?
var question: String?
```

Инициализируйте NSObject, который мы вернули в наш ViewController.swift var array = ParsingObject. Затем мы выполнили тот же самый метод, который мы имели раньше, с незначительной модификацией.

```
     func jsonParsingMethod() {
            //get the file
            let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
            let content = try! String(contentsOfFile: filePath!)

            let data: Data = content.data(using: String.Encoding.utf8)!
            let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//This time let's get Sea Animals
            let results = json["Sea Animals"] as? [[String: AnyObject]]

//Get all the stuff using a for-loop
            for i in 0 ..< results!.count {

//get the value
                let dict = results?[i]
                let resultsArray = ParsingObject()

//append the value to our NSObject file
                resultsArray.setValuesForKeys(dict!)
                array.append(resultsArray)
```

```
            }

        }
```

Затем мы показываем это в нашем столе, делая это,

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
            return array.count
        }

        func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
            let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//This is where our values are stored
            let object = array[indexPath.row]
            cell.textLabel?.text = object.name
            cell.detailTextLabel?.text = object.question
            return cell
        }
```

Прочитайте Чтение и письмо JSON онлайн: https://riptutorial.com/ru/swift/topic/223/чтение-и-письмо-json

глава 58: Шаблоны проектирования - создание

Вступление

Шаблоны проектирования - это общие решения проблем, которые часто возникают при разработке программного обеспечения. Ниже приведены шаблоны стандартизованных передовых методов структурирования и проектирования кода, а также примеры общих контекстов, в которых эти шаблоны проектирования были бы подходящими.

Созданные шаблоны проектирования абстрагируют создание объектов, чтобы сделать систему более независимой от процесса создания, составления и представления.

Examples

одиночка

Синглтоны - это часто используемый шаблон проектирования, который состоит из одного экземпляра класса, который используется во всей программе.

В следующем примере мы создаем static свойство, которое содержит экземпляр класса Foo . Помните, что static свойство разделяется между всеми объектами класса и не может быть перезаписано подклассом.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Использование:

```
Foo.shared.doSomething()
```

Обязательно запомните private инициализатор:

> Это гарантирует, что ваши синглтоны действительно уникальны и не позволяют внешним объектам создавать собственные экземпляры вашего класса благодаря контролю доступа. Поскольку все объекты поставляются с открытым инициализатором по умолчанию в Swift, вам необходимо переопределить ваш init и сделать его приватным. KrakenDev

Заводской метод

> В программировании на основе классов шаблон фабричного метода представляет собой шаблон создания, который использует фабричные методы для решения проблемы создания объектов без указания точного класса объекта, который будет создан. Ссылка на Wikipedia

```
protocol SenderProtocol
{
    func send(package: AnyObject)
}
```

```
class Fedex: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Fedex deliver")
    }
}

class RegularPriorityMail: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Regular Priority Mail deliver")
    }
}

// This is our Factory
class DeliverFactory
{
    // It will be responsable for returning the proper instance that will handle the task
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// Usage:
let package = ["Item 1", "Item 2"]

// Fedex class will handle the delivery
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail class will handle the delivery
DeliverFactory.makeSender(isLate:false).send(package)
```

Делая это, мы не зависим от реальной реализации класса, делая sender() полностью прозрачным для тех, кто его потребляет.

В этом случае все, что нам нужно знать, это то, что отправитель будет обрабатывать доставку и выдает метод send() . Есть еще несколько преимуществ: уменьшить сцепление классов, легче протестировать, проще добавить новое поведение, не изменяя, кто его потребляет.

> В рамках объектно-ориентированного проектирования интерфейсы обеспечивают уровни абстракции, которые облегчают концептуальное объяснение кода и создают барьер, предотвращающий зависимость. Ссылка на Wikipedia

наблюдатель

> Шаблон наблюдателя - это объект, называемый субъектом, который ведет список своих иждивенцев, называемых наблюдателями, и автоматически уведомляет их о любых изменениях состояния, обычно, вызывая один из своих методов. Он в основном используется для реализации распределенных систем обработки событий. Шаблон Observer также является ключевой частью знакомого архитектурного шаблона model-view-controller (MVC). Ссылка на Wikipedia

В принципе шаблон наблюдателя используется, когда у вас есть объект, который может уведомить наблюдателей о некоторых изменениях поведения или состояния.

