

The Swift Programming Language

НА РУССКОМ

```
var people = ["Dave", "Brian", "Alex", "A  
let name = "Alex"  
if let index = find(people, name) {  
    println("\(name) is person \(index +  
    delegate?.didFindPersonWithName(name,  
} else {  
    println("Unable to find \(name) in t  
}
```



Developer

Оглавление 10

О языке Swift 11

Знакомство со Swift 13

- Простые типы данных 14
- Управление потоком 17
- Функции и замыкание 21
- Объекты и классы 26
- Перечисление и структура 32
- Протоколы и расширения 37
- Универсальные шаблоны 40

Основы 42

- Константы и переменные 43
- Комментарии 49
- Точка с запятой 50
- Целые числа 50
- Числа с плавающей запятой 53
- Строгая типизация и Вывод типов 54
- Числовые литералы 56
- Преобразование числовых типов 57
- Псевдонимы типов 61
- Логические типы 62
- Кортежи 64
- Опциональные типы 67
- Утверждения 75

Базовые типы 78

- Терминология 79
- Оператор присваивание 80
- Арифметические операторы 81
- Составные операторы присваивание 87
- Операторы сравнения 88
- Тернарный условный оператор 90

- [Оператор объединения по нулевому указателю](#) 92
- [Операторы диапазона](#) 94
- [Логические операторы](#) 96

Строки и символы 100

- [Строковые литералы](#) 101
- [Инициализация пустых строк](#) 102
- [Изменчивость строк](#) 103
- [Строка является типом значений](#) 104
- [Работа с символами](#) 105
- [Конкатенация строк и символов](#) 106
- [Интерполяция строк](#) 107

Типы коллекций 109

- [Изменчивость коллекций](#) 110
- [Массивы](#) 110
- [Множества](#) 119
- [Словари](#) 130

Управление потоком 140

- [Циклы For](#) 141
- [Циклы While](#) 147
- [Условные операторы](#) 153
- [Операторы передачи управления](#) 169

Функции 180

- [Объявление и вызов функции](#) 181
- [Параметры функции и возвращаемые значения](#) 183
- [Имена параметров функции](#) 189
- [Функциональные типы](#) 200
- [Вложенные функции](#) 206

Замыкания 208

- Замыкающие выражение 209
- Последующие замыкание 216
- Захват значений 220
- Замыкание является [ссылочным типом](#) 224

Перечисления 226

- Синтаксис перечислений 227
- Использование перечислений с оператором `switch` 229
- Связанные значения 230
- Исходные значения 234

Классы и структуры 238

- Сравнение классов и структур 239
- Структуры и перечисления – типы значения 244
- Классы – [ссылочный тип](#) 247
- Выбираем между классом и структурой 251
- Присваивание и копирование поведения для строк массивов и словарей 253

Свойства 254

- Хранимые свойства 255
- Вычисляемые свойства 260
- Наблюдатели свойства 265
- Глобальные и локальные переменные 268
- Свойства типа 269

Методы 277

- Методы экземпляра 278
- Методы типа 287

Индексы 292

- Индексный синтаксис 293
- Использование индекса 295
- Опции индекса 296

Наследование 300

- Определение базового класса 301
- Наследование подклассом 302
- Переопределение 305
- Предотвращение переопределений 311

Инициализация 312

- Установка начальных значений для хранимых свойств 313
- Дефолтные инициализаторы 323
- Делегирование инициализатора для типов значений 325
- Наследование и инициализация класса 329
- Проваливающийся инициализаторы 351
- Требуемые инициализаторы 365
- Начальное значение свойства в виде функции или замыкания 366

Деинициализация 370

- Как работает деинициализация 370
- Деинициализаторы в действии 372

Автоматический подсчет ссылок (ARC) 376

- Работа ARC 377
- ARC в действие 378
- Циклы сильных ссылок между экземплярами классов 380
- Замена циклов сильных ссылок между экземплярами классов 386
- Циклы сильных ссылок для замыкания 400
- Замена циклов сильных ссылок для замыкания 405

Опциональная последовательность 410

- [Опциональная последовательность как альтернатива принудительному разворачиванию 411](#)
- [Определение классовых моделей для опциональной последовательности 415](#)
- [Доступ к свойствам через опциональную последовательность 418](#)
- [Вызов методов через опциональную последовательность 419](#)
- [Доступ к индексам через опциональную последовательность 421](#)
- [Соединение нескольких уровней ОП 424](#)
- [Прикрепление методов к ОП с опциональными возвращаемыми значениями 427](#)

Приведение типов 429

- [Определение типов классовой иерархии для приведения типов 429](#)
- [Проверка типа 432](#)
- [Понижающее приведение 433](#)
- [Приведение типов для Any и AnyObject 436](#)

Вложенные типы 441

- [Вложенные типы в действие 441](#)
- [Ссылка на вложенные типы 445](#)

Расширения 446

- [Синтаксис расширений 447](#)
- [Вычисляемые свойства в расширениях 448](#)
- [Инициализаторы в расширениях 450](#)
- [Методы в расширениях 452](#)
- [Сабскрипты в расширениях 455](#)
- [Вложенные типы в расширениях 456](#)

Протоколы 458

- Синтаксис протокола 458
- Требуемые свойства 459
- Требуемые методы 463
- Требуемые изменяющиеся методы 465
- Требуемые инициализаторы 467
- Протоколы как типы 469
- Делегирование 472
- Добавление соответствия протоколу через расширение 477
- Коллекции типов протокола 480
- Наследование протокола 481
- Классовые протоколы 483
- Композиция протоколов 484
- Проверка соответствия протокола 486
- Опциональные требования протокола 490

Универсальные шаблоны 497

- Проблема, которую решают универсальные шаблоны 498
- Универсальные функции 500
- Параметры типа 502
- Именованное параметров типа 503
- Ограничения типа 508
- Расширяем универсальные тип 513
- Связанные типы 515
- Оговорка where 520

Контроль доступа 525

- Модули и исходные файлы 526
- Уровни доступа 527
- Синтаксис уровня контроля 529
- Пользовательские типы 530
- Уровень доступа класса и подкласса 535
- Константы, переменные, свойства и сабскрипты 536
- Инициализаторы 540
- Протоколы и уровень доступа 541
- Расширения и уровень доступа 543
- Универсальный код, Алиасы типов 544

Продвинутые операторы 545

- Побитовые операторы 546
- Операторы переполнения 557
- Приоритет и ассоциативность 562
- Операторные функции 564
- Пользовательские операторы 572

Документация языка программирования Swift на русском языке

Данная документация является переводом официальной книги "Swift Programming Language" от Apple. Работа была проделана профессионалами, но в связи с огромным объемом текста и постоянным обновлением книги, мы не исключаем незначительных ошибок. Если вы вдруг обнаружили то, что на ваш взгляд нуждается в корректировке, то напишите нам об этом либо в комментариях, либо на почту и мы обязательно примем это во внимание. Кроме того, мы стараемся постоянно обновлять документацию, как только выходят официальные правки английского варианта книги. В конечном итоге, нашей целью является создание качественного и постоянно обновляемого источника информации о языке Swift.

Мы хотим помочь начинающим разработчикам, сделать уверенные шаги на пути изучения Swift. Мы считаем, что языковой барьер не должен быть препятствием для изучения. Мы искренне верим, что такой мощный, быстрый и современный язык программирования как Swift, должен быть доступен каждому.

Если вам нужно быстро пройти по возможностям языка, то начните с главы "Знакомство со Swift". Если же требуется углубиться в Swift, пропустите главу "Знакомство со Swift" и сразу начинайте читать с главы "Основы".

О языке Swift

Swift - это новый язык программирования для разработки приложений на iOS, OS X и watchOS, который сочетает в себе все лучшее от C и Objective-C, при этом нет ограничений в совместимости с C. Swift использует паттерны безопасного программирования и содержит современные функции, которые помогают сделать программирование легким, гибким и увлекательным. Созданный с нуля, Swift, опирающийся на зрелые и всеми любимые фреймворки Cocoa и Cocoa Touch, это возможность представить, как разрабатываются приложения.

Swift разрабатывался несколько лет. Apple заложила в основу Swift существующий компилятор, отладчик и структуру фреймворков. Мы упростили процесс управления памятью с системой автоматического подсчета ссылок - Automatic Reference Counting (ARC). Наши фреймворки, основанные на Foundation и Cocoa, также были модернизированы и стандартизированы. Objective-C начал поддерживать блоки, коллекции литералов, и модули, включая заимствование новых возможностей языка фреймворком, без каких-либо проблем. Благодаря этой проделанной работе, мы можем представить новый язык для будущих разработок приложений Apple.

Swift покажется знакомым для разработчиков Objective-C. Он заимствует читабельность именованных параметров Objective-C и мощь динамической модели объектов Objective-C. Он обеспечивает плавный доступ к существующим фреймворкам Cocoa и возможность смешивать код с кодом Objective-C.

Построенный на этой общей основе, Swift предоставляет много новых возможностей и унифицирует процедурный и объектно-ориентированный части языка.

Swift дружелюбен для новичков в программировании. Это первый язык программирования промышленного качества, который так же понятен и увлекателен, как скриптовый язык. Он поддерживает playground'ы - инновационная функция, которая позволяет экспериментировать с кодом Swift и видеть результат мгновенно, без необходимости компилировать и запускать приложение.

Swift вобрал в себя лучшие идеи современных языков с мудростью инженерной культуры Apple. Компилятор оптимизирован для производительности, а язык оптимизирован для разработки, без компромиссов с одной или другой стороны. Он спроектирован для разработки приложений начиная от "Hello, world", заканчивая масштабами операционной системы. Все это делает Swift перспективным инструментом для разработчиков и самой компании Apple.

Swift - это фантастический способ писать приложения на iOS, OS X и watchOS, и продолжать знакомиться с новыми функциями и возможностями. У нас амбициозные цели на Swift. Мы с нетерпением ждем что вы создадите с помощью него.

Знакомство со Swift

Заметка

Внимание! Данная глава "Знакомство со Swift" является кратким содержанием всего руководства и предназначена тем, кто хочет бегло освоить особенности Swift. Для тех, кто хочет читать руководство от начала и до конца, мы рекомендуем начать изучение с главы "Основы".

По традиции, первая программа на новом языке должна выводить на экран словосочетание «Hello, world». На Swift это пишется всего в одну строку:

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

Если вы писали до этого код на C или Objective-C, то этот синтаксис должен быть вам знаком – на Swift, эта строка является законченной программой. Не нужно больше импортировать отдельные библиотеки для таких функций, как ввод/вывод или обработка строк. Код, написанный в глобальной области, используется как входная точка для программы, так что функция `main()` больше не нужна. Также вам не нужно писать точки с запятой после каждой строки.

Наглядно показывая решение типичных задач, это введение даст вам достаточно информации для того, чтобы начать писать код на языке Swift. Не волнуйтесь если что-то покажется непонятным – все, что показано в этом введении, детально поясняется в течение всей книги.

Простые типы данных

Используйте `let` для создания констант и `var` для объявления переменных. Значение константы не обязательно должно быть известно на момент компиляции, но оно должно присваиваться *строго* один раз. Это значит, что вы можете использовать константу для обозначения значения, определяемого единожды, но используемого во многих местах.

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

Константа или переменная должны иметь те же типы данных, которые вы хотите им присвоить. Хотя, вы не должны всегда явно объявлять тип. Когда вы присваиваете значение при создании константы или переменной, компилятор логически предугадывает его тип. В примере выше, компилятор предугадал, что значение `myVariable` `integer` (целое число), потому что присвоенное ему значение - `integer`.

Если присвоенное значение не дает достаточной информации (или когда значение еще не присвоено), укажите тип, написав его после названия, разделенной с помощью двоеточия.

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

Задание

Создайте константу с явным типом `Float` (число с плавающей точкой) и значением 4.

Значения никогда не должны неявно конвертироваться в другой тип. Если вам нужно конвертировать значение в другой тип, тогда явно создайте экземпляр класса нужного типа.

```
let label = "The width is "  
let width = 94  
let widthLabel = label + String(width)
```

Задание

Создайте константу с явным типом Float (число с плавающей точкой) и значением 4.

Значения никогда не должны неявно конвертироваться в другой тип. Если вам нужно конвертировать значение в другой тип, тогда явно создайте экземпляр класса нужного типа.

```
let label = "The width is "  
let width = 94  
let widthLabel = label + String(width)
```

Задание

Попробуйте удалить конвертирование в String из последней строки. Какую ошибку вы получите?

Есть еще один простой способ поместить переменные в строку. Запишите переменную в скобках, и поставьте перед скобками обратный слэш (\), как показано ниже.

```
let apples = 3  
let oranges = 5  
let appleSummary = "I have \(apples) apples."  
let fruitSummary = "I have \(apples + oranges)  
pieces of fruit."
```

Задание

Используйте `\` чтобы добавить выражение вычисления числа с плавающей точкой в строку. Также, попробуйте вставить чье-нибудь имя в приветствие.

Массивы и словари создаются с помощью квадратных скобок `[]`, а получить доступ к их значениям можно указав индекс или ключ в квадратных скобках. Ставить запятую после последнего элемента разрешается.

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

Чтобы создать пустой массив или словарь, используйте выражение инициализации.

```
let emptyArray = [String]()  
let emptyDictionary = [String: Float]()
```

Если информация о типе переменной или константы должна быть предугадана, то вы можете написать пустой массив через `[]` и пустой словарь через `[:]` — например, когда вы присваиваете новое значение переменной или назначаете аргумент функции.

```
shoppingList = []  
occupations = [:]
```


Управление потоком

Для создания условий используйте `if` и `switch`, а для создания циклов используйте `for-in`, `for,while`, и `repeat-while`. Скобки вокруг условий или циклов не обязательны. Фигурные скобки вокруг тела условия или цикла — обязательны.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
```

В операторе `if`, условие должно быть Boolean выражение — это означает, что такой код как `if score {...}` ошибочный, никакого неявного сравнения с нулем не будет.

Вы можете использовать `if` и `let` вместе, чтобы работать со значениями, которые могут отсутствовать. Эти значения представлены как опциональные. Опциональные значения либо содержат значение, либо содержат `nil`, который обозначает отсутствие значения. Можно писать вопросительный знак (?) после типа значения, чтобы обозначить что оно опциональное.

```
var optionalString: String? = "Hello"
print(optionalString == nil)
```

```
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

Задание

Поменяйте `optionalName` на `nil`. Какое приветствие вы получите? Добавьте `else` условие, которое установит другое приветствие, если `optionalName` равно `nil`.

Если опциональное значение `nil`, то условие считается ложным и код в фигурных скобках пропускается. Иначе, опциональное значение извлекается и назначается константе написанной после `let`, что делает извлеченное значение доступным внутри блока кода.

Switch поддерживает любые данные и множество операторов сравнения — они не ограничены целыми числами и сравнениями.

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some
raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make
a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy
\(x)?"
default:
    let vegetableComment = "Everything
tastes good in soup."
}
```

Задание

Попробуйте удалить блок `default`. Какую ошибку вы получите?

Вы обратили внимание как мы можем использовать `let` в примере, назначая ее значению, которое удовлетворяет условию?

После исполнения кода внутри `switch` блока `case`, который попал под условие, программа выходит из оператора `switch`. Исполнение не продолжается к следующему блоку `case` - это означает, что не нужно прерывать `switch` после каждого блока `case` с помощью оператора `break`.

Вы можете использовать `for-in`, чтобы выполнить итерацию по элементам словаря, указывая пару имен для каждой пары ключ-значение.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
```

Задание

Добавьте другую переменную, чтобы посмотреть какое число самое большое, а также, из какой последовательности будет это число.