Сначала давайте создадим глобальную ссылку (вне класса) для Центра уведомлений:

```
let notifCentre = NotificationCenter.default
```

Отлично, теперь мы можем назвать это где угодно. Затем мы хотели бы зарегистрировать класс в качестве наблюдателя ...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification",
object: nil)
```

Это добавляет класс в качестве наблюдателя для «readForMyFunc». Он также указывает, что функция myFunc должна вызываться при получении этого уведомления. Эта функция должна быть написана в том же классе:

```
func myFunc(){
    print("The notification has been received")
}
```

> Одно из преимуществ этого шаблона состоит в том, что вы можете добавлять в качестве наблюдателей множество классов и, таким образом, выполнять много действий после одного уведомления.

Теперь уведомление может быть просто отправлено (или отправлено, если хотите) почти из любого места в коде с линией:

```
notifCentre.post(name: "myNotification", object: nil)
```

> Вы также можете передавать информацию с уведомлением в виде словаря

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

Но тогда вам нужно добавить уведомление о своей функции:

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("The notification \(moreInfo) has been received")
    //prints - The notification pass this on has been received
}
```

Цепочка ответственности

> В объектно-ориентированном дизайне шаблон цепочки ответственности представляет собой шаблон проектирования, состоящий из источника command объектов и серии объектов processing . Каждый объект processing содержит логику, которая определяет типы объектов команд, которые он может обрабатывать; остальные передаются на следующий объект processing в цепочке. Механизм также существует для добавления новых объектов processing в конец этой цепочки. Википедия

Создание классов, составляющих цепочку ответственности.

Сначала мы создаем интерфейс для всех объектов processing .

```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
```

```
      print(self.role + " will approve $ \(request.amount) for \(request.purpose)")
    } else if successor != nil {
      successor?.process(request: request)
    }
  }
}
```

Затем мы создаем объект command .

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Наконец, создавая объекты, составляющие цепочку ответственности.

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "Manager"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "Director"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "President"
  var successor: PurchasePower?
}
```

Инициируйте и соедините его вместе:

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

> Механизм объединения объектов здесь - это доступ к свойствам

Создание запроса для его запуска:

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "buying a pen"))  // Manager will
approve $ 2.0 for buying a pen
manager.process(request: PurchaseRequest(amount: 90, purpose: "buying a printer")) // Director
will approve $ 90.0 for buying a printer

manager.process(request: PurchaseRequest(amount: 2000, purpose: "invest in stock")) //
President will approve $ 2000.0 for invest in stock
```

Итератор

> В компьютерном программировании итератор является объектом, который позволяет программисту перемещаться по контейнеру, в частности спискам. Википедия

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

  mutating func next() -> Turtle? {
    defer { current += 1 }
    return turtles.count > current ? turtles[current] : nil
  }
}

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

И пример использования

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                     Turtle(name: "Mickey"),
                                     Turtle(name: "Raph"),
                                     Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("The great: \(turtle)")
}
```

Шаблон Builder

> Шаблон построителя представляет собой шаблон разработки **программного обеспечения для создания объектов** . В отличие от абстрактного заводского шаблона и шаблона фабричного метода, целью которого является включение полиморфизма, намерение шаблона-строителя состоит в том, чтобы найти решение для анти-шаблона конструктора телескопа. Конструкция антивибратора конструктора телескопов возникает, когда увеличение комбинации параметров конструктора объекта приводит к экспоненциальному списку конструкторов. Вместо использования множества конструкторов шаблон построителя использует другой объект, строитель, который по шагам получает каждый параметр инициализации и затем возвращает результирующий построенный объект одновременно.
>
> *-Wikipedia*

Основная цель шаблона построителя - установить конфигурацию по умолчанию для объекта из его создания. Это посредник между объектом будет построен и все другие объекты, связанные с его построением.

Пример:

Чтобы это стало понятным, давайте посмотрим на пример *Car Builder* .

Подумайте, что у нас есть класс *Car*, содержащий множество вариантов для создания объекта, например:

- Цвет.
- Количество мест.
- Количество колес.
- Тип.
- Тип передачи.
- Мотор.
- Доступность подушки безопасности.

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

class Car: CustomStringConvertible {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags
```

```
    }
}
```

Создание объекта автомобиля:

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
               gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* Printing
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 4
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 Airbag Availability: true
*/
```

Проблема возникает, когда создание объекта автомобиля состоит в том, что автомобиль требует создания многих конфигурационных данных.

Для применения шаблона Builder параметры инициализатора должны иметь значения по умолчанию, *которые при необходимости изменяются* .

Класс CarBuilder:

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

Класс CarBuilder определяет свойства, которые можно изменить для редактирования значений созданного объекта автомобиля.

Давайте построим новые автомобили с помощью CarBuilder :

```
var builder = CarBuilder()
// currently, the builder creates cars with default configuration.

let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
```

```
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
*/

builder.shouldHasAirbags = true
// now, the builder creates cars with default configuration,
// but with a small edit on making the airbags available

let safeCar = builder.buildCar()
print(safeCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
 */

builder.color = UIColor.purple
// now, the builder creates cars with default configuration
// with some extra features: the airbags are available and the color is purple

let femaleCar = builder.buildCar()
print(femaleCar)
/* prints
 color: UIExtendedSRGBColorSpace 0.5 0 0.5 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
*/
```

Преимущество применения шаблона Builder заключается в простоте создания объектов, которые должны содержать большую часть конфигураций, устанавливая значения по умолчанию, а также легкость изменения этих значений по умолчанию.

Возьмите дальше:

В качестве хорошей практики все свойства, требующие значений по умолчанию, должны быть в *отдельном протоколе* , который должен быть реализован самим классом и его создателем.

Вернемся к нашему примеру, давайте создадим новый протокол под названием CarBluePrint :

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
```

```
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }
}

class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
```

```
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

Преимущество объявления свойств, требующих значения по умолчанию в протоколе, - это принудительное внедрение любого нового добавленного свойства; Когда класс соответствует протоколу, он должен объявлять все свои свойства / методы.

Учтите, что требуется новая функция, которая должна быть добавлена к проекту создания автомобиля под названием «название батареи»:

```
protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }

    // adding the new property
    var batteryName: String { get set }
}
```

После добавления нового свойства обратите внимание на то, что возникнут две ошибки времени компиляции, сообщив, что в соответствии с протоколом CarBluePrint требуется объявить свойство « batteryName». Это гарантирует, что CarBuilder объявит и установит значение по умолчанию для свойства batteryName .

После добавления batteryName new в протокол CarBluePrint реализация классов Car и CarBuilder должна быть:

```
class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\nType: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)\nBattery Name: \(batteryName)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
}
```

```
class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "Default Battery Name"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

Опять же, давайте построим новые автомобили с помощью CarBuilder :

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* prints
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: Default Battery Name
*/

builder.batteryName = "New Battery Name"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: New Battery Name
 */
```

Прочитайте Шаблоны проектирования - создание онлайн:
https://riptutorial.com/ru/swift/topic/4941/шаблоны-проектирования---создание

глава 59: Шаблоны проектирования - структурные

Вступление

Шаблоны проектирования - это общие решения проблем, которые часто возникают при разработке программного обеспечения. Ниже приведены шаблоны стандартизованных передовых методов структурирования и проектирования кода, а также примеры общих контекстов, в которых эти шаблоны проектирования были бы подходящими.

Структурные шаблоны проектирования сосредоточены на составлении классов и объектов для создания интерфейсов и обеспечения большей функциональности.

Examples

адаптер

Адаптеры используются для преобразования интерфейса данного класса, известного как **Adaptee** , в другой интерфейс, называемый **Target** . Операции над объектом вызываются **клиентом** , и эти операции *адаптируются* адаптером и передаются в Adaptee.

В Swift адаптеры часто могут формироваться с использованием протоколов. В следующем примере клиент, способный связываться с Target, получает возможность выполнять функции класса Adaptee с помощью адаптера.