Используйте **while** чтобы повторять код, пока условие не поменяется. Условие цикла может также быть в конце, если нужно чтобы цикл выполнялся хотя-бы один раз.

```
var n = 2
while n < 100 {
    n = n * 2
}
print(n)
var m = 2
repeat {
    m = m * 2
} while m < 100
print(m)
```

Вы можете ставить индекс в цикле двумя способами: либо, используя оператор **..**<**** чтобы создать диапазон индексов, либо явно написать объявление, условие, и инкремент. Эти два цикла делают то же самое:

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
print(firstForLoop)

var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
print(secondForLoop)
```

Используйте `..<` чтобы создать диапазон который будет пропускать последнее значение, либо используйте `...` – диапазон который включает оба значения, и начальное и конечное.

Функции и замыкания

Чтобы объявить функцию используйте оператор `func`. Чтобы вызвать функцию просто напишите его имя со списком аргументов в скобках. Используйте `->` чтобы разделить имена и типы аргументов от возвращаемого типа функции.

```
func greet(name: String, day: String) ->
String {
    return "Hello \(name), today is
\ (day) ."
}
greet("Bob", day: "Tuesday")
```

Задание

Удалите аргумент `day`. Добавьте аргумент, чтобы вставить сегодняшнее блюдо дня в приветствие.

Используйте кортеж для создания единого составного значения, например для возвращения нескольких значений из функции. Ссылаться на элементы кортежа можно либо через имена, либо через порядковые номера

```

func calculateStatistics(scores: [Int]) ->
(min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0
    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }
    return (min, max, sum)
}

```

```

let statistics = calculateStatistics([5, 3,
100, 3, 9])
print(statistics.sum)
print(statistics.2

```

Функция также может принимать переменное количество аргументов, собирая их в массив:

```

func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}

```

```

sumOf()
sumOf(42, 597, 12)

```

Задание

Напишите функцию, которая бы считала среднее значение его аргументов.

Функции могут быть вложенными. Вложенные функции имеют доступ к переменным, которые были объявлены во внешней функции. Вы можете использовать вложенные функции, чтобы упорядочивать код в длинных или сложных функциях.

```
func returnFifteen() -> Int {  
    var y = 10  
    func add() {  
        y += 5  
    }  
    add()  
    return y  
}
```

```
returnFifteen()
```

Функции - это тип первого класса. Это означает, что результатом функции может быть другая функция.

```
func makeIncrementer() -> (Int -> Int) {  
    func addOne(number: Int) -> Int {  
        return 1 + number  
    }  
    return addOne  
}
```

```
var increment = makeIncrementer()  
increment(7)
```

Функция может принимать другую функцию в качестве аргумента.

```

func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {
  for item in list {
    if condition(item) {
      return true
    }
  }

  return false
}

func lessThanTen(number: Int) -> Bool {
  return number < 10
}

var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, condition: lessThanTen)

```

Функции на самом деле - частный случай замыканий. Замыкания представляют из себя блок кода, который может быть вызван позже. Код внутри замыканий имеет доступ к таким объектам, как к переменные и функции, которые были созданы в тех же рамках, что и сами замыкания. Даже если замыкание находится и запускается в другом блоке, вы уже видели этот пример во вложенных функциях. Вы можете написать замыкание без имени, просто обозначив код фигурными скобками и круглыми скобками (`{ }`). Внутри скобок используйте `in` для разграничения аргументов и возвращаемого типа.

```

numbers.map({
  (number: Int) -> Int in
  let result = 3 * number
  return result
})

```


Задание

Перепишите замыкание так, чтобы оно вернуло ноль для всех нечетных чисел.

У вас есть несколько способов для того, чтобы написать замыкание более кратко. Когда тип замыкания точно известен, например - обратный вызов делегата (**callback**), вы можете пропустить тип его аргументов, тип возвращаемого значения, либо и то и другое. Одиночный оператор замыкания неявно возвращает значение своего единственного выражения.

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

Вы можете обращаться к аргументам по номеру, вместо имени. Этот подход особенно полезен в очень коротких замыканиях. Замыкание, переданное как последний аргумент функции, может появиться непосредственно после скобок.

```
let sortedNumbers = numbers.sort { $0 > $1 }
print(sortedNumbers)
```

Объекты и классы

Чтобы создать класс используйте оператор `class` и дальше имя класса. Объявление свойств класса пишется таким же способом, как и объявление константы или переменной, за исключением того, что они объявляются в пределах класса. Подобно этому, методы класса объявляются тем же способом что и функции.

```
class Shape {  
    var numberOfSides = 0  
    func simpleDescription() -> String {  
        return "A shape with  
    \ (numberOfSides) sides."  
    }  
}
```

Задание

Добавьте константное свойство класса, используя `let`. Также, добавьте другой метод, который принимает какой-нибудь параметр.

Экземпляры класса создаются с помощью добавления скобок после имени класса. Получить доступ к свойствам и методам класса можно через точку.

```
var shape = Shape()  
    shape.numberOfSides = 7  
var shapeDescription =  
shape.simpleDescription()
```

В этой версии класса `Shape` отсутствует кое-что важное, а именно инициализатор. Он нужен для того, чтобы подготовить класс, когда создается экземпляр класса. Для его создания используйте оператор `init`.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String
    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with
\ (numberOfSides) sides."
    }
}
```

Обратите внимание, как `self` используется, чтобы различать `name` - как свойство класса, и `name` - как аргумент инициализатора. Аргументы инициализатору передаются как вызов функции при создании экземпляра класса. Каждому свойству должно присваиваться значение: либо через объявление (как с `numberOfSides`), либо через инициализатор (как с `name`).

Используйте `deinit` для создания деинициализатора, если вам нужно выполнить некоторую очистку, прежде чем объект будет освобожден.

Подклассы разделяются от имени родительского класса двоеточием. Для классов нет необходимости каждый раз писать родительский класс, его можно включить по мере необходимости.

Методы подкласса, которые переопределяют методы родителя, отмечаются с помощью оператора `override`. При попытке переопределения без `override` компилятор выдаст ошибку. Компилятор также обнаруживает методы с `override`, которые на самом деле не переопределяют никакие методы родительского класса.

```
class Square: NamedShape
    var sideLength: Double
    init(sideLength: Double, name: String)
    {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }
    func area() -> Double {
        return sideLength * sideLength
    }
    override func simpleDescription() ->
String {
    return "A square with sides of length
\\(sideLength)."
}
}
let test = Square(sideLength: 5.2, name: "my
test square")
test.area()
test.simpleDescription()
```

В дополнение к простым хранящимся свойствам, свойства могут иметь также getter и setter.

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0
    init(sideLength: Double, name: String){
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() ->
String {
    return "An equilateral triangle with
sides of length \(sideLength)."
}
}

var triangle = EquilateralTriangle(sideLength:
3.1, name: "a triangle")
print(triangle.perimeter)
triangle.perimeter = 9.9
print(triangle.sideLength)
```

В setter для свойства `perimeter`, новое значение имеет неявное имя `newValue`. Вы можете явно назначить другое имя, указав его в скобках после `set`.

Обратите внимание, что инициализатор для `EquilateralTriangle` имеет три разных шага:

1. Устанавливает значение свойств объявляемых подклассом.
2. Вызывает инициализатор родителя.
3. Изменяет значение свойств, объявленных родителем. Любые дополнительные работы по начальной установке, которые используют методы, `getter`'ы или `setter`'ы могут быть включены в этом месте.

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

```

class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength =
newValue.sideLength
        }
    }

    var square: Square {
        willSet {
            triangle.sideLength =
newValue.sideLength
        }
    }

    init(size: Double, name: String) {
        square = Square(sideLength: size,
name: name)
        triangle = EquilateralTriangle(sideLength:
size, name: name)
    }
}

var triangleAndSquare =
TriangleAndSquare(size: 10, name: "another
test shape")
print(triangleAndSquare.square.sideLength)
print(triangleAndSquare.triangle.sideLength)
triangleAndSquare.square = Square(sideLength:
50, name: "larger square")
print(triangleAndSquare.triangle.sideLength)

```

Когда вы работаете с опциональными значениями, вы можете написать `?` перед такими операциями как: методы, свойства или индексаторы. Если значение перед `?` - `nil`, то все что после `?` игнорируется и значение всего выражения становится `nil`. В противном случае, опциональное значение

извлекается, и все что после `?` выполняется. В обоих случаях значение всего выражения является опциональным значением.

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

```
let sideLength = optionalSquare?.sideLength
```

Перечисление и структуры

Используйте оператор `enum` для создания перечислений. Подобно классам и другим именованным типам, перечисления могут иметь методы связанные с ним.