```
// The functionality to which a Client has no direct access
class Adaptee {
    func foo() {
        // ...
    }
}

// Target's functionality, to which a Client does have direct access
protocol TargetFunctionality {
    func fooBar() {}
}

// Adapter used to pass the request on the Target to a request on the Adaptee
extension Adaptee: TargetFunctionality {
    func fooBar() {
        foo()
    }
}
```

Пример потока одностороннего адаптера: Client -> Target -> Adapter -> Adaptee

Адаптеры также могут быть двунаправленными, и они известны как **двухсторонние адаптеры** . Двухсторонний адаптер может быть полезен, когда два разных клиента должны просматривать объект по-разному.

Фасад

Фасад обеспечивает унифицированный интерфейс высокого уровня для интерфейсов подсистем. Это обеспечивает более простой и безопасный доступ к более общим функциям подсистемы.

Ниже приведен пример Фасад, используемый для установки и извлечения объектов в UserDefaults.

```
enum Defaults {
```

```
    static func set(_ object: Any, forKey defaultName: String) {
        let defaults: UserDefaults = UserDefaults.standard
        defaults.set(object, forKey:defaultName)
        defaults.synchronize()
    }

    static func object(forKey key: String) -> AnyObject! {
        let defaults: UserDefaults = UserDefaults.standard
        return defaults.object(forKey: key) as AnyObject!
    }

}
```

Использование может выглядеть следующим образом.

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

Сложности доступа к общему экземпляру и синхронизации UserDefaults скрыты от клиента, и к этому интерфейсу можно получить доступ из любой точки программы.

Прочитайте Шаблоны проектирования - структурные онлайн:
https://riptutorial.com/ru/swift/topic/9497/шаблоны-проектирования---структурные

глава 60: Шифрование AES

Examples

AES в режиме CBC со случайным IV (Swift 3.0)

Iv предварительно привязан к зашифрованным данным

aesCBC128Encrypt создаст случайный IV и префикс для зашифрованного кода.
aesCBC128Decrypt будет использовать префикс IV во время дешифрования.

Входы - это данные и ключ - объекты данных. Если закодированная форма, такая как Base64, если требуется, конвертировать в и / или из вызывающего метода.

Ключ должен состоять из 128-битных (16-байтовых), 192-битных (24 байта) или 256-битных (32 байта) в длину. Если используется другой размер ключа, будет выдана ошибка.

По умолчанию устанавливается PKCS # 7 .

Этот пример требует Common Crypto
Необходимо иметь заголовок заголовка проекта:
#import <CommonCrypto/CommonCrypto.h>
Добавьте в проект Security.framework .

Это пример, а не производственный код.

```
enum AESError: Error {
    case KeyError((String, Int))
    case IVError((String, Int))
    case CryptorError((String, Int))
}

// The iv is prefixed to the encrypted data
func aesCBCEncrypt(data:Data, keyData:Data) throws -> Data {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let cryptLength = size_t(ivSize + data.count + kCCBlockSizeAES128)
    var cryptData = Data(count:cryptLength)

    let status = cryptData.withUnsafeMutableBytes {ivBytes in
        SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ivBytes)
    }
    if (status != 0) {
        throw AESError.IVError(("IV generation failed", Int(status)))
    }

    var numBytesEncrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCEncrypt),
                        CCAlgorithm(kCCAlgorithmAES),
                        options,
                        keyBytes, keyLength,
```

```
                        cryptBytes,
                        dataBytes, data.count,
                        cryptBytes+kCCBlockSizeAES128, cryptLength,
                        &numBytesEncrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.count = numBytesEncrypted + ivSize
    }
    else {
        throw AESError.CryptorError(("Encryption failed", Int(cryptStatus)))
    }

    return cryptData;
}

// The iv is prefixed to the encrypted data
func aesCBCDecrypt(data:Data, keyData:Data) throws -> Data? {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let clearLength = size_t(data.count - ivSize)
    var clearData = Data(count:clearLength)

    var numBytesDecrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = clearData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCDecrypt),
                        CCAlgorithm(kCCAlgorithmAES128),
                        options,
                        keyBytes, keyLength,
                        dataBytes,
                        dataBytes+kCCBlockSizeAES128, clearLength,
                        cryptBytes, clearLength,
                        &numBytesDecrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.count = numBytesDecrypted
    }
    else {
        throw AESError.CryptorError(("Decryption failed", Int(cryptStatus)))
    }

    return clearData;
}
```

Пример использования:

```
let clearData = "clearData0123456".data(using:String.Encoding.utf8)!
```

```
let keyData   = "keyData890123456".data(using:String.Encoding.utf8)!
print("clearData:   \(clearData as NSData)")
print("keyData:     \(keyData as NSData)")

var cryptData :Data?
do {
    cryptData = try aesCBCEncrypt(data:clearData, keyData:keyData)
    print("cryptData:   \(cryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCEncrypt: \(status)")
}

let decryptData :Data?
do {
    let decryptData = try aesCBCDecrypt(data:cryptData!, keyData:keyData)
    print("decryptData: \(decryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCDecrypt: \(status)")
}
```

Пример:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <92c57393 f454d959 5a4d158f 6e1cd3e7 77986ee9 b2970f49 2bafcf1a 8ee9d51a bde49c31
d7780256 71837a61 60fa4be0>
decryptData: <636c6561 72446174 61303132 33343536>
```

Заметки:
Одна типичная проблема с примером кода режима CBC заключается в том, что он оставляет создание и совместное использование случайного IV пользователю. Этот пример включает в себя создание IV, префиксацию зашифрованных данных и использование префикса IV во время дешифрования. Это освобождает случайного пользователя от деталей, которые необходимы для режима CBC .

Для безопасности зашифрованные данные также должны иметь аутентификацию, этот примерный код не предусматривает того, чтобы быть небольшим и обеспечивать лучшую совместимость для других платформ.

Также отсутствует ключевой вывод ключа из пароля, предлагается использовать PBKDF2, поскольку текстовые пароли используются в качестве материала для ввода ключей.

Для надежного производства готовый многоплатформенный код шифрования см. В RNCryptor .

> Обновлено для использования throw / catch и нескольких ключевых размеров на основе предоставленного ключа.

AES в режиме CBC со случайным IV (Swift 2.3)

Iv предварительно привязан к зашифрованным данным

aesCBC128Encrypt создаст случайный IV и префикс для зашифрованного кода. aesCBC128Decrypt будет использовать префикс IV во время дешифрования.

Входы - это данные и ключ - объекты данных. Если закодированная форма, такая как Base64, если требуется, конвертировать в и / или из вызывающего метода.

Ключ должен быть точно 128-битным (16-байтовый). Для других размеров ключей см. Пример Swift 3.0.

По умолчанию устанавливается PKCS # 7.

Для этого примера требуется Common Crypto. Необходимо иметь заголовок для моста для проекта: # import <CommonCrypto / CommonCrypto.h> Добавьте в проект Security.framework.

См. Пример Swift 3 для заметок.

Это пример, а не производственный код.

```
func aesCBC128Encrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)
    let cryptDataLength = size_t(data.count + kCCBlockSizeAES128)
    var cryptData = [UInt8](count:ivLength + cryptDataLength, repeatedValue:0)

    let status = SecRandomCopyBytes(kSecRandomDefault, Int(ivLength),
UnsafeMutablePointer<UInt8>(cryptData));
    if (status != 0) {
        print("IV Error, errno: \(status)")
        return nil
    }

    var numBytesEncrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCEncrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              cryptData,
                              data, data.count,
                              &cryptData + ivLength, cryptDataLength,
                              &numBytesEncrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.removeRange(numBytesEncrypted+ivLength..<cryptData.count)
    }
    else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return cryptData;
}

func aesCBC128Decrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let clearLength = size_t(data.count)
    var clearData   = [UInt8](count:clearLength, repeatedValue:0)

    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)

    var numBytesDecrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCDecrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              data,
                              UnsafePointer<UInt8>(data) + ivLength, data.count - ivLength,
                              &clearData, clearLength,
                              &numBytesDecrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.removeRange(numBytesDecrypted..<clearLength)
```

```
    } else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return clearData;
}
```