```
enum Rank: Int {  
    case Ace = 1  
    case Two, Three, Four, Five, Six,  
Seven, Eight, Nine, Ten  
    case Jack, Queen, King  
    func simpleDescription() -> String {  
        switch self {  
            case .Ace:  
                return "ace"  
            case .Jack:  
                return "jack"  
            case .Queen:  
                return "queen"  
            case .King:  
                return "king"
```



```

default:
    return String(self.rawValue)
    }
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue

```

Задание

Напишите функцию, сравнивающую два `Rank` значения, с помощью сравнения их исходных (`raw`) значений.

В приведенном выше примере, тип исходного значения перечисления является `Int`, а это значит, что вы можете указать только первое исходное значение. Остальные исходные значения присваиваются по порядку. Вы также можете использовать строки или числа с плавающей точкой в качестве типа исходного значения перечисления. Используйте свойство `rawValue` для доступа к исходному значению элемента перечисления.

Используйте инициализатор `init?(rawValue:)` для того, чтобы создать экземпляр перечисления из исходного значения:

```

if let convertedRank = Rank(rawValue: 3) {
    let threeDescription =
convertedRank.simpleDescription()
}

```

Значения элементов перечисления - это реальные значения, а не просто еще один способ написания исходных значений. Фактически, в тех случаях, когда нет ясного представления об исходном значении, вы можете их не писать.

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}

let hearts = Suit.Hearts
let heartsDescription =
hearts.simpleDescription()
```

Задание

Добавьте метод `color` для `Suit`, который возвращает "black" для spades (пики) и `clubs` (трефы), и возвращает "red" для hearts (червы) и diamonds (бубны).

Обратите внимание на два способа обращения к элементам перечисления `Hearts` выше. Во время присвоения значения константе `hearts`, обращение к элементу перечисления `Suit.Hearts` происходит через его полное имя, потому что константа не имеет явно указанного типа. Внутри `switch`, обращение к перечислению происходит через сокращённую форму `.Hearts`, потому что значение `self` уже

известно, и оно `Suit`. Вы можете использовать сокращенную форму каждый раз, когда тип значения известен.

Используйте оператор `struct` для создания структур. Структуры поддерживают многие характерные черты классов, в том числе методы и инициализаторы. Одно из наиболее важных различий между классами и структурами в том, что структуры всегда копируются когда они передаются в коде, а классы, передаются по ссылке.

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The
\\(rank.simpleDescription()) of
\\(suit.simpleDescription()) "
    }
}

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

Задание

Добавьте метод для `Card`, который бы создал полную колоду карт, с одной картой из каждой комбинации ранга (`rank`) и масти (`suit`).

Экземпляр элемента перечисления может иметь значения, связанные с экземпляром. Экземпляры одинаковых

элементов перечисления могут иметь разные значения, связанные с ним. Вы указываете связанные значения, когда создаете экземпляр. Связанные значения и исходные значения - это разные понятия. Исходное значение элемента перечисления одно и то же для всех экземпляров, вы указываете исходное значение, когда объявляете перечисление.

Например, рассмотрим случай с запросом времени восхода и заката солнца у сервера. Сервер, либо возвращает информацию, либо возвращает ошибку.

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}

let success = ServerResponse.Result("6:00 am",
    "8:09 pm")
let failure = ServerResponse.Error("Out of
cheese.")

switch success {
case let .Result(sunrise, sunset):
    let serverResponse = "Sunrise is at
\(sunrise) and sunset is at \(sunset)."
case let .Error(error):
    let serverResponse = "Failure...
\(error)"
}
```

Задание

Добавьте третий случай case в `ServerResponse` и в `switch`.

Обратите внимание, как время восхода (`sunrise`) и заката (`sunset`) извлекаются из значения `ServerResponse`, так же как значение для части поиска совпадений в блоках `case` у `switch`.

Протоколы и расширения

Используйте оператор `protocol` для объявления протокола.

```
1. protocol ExampleProtocol {
2.   var simpleDescription: String { get }
3.   mutating func adjust()
4. }
```

Классы, перечисления, и структуры могут соответствовать протоколам.

```
1. class SimpleClass: ExampleProtocol {
2.   var simpleDescription: String = "A very simple
   class."
3.   var anotherProperty: Int = 69105
4.   func adjust() {
5.     simpleDescription += " Now 100% adjusted."
6.   }
7. }
8. var a = SimpleClass()
9. a.adjust()
10.     let aDescription = a.simpleDescription
11.
12.     struct SimpleStructure: ExampleProtocol {
```

```

13.         var simpleDescription: String = "A simple
    structure"
14.         mutating func adjust() {
15.             simpleDescription += " (adjusted)"
16.         }
17.     }
18.     var b = SimpleStructure()
19.     b.adjust()
20.     let bDescription = b.simpleDescription

```

Задание

Напишите перечисление, которое будет соответствовать этому протоколу.

Обратите внимание на использование ключевого слова `mutating` в объявлении `SimpleStructure`, оно обозначает метод который модифицирует структуру. Объявление `SimpleClass` не нуждается в `mutating` методах, так как методы класса всегда могут модифицировать класс.

Используйте оператор `extension` (расширение) для того чтобы добавить новый функционал для существующего типа, такой как объявление новых методов и вычисляемых свойств. Вы можете использовать расширение для добавления совместимости с протоколом типу, который объявлен в другом месте, или даже типу, который вы импортировали из библиотеки или фреймворка:

```

1. extension Int: ExampleProtocol {
2.     var simpleDescription: String {
3.         return "The number \(self)"
4.     }
5.     mutating func adjust() {
6.         self += 42
7.     }
8. }
9. 7.simpleDescription

```

Задание

Напишите расширение для типа `Double`, которое добавляет свойство `absoluteValue`.

Вы можете использовать имя протокола точно так же как другие именованные типы - например, чтобы создать коллекцию объектов, которые имеют разные типы, но все соответствуют одному протоколу. Когда вы работаете со значениями, чей тип - протокол, методы за пределами объявления протокола не доступны.

1. `let protocolValue: ExampleProtocol = a`
2. `protocolValue.simpleDescription`
3. `// protocolValue.anotherProperty` // Разкомментируйте чтобы увидит ошибку

Несмотря на то, что переменная `protocolValue` имеет исполняемый тип `SimpleClass`, компилятор обрабатывает его тип как присвоенный ему `ExampleProtocol`. Это означает, что вы не сможете случайно получить доступ к методам или свойствам, которые класс реализует в дополнение к протоколу соответствия.

Универсальные типы

Напишите имя внутри угловых скобок, чтобы создать универсальную (**generic**) функцию или тип.

```
1. func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {
2.     var result = ItemType[]()
3.     for i in 0..times {
4.         result += item
5.     }
6.     return result
7. }
8. repeat("knock", 4)
```

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

```
1. // Reimplement the Swift standard library's optional type
2. enum OptionalValue<T> {
3.     case None
4.     case Some(T)
5. }
6. var possibleInteger: OptionalValue<Int> = .None
7. possibleInteger = .Some(100)
```

Используйте **where** после названия типа, чтобы указать список требований, например - потребовать, чтобы тип реализовал протокол, потребовать, чтобы два типа были одинаковы, или потребовать, чтобы класс имел определенный суперкласс.

```
1. func anyCommonElements <T, U where T: Sequence, U: Sequence, T.IteratorType.Element: Equatable, T.IteratorType.Element == U.IteratorType.Element>
   (lhs: T, rhs: U) -> Bool {
```



```
2.  for lhsItem in lhs {
3.      for rhsItem in rhs {
4.          if lhsItem == rhsItem {
5.              return true
6.          }
7.      }
8.  }
9.  return false
10. }
11.     anyCommonElements([1, 2, 3], [3])
```

Задание

Модифицируйте функцию `anyCommonElements`, чтобы сделать функцию, которая возвращает массив элементов, которые общие для обеих последовательностей.

В простых случаях, вы можете пропустить `where` и просто написать протокол или имя класса после двоеточия. Запись `<T: Equatable>` то же самое, что и `<T where T: Equatable>`.

ОСНОВЫ

Swift - новый язык программирования для разработки приложений под iOS и OS X. Несмотря на это, многие части Swift могут быть вам знакомы из вашего опыта разработки на C и Objective-C.

Swift предоставляет свои собственные версии фундаментальных типов C и Objective-C, включая `Int` для целых чисел, `Double` и `Float` для значений с плавающей точкой, `Bool` для булевых значений, `String` для текста. Swift также предоставляет мощные версии двух основных типов коллекций, как описано в разделе Типы коллекций.

Подобно C, Swift использует переменные для хранения и обращения к значениям по уникальному имени. Swift также широко использует переменные, значения которых не могут быть изменены. Они известны как константы, и являются гораздо более мощными, чем константы в C. Константы используются в Swift повсеместно, чтобы сделать код безопаснее и чище в случаях, когда вы работаете со значениями, которые не должны меняться.

В дополнение к знакомым типам, Swift включает расширенные типы, которых нет в Objective-C. К ним относятся кортежи, которые позволяют создавать и передавать группы значений. Кортежи, могут возвращать несколько значений из функции как одно целое значение.

Swift также включает опциональные типы, которые позволяют работать с отсутствующими значениям. Опциональные значения говорят либо «здесь есть значение, и оно равно x», либо «здесь нет значения вообще». Опциональные типы подобны использованию `nil` с указателями в Objective-C, но они работают со всеми типами, не только с классами.

Опциональные значения безопаснее и выразительнее чем `nil` указатели в Objective-C, и находятся в сердце многих наиболее мощных особенностей Swift.

Опциональные значения - это пример того факта, что Swift - язык безопасных типов (type safe). Swift помогает понять, с какими типами значений ваш код может работать. Если кусок вашего кода ожидает `String`, безопасность типов не даст вам передать ему `Int` по ошибке. Это позволяет вам улавливать и исправлять ошибки как можно раньше в процессе разработки.

Константы и переменные

Константы и переменные связывают имя (например, `maximumNumberOfLoginAttempts` или `welcomeMessage`) со значением определенного типа (например, число `10` или строка `"Hello"`). Значение константы не может быть изменено после его установки, тогда как переменной может быть установлено другое значение в будущем.

Объявление констант и переменных

Константы и переменные должны быть объявлены, перед тем как их использовать. Константы объявляются с помощью ключевого слова `let`, а переменные с помощью `var`. Вот пример того, как константы и переменные могут быть использованы для отслеживания количества попыток входа, которые совершил пользователь:

```
1. let maximumNumberOfLoginAttempts = 10
2. var currentLoginAttempt = 0
```

Этот код можно прочесть как:

«Объяви новую константу с именем `maximumNumberOfLoginAttempts`, и задай ей значение `10`. Потом, объяви новую переменную с именем `currentLoginAttempt`, и задай ей начальное значение `0`.»

В этом примере максимальное количество доступных попыток входа объявлено как константа, потому что максимальное значение никогда не меняется. Счетчик текущего количества попыток входа объявлен как переменная, потому что это значение должно увеличиваться после каждой неудачной попытки входа.

Вы можете объявить несколько констант или несколько переменных на одной строке, разделяя их запятыми:

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

Замечание

Если хранимое значение в вашем коде не будет меняться, всегда объявляйте его как константу, используя ключевое слово `let`. Используйте переменные только для хранения значений, которые должны будут меняться.

Обозначение типов

Вы можете добавить обозначение типа, когда объявляете константу или переменную, чтобы иметь четкое представление о типах значений, которые могут хранить константы или переменные. Написать обозначение типа, можно поместив двоеточие после имени константы или

переменной, затем пробел, за которым следует название используемого типа.

Этот пример добавляет обозначение типа для переменной с именем `welcomeMessage`, чтобы обозначить, что переменная может хранить `String`:

```
var welcomeMessage: String
```

Код выше может быть прочитан как:

«Объяви переменную с именем `welcomeMessage`, тип которой будет `String`»

Фраза «тип которой будет `String`» означает «может хранить любое `String` значение». Представьте, что словосочетание «тип которой будет такой-то» означает - значение, которое будет храниться.

Теперь переменной `welcomeMessage` можно присвоить любое текстовое значение, без каких либо ошибок:

```
welcomeMessage = "Hello"
```

Замечание

Редко когда вам понадобится обозначать тип на практике. Когда вы даете начальное значение константе или переменной на момент объявления, Swift всегда может вывести тип, который будет использовать в константе или переменной. Это описано в Строгая типизация и Вывод типов. В примере `welcomeMessage` выше, не было присвоения начального значения, так что тип переменной `welcomeMessage` указывается с помощью

обозначения типа вместо того, чтобы вывести из начального значения.

Название констант и переменных

Вы можете использовать почти любые символы для названий констант и переменных, включая Unicode-символы:

```
1. let π = 3.14159
2. let 你好 = "你好世界"
3. let     = "dogcow"
```

Имена констант и переменных не могут содержать математических символов, стрелок, Unicode-символов из области приватных символов (или недопустимых символов), или символов для рисования линий и прямоугольников. Также они не могут начинаться с цифры, хотя цифры могут быть использованы в любом другом месте внутри имени.

После того, как вы объявили константу или переменную определенного типа, вы не можете снова объявить ее с тем же именем, или поменять имя, для хранения значения другого типа. Также нельзя менять константу в переменную или переменную в константу.

Замечание

Если переменной или константе необходимо назначить имя такое же, как у зарезервированного ключевого слова в Swift, вы можете сделать это, окружив ключевое слово обратными одинарными кавычками (```), и дальше использовать как имя. Но, тем не менее, вы должны избегать использования ключевых слов как имен, до тех пор, пока у вас не будет другого выбора.

Вы можете поменять значение существующей переменной на другое значение совместимого типа. В этом примере значение `friendlyWelcome` изменяется с `"Hello!"` на `"Bonjour!"`:

```
1. var friendlyWelcome = "Hello!"
2. friendlyWelcome = "Bonjour!"
3. // friendlyWelcome теперь "Bonjour!"
```

В отличие от переменных, значение константы не может быть изменено после его установки. Попытка сделать это выведет ошибку вовремя компиляции:

```
1. let languageName = "Swift"
2. languageName = "Swift++"
3. // это ошибка компиляции - languageName не может
   быть изменен
```

Вывод на экран констант и переменных

Вы можете напечатать на экране текущее значение константы или переменной с помощью функции `println`:

```
1. println(friendlyWelcome)
2. // напечатает "Bonjour!"
```

`println` - глобальная функция, которая печатает значение в назначенный вывод, с последующим разрывом строки. Например, если вы работаете в Xcode, `println` печатает его результат в панели «консоль» в Xcode. (Вторая функция, `print`, выполняет те же действия, только без добавления разрыва строки в конце).

Функция `println` печатает любое `String` значение, которое вы ему передадите:

```
1. println("This is a string")
2. // напечатает "This is a string"
```

Функция `println` может печатать более сложные сообщения, также как это делает функция `NSLog` из Сосоа. Эти сообщения могут включать значения констант и переменных.

Swift использует *интерполяцию строк* , для возможности включить имя константы или переменной как указатель в длинных строках, и запросить Swift поменять его с текущим значением константы или переменной. Заключите имя в скобки и экранируйте его с помощью обратного слеша перед открывающей скобкой.