Пример использования:

```
let clearData = toData("clearData0123456")
let keyData   = toData("keyData890123456")

print("clearData:   \(toHex(clearData))")
print("keyData:     \(toHex(keyData))")
let cryptData = aesCBC128Encrypt(data:clearData, keyData:keyData)!
print("cryptData:   \(toHex(cryptData))")
let decryptData = aesCBC128Decrypt(data:cryptData, keyData:keyData)!
print("decryptData: \(toHex(decryptData))")
```

Пример:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <9fce4323 830e3734 93dd93bf e464f72a a653a3a5 2c40d5ea e90c1017 958750a7 ff094c53
6a81b458 b1fbd6d4 1f583298>
decryptData: <636c6561 72446174 61303132 33343536>
```

Шифрование AES в режиме ECB с дополнением PKCS7

Из документации Apple для IV,

> Этот параметр игнорируется, если используется режим ECB или выбран алгоритм шифрования потока.

```
func AESEncryption(key: String) -> String? {

        let keyData: NSData! = (key as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let data: NSData! = (self as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let cryptData    = NSMutableData(length: Int(data.length) + kCCBlockSizeAES128)!

        let keyLength              = size_t(kCCKeySizeAES128)
        let operation: CCOperation = UInt32(kCCEncrypt)
        let algoritm:  CCAlgorithm = UInt32(kCCAlgorithmAES128)
        let options:   CCOptions   = UInt32(kCCOptionECBMode + kCCOptionPKCS7Padding)

        var numBytesEncrypted :size_t = 0


        let cryptStatus = CCCrypt(operation,
                                  algoritm,
                                  options,
                                  keyData.bytes, keyLength,
                                  nil,
                                  data.bytes, data.length,
```

```
                                cryptData.mutableBytes, cryptData.length,
                                &numBytesEncrypted)

        if UInt32(cryptStatus) == UInt32(kCCSuccess) {
            cryptData.length = Int(numBytesEncrypted)

            var bytes = [UInt8](repeating: 0, count: cryptData.length)
            cryptData.getBytes(&bytes, length: cryptData.length)

            var hexString = ""
            for byte in bytes {
                hexString += String(format:"%02x", UInt8(byte))
            }

            return hexString
        }

        return nil
    }
```

Прочитайте Шифрование AES онлайн: https://riptutorial.com/ru/swift/topic/7026/шифрование-aes

кредиты

S. No	Главы	Contributors
1	Начало работы с Swift Language	Ahmad F, Anas, andy, Cailean Wilkinson, Claw, Community, esthepiking, Ferenc Kiss, Jim, jtbandes, Luca Angeletti, Luca Angioloni, Moritz, nmnsud, Seyyed Parsa Neshaei, sudo, Sunil Prajapati, Tanner, user3581248
2	(Небезопасные) Буферные указатели	Tommie C.
3	Conditionals	AK1, atxe, Brduca, Community, Dalija Prasnikar, DarkDust, Hamish, jtbandes, ThaNerd, Thomas Gerot, tktsubota, toofani, torinpitchers
4	Loops	Caleb Kleveter, D31, Efraim Weiss, Fred Faust, Hamish, Idan, Irfan, Jeff Lewis, Luca Angeletti, Moritz, Mr. Xcoder, Saagar Jha, Santa Claus, WMios, xoudini
5	NSRegularExpression в Swift	Echelon, Hady Nourallah, ThrowingSpoon
6	Optionals	Anand Nimje, Andrey Gordeev, Arnaud, Caleb Kleveter, Hamish, Ian Rahman, iwillnot, Jason Sturges, Jojodmo, juanjo, Kevin, Michaël Azevedo, Moritz, Nathan Kellert, Paulw11, shannoga, SKOOP, Tanner, tktsubota, Tommie C.
7	OptionSet	4444, Alessandro
8	RxSwift	Alexander Olferuk, FelixSFD, imagngames, Moritz, Victor Sigler
9	Typealias	Bartłomiej Semańczyk, Caleb Kleveter, D4ttatraya, Moritz
10	Алгоритмы с Swift	Austin Conlon, Bohdan Savych, Hady Nourallah, SteBra, Stephen Leppik, Tommie C.
11	Блоки	Matt
12	Булевы	Andreas, jtbandes, Kevin, pableiros
13	Быстрые функции Advance	DarkDust, Sagar Thummar
14	Быстрый HTTP-сервер от Kitura	Fangming Ning
15	Внедрение зависимости	Bear with me, JPetric
16	Вход в Swift	Adam Bardon, D4ttatraya, DanHabib, jglasse, Moritz, paper1111, RamenChef
17	Вывод ключа PBKDF2	BUZZE, zaph
18	Дженерики	Andrey Gordeev, DarkDust, FelixSFD, Glenn R. Fisher, Hamish, Jojodmo, Kent Liau, Luca D'Alberti, Suneet Tipirneni, Ven, xoudini
19	Затворы	ctietze, Duncan C, Hamish, Jojodmo, jtbandes, LopSae, Matthew

		Seaman, Moritz, Timothy Rascher, Tom Magnusson
20	Инициализаторы	Brduca, FelixSFD, rashfmnb, Santa Claus, Vinupriya Arivazhagan
21	Классы	Dalija Prasnikar, esthepiking, FelixSFD, jtbandes, Luca Angeletti, Matt, Ryan H., tktsubota, Tommie C., Zack
22	Контроль доступа	4444, Asdrubal, FelixSFD
23	Кортеж	Accepted Answer, BaSha, Caleb Kleveter, JAL, Jason Sturges, Jojodmo, kabiroberai, LopSae, Luca Angeletti, Moritz, Nathan Kellert, Rick Pasveer, Ronald Martin, tktsubota
24	Криптографическое Хеширование	zaph
25	Кэширование на диске	Viktor Gardart
26	Литье под давлением	Anand Nimje, andyvn22, godisgood4, LopSae, Nick Podratz
27	Массивы	BaSha, Ben Trengrove, D4ttatraya, DarkDust, Hamish, jtbandes, Kevin, Luca Angeletti, Moritz, Moriya, nathan, pableiros, Palle , Saagar Jha, Stephen Leppik, ThrowingSpoon, tomahh, toofani, vacawama, Vladimir Nul
28	Менеджер пакетов Swift	Moritz
29	Метод Swizzling	JAL, Noam, Umberto Raimondi
30	наборы	Community, Dalija Prasnikar, Luca Angeletti, Moritz, Steve Moser
31	Начало работы с программно-ориентированным программированием	Alessandro Orrù, Fred Faust, kabiroberai, Krzysztof Romanowski
32	Обработка ошибок	Anil Varghese, cpimhoff, egor.zhdan, Jason Bourne, jtbandes, Mehul Sojitra, Moritz, Tom Magnusson
33	Обработчик завершения	Maysam, Moritz
34	отражение	Asdrubal, LopSae, Sajjon
35	переключатель	Ajwhiteway, AK1, Duncan C, elprl, Harshal Bhavsar, joan, Josh Brown, Luca Angeletti, Moritz, Santa Claus, ThrowingSpoon
36	Переменные и свойства	Christopher Oezbek, FelixSFD, Jojodmo, Luke, Santa Claus, tktsubota
37	Перечисления	Alex Popov, Anh Pham, Avi, Caleb Kleveter, Diogo Antunes, Fantattitude, fredpi, Hamish, Jason Sturges, Jojodmo, jtbandes, juanjo, Justin Whitney, Matt, Matthew Seaman, Nathan Kellert, Nick Podratz, Nikolai Ruhe, SeanRobinson159, shannoga, user3480295
38	протоколы	Accepted Answer, Ash Furrow, Cory Wilhite, Dalija Prasnikar, esthepiking, Hamish, iBelieve, Igor Bidiniuc, Jason Sturges, Jojodmo, jtbandes, Luca D'Alberti, Matt, matt.baranowski, Matthew Seaman, Oleg Danu, Rahul, SeanRobinson159, SKOOP, Tim

		Vermeulen, tktsubota, Undo, Victor Sigler
39	Работа с C и Objective-C	4444, Accepted Answer, jtbandes, Mark
40	Разметка документации	Abdul Yasin, Martin Delille, Moritz, Rashwan L
41	расширения	Brduca, David, Esqarrouth, Jojodmo, jtbandes, Luca Angeletti, Moritz, rigdonmr
42	Расширенные операторы	avismara, egor.zhdan, Fluidity, Hamish, Intentss, JAL, jtbandes, kennytm, Matthew Seaman, orccrusher99, tharkay
43	Связанные объекты	Fattie, JAL
44	Словари	dasdom, Diogo Antunes, egor.zhdan, iOSDevCenter, Jason Bourne, Kirit Modi, Koushik, Magisch, Moritz, RamenChef, Saagar Jha, sasquatch, Suneet Tipirneni, That lazy iOS Guy , ThrowingSpoon
45	совпадение	Adda_25, Ahmad F, FelixSFD, JAL, LukeSideWalker, M_G, Matthew Seaman, Palle, Rob, Santa Claus
46	Соглашения стилей	Grimxn, Moritz, Palle, Ryan H.
47	Создать UIImage из инициалов из строки	RubberDucky4444
48	Спектакль	Matthew Seaman
49	Строки и символы	Akshit Soota, Andrea Antonioni, antonio081014, AstroCB, Caleb Kleveter, Carpsen90, egor.zhdan, Feldur, Franck Dernoncourt, Govind Rai, Greg, Guilherme Torres Castro, Hamish, HariKrishnan.P, HeMet, JAL, Jason Sturges, Jojodmo, jtbandes, kabiroberai, Kirit Modi, Kyle KIM, Lope, LopSae, Luca Angeletti, LukeSideWalker, Magisch, Mahmoud Adam, Matt, Matthew Seaman, Max Desiatov, maxkonovalov, Moritz, Nate Cook, Nikolai Ruhe, Panda, Patrick, pixatlazaki, QoP, sdasdadas, Shanmugaraja G, shim, solidcell, Sunil Sharma, Suragch, taylor swift, The_Curry_Man, ThrowingSpoon, user3480295, Victor Sigler, Vinupriya Arivazhagan, WMios
50	Структуры	Accepted Answer, AK1, Diogo Antunes, fredpi, Josh Brown, Kevin, Luca Angeletti, Marcus Rossel, Moritz, pbush25, Rob Napier, SamG
51	Управление памятью	Accepted Answer, Daniel Firsht, jtbandes, Marc Gravell, Moritz, Palle, Tricertops
52	Утверждение Отсрочки	Palle
53	функции	Ajith R Nayak, Andy Ibanez, Caleb Kleveter, jtbandes, Kote, Luca Angeletti, Matt Le Fleur, Nikita Kurtin, noor, ntoonio, Saagar Jha, SKOOP, Stephen Schaub, ThrowingSpoon, tktsubota, ZGski
54	Функциональное программирование в Swift	Echelon, Luca Angeletti, Luke, Matthew Seaman, Shijing Lv
55	Функционируйте как граждане первого класса в Свифте	Kumar Vivek Mitra

56	чисел	Arsen, jtbandes, Suragch, WMios, ZGski
57	Чтение и письмо JSON	Cyril Ivar Garcia, Ethan Kay, Glenn R. Fisher, Ian Rahman, infl3x, Jack C, Jason Sturges, jtbandes, Leo Dabus, lostAtSeaJoshua, Luca D'Alberti, maxkonovalov, Moritz, nstefan, Steffen D. Sommer, Stephen Leppik, toofani
58	Шаблоны проектирования - создание	Ahmad F, AMAN77, Brduca, Dalija Prasnikar, Ian Rahman, Moritz, SeanRobinson159, SimpleBeat, Sơn Đỗ Đình Thy, Stephen Leppik, Thorax, Tommie C.
59	Шаблоны проектирования - структурные	Ian Rahman
60	Шифрование AES	Matt, Stephen Leppik, zaph