```
1. println("The current value of friendlyWelcome
   is \(friendlyWelcome)")
2. // напечатает "The current value of friendlyWelcome
   is Bonjour!"
```

Замечание

Все возможности, которые вы можете использовать с интерполяцией строк, описаны в разделе Интерполяция строк

Комментарии

Используйте комментарии, чтобы добавить неисполняемый текст в коде, как примечание или напоминание самому себе. Комментарии игнорируются компилятором Swift во время компиляции кода.

Комментарии в Swift очень похожи на комментарии в C. Однострочные комментарии начинаются с двух слешей (//):

```
// это комментарий
```

Вы также можете написать многострочные комментарии, которые начинаются со слеша и звездочки (/*) и заканчиваются звездочкой, за которой следует слеш (*):

1. /* это тоже комментарий,
2. но написанный на двух строках */

В отличие от многострочных комментариев в C, многострочные комментарии в Swift могут быть вложены в другие многострочные комментарии. Вы можете написать вложенные комментарии, начав многострочный блок комментариев, а затем, начать второй многострочный комментарий внутри первого блока. Затем второй блок закрывается, а за ним закрывается первый блок:

1. /* это начало первого многострочного комментария
2. /* это второго, вложенного многострочного комментария */
3. это конец первого многострочного комментария */

Вложенные многострочные комментарии позволяют закомментировать большие блоки кода быстро и легко, даже если код уже содержит многострочные комментарии.

Точка с запятой

В отличие от многих других языков, Swift не требует писать точку с запятой (;) после каждого выражения в коде, хотя вы можете делать это, если хотите. Однако точки с запятой требуются, если вы хотите написать несколько отдельных выражений на одной строке:

1. `let cat = "(смайл кота)"; println(cat)`
2. `// напечатает " "`

Целые числа

Integer (целое число) - это число, не содержащее дробной части, например как 42 и -23. Целые числа могут быть либо знаковыми (положительными, ноль или отрицательными) либо беззнаковыми (положительными или ноль).

Swift предусматривает знаковые и беззнаковые целые числа в 8, 16, 32 и 64 битном форматах. Эти целые числа придерживаются соглашения об именах аналогичных именам в C, в том, что 8-разрядное беззнаковое целое число имеет тип `UInt8`, а 32-разрядное целое число имеет тип `Int32`. Как и все типы в Swift, эти типы целых чисел пишутся с заглавной буквой.

Границы целых чисел

Вы можете получить доступ к минимальному и максимальному значению каждого типа целого числа с помощью его свойств `min` и `max`:

1. `let minValue = UInt8.min` // `minValue` равен 0, а его тип `UInt8`
2. `let maxValue = UInt8.max` // `maxValue` равен to 255, а его тип `UInt8`

Тип значения этих свойств соответствует размеру числа (в примере выше этот тип `UInt8`) и поэтому может быть использован в выражениях наряду с другими значениями того же типа.

Int

В большинстве случаев вам не нужно будет указывать конкретный размер целого числа для использования в коде. В Swift есть дополнительный тип целого числа - `Int`, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, `Int` того же размера что и `Int32`
- На 64-битной платформе, `Int` того же размера что и `Int64`

Если вам не нужно работать с конкретным размером целого числа, всегда используйте в своем коде `Int` для целых чисел. Это придает коду логичности и совместимости. Даже на 32-битных платформах, `Int` может хранить любое значение в пределах `-2,147,483,648` и `2,147,483,647`, а этого достаточно для многих диапазонов целых чисел.

UInt

Swift также предусматривает беззнаковый тип целого числа - `UInt`, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, `UInt` того же размера что и `UInt32`
- На 64-битной платформе, `UInt` того же размера что и `UInt64`

Заметка

Используйте `UInt`, только когда вам действительно нужен тип беззнакового целого с размером таким же, как разрядность системы. Если это не так, использовать `Int` предпочтительнее, даже когда известно, что значения будут неотрицательными. Постоянное использование `Int` для целых чисел способствует совместимости кода, позволяет избежать преобразования между разными типами чисел, и соответствует выводу типа целого числа, как описано в [Строгая типизация и Вывод Типов](#).

Числа с плавающей запятой

Число с плавающей точкой - это число с дробной частью, например как 3.14159, 0.1, и -273.15.

Типы с плавающей точкой могут представлять гораздо более широкий спектр значений, чем типы целых значений, и могут хранить числа намного больше (или меньше) чем может хранить `Int`. Swift предоставляет два знаковых типа с плавающей точкой:

- **Double** - представляет собой 64-битное число с плавающей точкой. Используйте его когда число с плавающей точкой должно быть очень большим или чрезвычайно точным
- **Float** - представляет собой 32-битное число с плавающей точкой. Используйте его, когда значение не нуждается в 64-битной точности.

Заметка

Double имеет точность минимум 15 десятичных цифр, в то время как точность **Float** может быть всего лишь 6 десятичных цифр. Соответствующий тип числа с плавающей точкой используется в зависимости от характера и диапазона значений, с которыми вы должны работать в коде.

Строгая типизация и

Вывод типов

Swift - язык со *строгой типизацией*. Язык со строгой типизацией призывает вас иметь четкое представление о типах значений с которыми может работать ваш код. Если часть вашего кода ожидает `String`, вы не сможете передать ему `Int` по ошибке.

Поскольку Swift имеет строгую типизацию, он выполняет проверку типов при компиляции кода и отмечает любые несоответствующие типы как ошибки. Это позволяет в процессе разработки ловить, и как можно раньше, исправлять ошибки.

Проверка типов поможет вам избежать ошибок при работе с различными типами значений. Тем не менее, это не означает, что при объявлении вы должны указывать тип каждой константы или переменной. Если вы не укажете нужному вам значению тип, то Swift будет использовать вывод типов, чтобы вычислить соответствующий тип. Вывод типов позволяет компилятору вывести тип конкретного выражения автоматически во время компиляции, просто путем изучения значения, которого вы ему передаете.

Благодаря выводу типов, Swift требует гораздо меньше объявления типов, чем языки, такие как C или Objective-C. Константам и переменным все же нужно присваивать тип, но большая часть работы с указанием типов будет сделана за вас.

Вывод типов особенно полезен, когда вы объявляете константу или переменную с начальным значением. Часто это делается путем присвоения литерального значения (или

литерала) к константам или переменным в момент объявления. (Литеральное значение - значение, которое появляется непосредственно в исходном коде, например как 42 и 3, 14159 в примерах ниже.)

Например, если вы присваиваете литеральное значение 42 к новой константе не сказав какого она типа, Swift делает вывод, что вы хотите чтобы константа была `Int`, потому что вы присвоили ей значение, которое похоже на целое число:

```
1. let meaningOfLife = 42
2. // meaningOfLife выводится как тип Int
```

Точно так же, если вы не указали тип для литерала с плавающей точкой, Swift делает вывод, что вы хотите создать `Double`:

```
1. let pi = 3.14159
2. // pi выводится как тип Double
```

Swift всегда выбирает `Double` (вместо `Float`), когда выводит тип чисел с плавающей точкой.

Если объединить целые литералы и литералы с плавающей точкой в одном выражении, в этом случае тип будет выводиться как `Double`:

```
1. let anotherPi = 3 + 0.14159
2. // anotherPi тоже выводится как тип Double
```

Литеральное значение 3 не имеет явного типа само по себе, так что соответствующий тип `Double` выводится из наличия литерала с плавающей точкой как часть сложения.

Числовые литералы

Числовые литералы могут быть написаны как:

- Десятичное число, без префикса
- Двоичное число, с префиксом `0b`
- Восьмеричное число, с префиксом `0o`
- Шестнадцатеричное число, с префиксом `0x`

Все эти литералы целого числа имеют десятичное значение 17:

```
1. let decimalInteger = 17
2. let binaryInteger = 0b10001 // 17 в двоичной
   нотации
3. let octalInteger = 0o21 // 17 в восьмеричной нотации
4. let hexadecimalInteger = 0x11 // 17 в
   шестнадцатеричной нотации
```

Литералы с плавающей точкой могут быть десятичными (без префикса) или шестнадцатеричными (с префиксом `0x`). Они всегда должны иметь число (десятичное или шестнадцатеричное) по обе стороны от дробной точки. Они также могут иметь экспоненту, с указанием в верхнем или нижнем регистре `e` для десятичных чисел с плавающей точкой, или в верхнем или нижнем регистре `p` для шестнадцатеричных чисел с плавающей точкой.

Для десятичных чисел с показателем степени `exp`, базовое число умножается на 10^{exp} :

- `1.25e2` означает 1.25×10^2 , или `125.0`.
- `1.25e-2` означает 1.25×10^{-2} , или `0.0125`.

Для шестнадцатеричных чисел с показателем степени `exp`, базовое число умножается на 2^{exp} :

- `0xFp2` означает 15×2^2 , или `60.0`.
- `0xFp-2` означает 15×2^{-2} , или `3.75`.

Все эти числа с плавающей точкой имеют десятичное значение `12.1875`:

```
1. let decimalDouble = 12.1875
2. let exponentDouble = 1.21875e1
3. let hexadecimalDouble = 0xC.3p0
```

Числовые литералы могут содержать дополнительное форматирование, чтобы их было удобнее читать. Целые числа и числа с плавающей точкой могут быть дополнены нулями и могут содержать символы подчеркивания для увеличения читабельности. Ни один тип форматирования не влияет на базовое значение литерала:

```
1. let paddedDouble = 000123.456
2. let oneMillion = 1_000_000
3. let justOverOneMillion = 1_000_000.000_000_1
```

Преобразование числовых типов

Используйте `Int` для всех целочисленных констант и переменных в коде, даже когда, они отрицательны. Использование стандартного типа целых чисел в большинстве случаев означает, что ваши целочисленные константы и переменные будут совместимы в коде и будут

соответствовать типу, выведенному из целочисленного литерала.

Используйте другие типы целых чисел, только если вам это действительно нужно, например, когда используются данные с заданным размером из внешнего источника, или для производительности, использования памяти или других важных оптимизаций. Использование типов с определенным размером в таких ситуациях помогает уловить случайное переполнение значения и неявно задокументированные данные, используемые в коде.

Преобразования целых чисел

Диапазон значений, который может храниться в целочисленных константах и переменных, различен для каждого числового типа. `Int8` константы и переменные могут хранить значения между `-128` и `127`, тогда как `UInt8` константы и переменные могут хранить числа между `0` и `255`. Если число не подходит для переменной или константы с определенным размером, выводится ошибка во время компиляции:

```
1. let cannotBeNegative: UInt8 = -1
2. // UInt8 не может хранить отрицательные значения,
   // поэтому эта строка выведет ошибку
3. let tooBig: Int8 = Int8.max + 1
4. // Int8 не может хранить число больше своего
   // максимального значения,
5. // так что это тоже выведет ошибку
```

Поскольку каждый числовой тип может хранить разный диапазон значений, в зависимости от конкретного случая вам придется обращаться к преобразованию числовых типов. Этот подход предотвращает скрытые ошибки преобразования и помогает сделать причину преобразования понятной. Чтобы

преобразовать один числовой тип в другой, необходимо создать новое число желаемого типа из существующего значения. Ниже, в примере, константа `twoThousand` имеет тип `UInt16`, тогда как константа `one` - `UInt8`. Сложить их напрямую не получится, потому что они разного типа. Вместо этого, в примере вызывается функция `UInt16(one)` для создания нового числа `UInt16` из значения константы `one`:

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

Теперь, из-за того, что обе части сложения имеют тип `UInt16` - операция сложения допустима. Для конечной константы (`twoThousandAndOne`) выведен тип `UInt16`, потому что это сложение двух `UInt16` значений. **НазваниеТипа (начальноеЗначение)** - стандартный способ вызвать инициализатор типов Swift и передать начальное значение. Честно говоря, у `UInt16` есть инициализатор, который принимает `UInt8` значение, и, таким образом, этот инициализатор используется, чтобы создать новый `UInt16` из существующего `UInt8`. Здесь вы не можете передать любой тип, однако это должен быть тип, для которого у `UInt16` есть инициализатор. Расширение существующих типов с помощью создания инициализаторов, которые принимают новые типы (включая объявление вашего типа) рассматривается в главе Расширения.

Преобразования целых чисел и чисел с плавающей точкой

Преобразование между целыми числами и числами с плавающей точкой должно происходить явно:

```
1. let three = 3
2. let pointOneFourOneFiveNine = 0.14159
3. let pi = Double(three) + pointOneFourOneFiveNine
```

```
4. // pi равно 3.14159, и для него выведен тип Double
```

Здесь, значение константы `three` используется для создания нового значения типа `Double`, так что обе части сложения имеют один тип. Без этого преобразования сложение не будет проходить. Обратное преобразование числа с плавающей точкой в целое число тоже должно происходить явно. Так что тип целого числа может быть инициализирован с помощью `Double` и `Float` значений

```
1. let integerPi = Int(pi)
2. // integerPi равен 3, и для него выведен тип Int
```

Числа с плавающей точкой всегда урезаются когда вы используете инициализацию целого числа через этот способ. Это означает, что `4.75` будет `4`, а `-3.9` будет `-3`.

Заметка

Правила объединения числовых констант и переменных отличается от правил числовых литералов. Литеральное значение `3` может напрямую сложиться с литеральным значением `0.14159`, потому что числовые литералы сами по себе не имеют явного типа. Их значение выводится только когда выполняется компилятором.

Псевдонимы типов

Псевдонимы типов задают альтернативное имя для существующего типа. Можно задать псевдоним типа с помощью ключевого слова `typealias`.

Псевдонимы типов полезны, когда вы хотите обратиться к существующему типу по имени, который больше подходит по контексту, например, когда вы работаете с данными определенного размера из внешнего источника:

```
typealias AudioSample = UInt16
```

После того как вы один раз задали псевдоним типа, вы можете использовать псевдоним везде, где вы хотели бы его использовать

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

Здесь `AudioSample` определен как псевдоним для `UInt16`. Поскольку это псевдоним, вызов `AudioSample.min` фактически вызовет `UInt16.min`, что показывает начальное значение 0 для переменной `maxAmplitudeFound`.

Логические типы

В Swift есть простой *логический* тип `Bool`. Этот тип называют логическим, потому что он может быть только `true` или `false`. Swift предусматривает две логические константы, `true` и `false` соответственно:

```
1. let orangesAreOrange = true
2. let turnipsAreDelicious = false
```

Типы для `orangesAreOrange` и `turnipsAreDelicious` были выведены как `Bool`, исходя из того факта, что мы им присвоили логические литералы. Так же как с `Int` и `Double` в предыдущих главах, вам не нужно указывать константы или переменные как `Bool`, если при создании вы присвоили им значения `true` или `false`. Вывод типов помогает сделать код Swift кратким и читабельным тогда, когда вы создаете константы или переменные со значениями которые точно известны.

Логические значения очень полезны когда вы работаете с условными операторами, такими как оператор `if`:

```
1. if turnipsAreDelicious {
2.     println("Mmm, tasty turnips!")
3. } else {
4.     println("Eww, turnips are horrible.")
5. }
6. // напечатает "Eww, turnips are horrible."
```

Условные операторы, такие как оператор `if` детально рассматриваются в главе [Управление потоком](#).

Строгая типизация Swift препятствует замене значения `Bool` на не логическое значение. Следующий пример выведет ошибку компиляции:

```
1. let i = 1
2. if i {
3.     // этот пример не скомпилируется, и выдаст ошибку
   компиляции
4. }
```

Тем не менее, альтернативный пример ниже правильный:

```
1. let i = 1
2. if i == 1 {
3.     // этот пример выполнится успешно
4. }
```

Результат сравнения `i == 1` имеет тип `Bool`, и поэтому этот второй пример совершает проверку типов. Такие сравнения как `i == 1` обсуждаются в главе [Базовые операторы](#).

Как в других примерах строгой типизации в Swift, этот подход предотвращает случайные ошибки и гарантирует, что замысел определенной части кода понятен.

Кортежи

Кортежи группируют несколько значений в одно составное значение. Значения внутри кортежа могут быть любого типа, то есть, нет необходимости, чтобы они были одного и того же типа.

В данном примере `(404, "Not Found")` это кортеж, который описывает *код HTTP статуса*. Код HTTP статуса — особое значение возвращаемое веб-сервером каждый раз, когда вы запрашиваете веб-страницу. Код статуса `404 Not Found` возвращается, когда вы запрашиваете страницу, которая не существует.

```
1. let http404Error = (404, "Not Found")
2. // http404Error имеет тип (Int, String), и равен
   (404, "Not Found")
```

Чтобы передать код статуса, кортеж `(404, "Not Found")` группирует вместе два отдельных значения `Int` и `String`: число и понятное человеку описание. Это может быть описано как "кортеж типа `(Int, String)`".

Вы можете создать кортеж с любой расстановкой типов, и они могут содержать сколько угодно нужных вам типов. Ничто вам не мешает иметь кортеж типа `(Int, Int, Int)`, или типа `(String, Bool)`, или же с любой другой расстановкой типов по вашему желанию.

Вы можете разложить содержимое кортежа на отдельные константы и переменные, к которым можно получить доступ привычным способом:


```
1. let (statusCode, statusMessage) = http404Error
2. println("The status code is \"(statusCode)\")
3. // напечатает "The status code is 404"
4. println("The status message is \"(statusMessage)\")
5. // напечатает "The status message is Not Found"
```

Если вам нужно только некоторые из значений кортежа, вы можете игнорировать части кортежа во время разложения с помощью символа подчеркивания ():

```
1. let (justTheStatusCode, _) = http404Error
2. println("The status code is \"(justTheStatusCode)\")
3. // напечатает "The status code is 404"
```

В качестве альтернативы можно получать доступ к отдельным частям кортежа, используя числовые индексы, начинающиеся с нуля:

```
1. println("The status code is \"(http404Error.0)\")
2. // напечатает "The status code is 404"
3. println("The status message is \"(http404Error.1)\")
4. // напечатает "The status message is Not Found"
```

Вы можете давать имена отдельным элементам кортежа во время объявления:

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

Когда вы присвоили имя элементу кортежа, вы можете обратиться к нему по имени:

```
1. println("The status code
   is \(http200Status.statusCode) ")
2. // напечатает "The status code is 200"
3. println("The status message
   is \(http200Status.description) ")
4. // напечатает "The status message is OK"
```

Кортежи особенно полезны в качестве возвращаемых значений функций. Функция, которая пытается получить веб-страницу, может вернуть кортеж типа `(Int, String)`, чтобы описать успех или неудачу в поиске страницы.

Возвращая кортеж с двумя отдельными значениями разного типа, функция дает более полезную информацию о ее результате, чем, если бы, возвращала единственное значение одного типа, возвращаемое функцией. Для более подробной информации смотрите главу [Функции с несколькими возвращаемыми значениями](#).

Заметка

Кортежи полезны для временной группировки связанных значений. Они не подходят для создания сложных структур данных. Если ваша структура данных, вероятно, будет выходить за рамки временной структуры, то такие вещи лучше проектируйте с помощью классов или структур, вместо кортежей. Для получения дополнительной информации смотрите главу [Классы и структуры](#).

Опциональные типы

Опциональные типы используются в тех случаях, когда значение может отсутствовать. Опциональный тип подразумевает:

- Значение *существует*, и оно равно x

или

- Значение *не существует* вовсе.

Заметка

В C или Objective-C нет понятия опционалов. Ближайшее понятие в Objective-C это возможность вернуть `nil` из метода, который в противном случае вернул бы объект. В этом случае `nil` обозначает «отсутствие допустимого объекта». Тем не менее, это работает только для объектов, и не работает для структур, простых типов C, или значений перечисления. Для этих типов, методы Objective-C, как правило, возвращают специальное значение (например `NSNotFound`), чтобы указать отсутствие значения. Этот подход предполагает, что разработчик, который вызвал метод, знает, что есть это специальное значение и что его нужно учитывать. Опционалы Swift позволяют указать отсутствие значения для абсолютно любого типа, без необходимости использования специальных констант.

Приведем пример, который покажет, как опционалы могут справиться с отсутствием значения. Тип `String` в Swift имеет метод `toInt`, который пытается преобразовать `String` значение в `Int`. Тем не менее, не каждая строка может быть преобразована в целое число. Строка `"123"` может быть преобразована в числовое

значение `123`, но строка `"hello, world"` не имеет очевидного числового значения для преобразования.

В приведенном ниже примере используется метод `toInt` для попытки преобразовать `String` в `Int`:

```
let possibleNumber = "123"

let convertedNumber = Int(possibleNumber)

// для convertedNumber выведен тип "Int?", или
"опциональный Int"
```

Поскольку метод `toInt` может иметь недопустимый аргумент, он возвращает *опциональный Int*, вместо `Int`.

Опциональный `Int` записывается как `Int?`, а не `Int`. Знак вопроса означает, что содержащееся в ней значение является опциональным, что означает, что он может содержать *некое Int* значение, или он может вообще не содержать *никакого значения*. (Он не может содержать ничего другого, например `Bool` значение или значение `String`. Он либо `Int`, либо вообще ничто)

nil

Мы можем установить опциональную переменную в состояние отсутствия значения, путем присвоения ему специального значения `3`

```
1. var serverResponseCode: Int? = 404
2. // serverResponseCode содержит реальное Int
   значение 404
3. serverResponseCode = nil
4. // serverResponseCode теперь не содержит значения
```

Заметка

`nil` не может быть использован с не опциональными константами и переменными. Если значение константы или переменной при определенных условиях в коде должно когда-нибудь отсутствовать, всегда объявляйте их как опциональное значение соответствующего типа.

Если объявить опциональную переменную без присвоения значения по умолчанию, константа или переменная автоматически установятся в `nil` для вас:

1. `var surveyAnswer: String?`
2. `// surveyAnswer автоматически установится в nil`

Заметка

`nil` в Swift не то же самое что `nil` в Objective-C. В Objective-C `nil` является указателем на несуществующий объект. В Swift `nil` не является указателем, а является отсутствием значения определенного типа. Устанавливаться в `nil` могут опционалы любого типа, а не только типы объектов.

Оператор If и Принудительное извлечение

Вы можете использовать оператор `if`, сравнивая опционал с `nil`, чтобы проверить, содержит ли опционал значение. Это сравнение можно сделать с помощью оператора «равенства» (`==`) или оператора «неравенства» (`!=`).

Если опционал имеет значение, он будет рассматриваться как «неравным» `nil`:

1. `if convertedNumber != nil {`
2. `println("convertedNumber contains some integer value.")`
3. `}`

```
4. // напечатает "convertedNumber contains some
integer value."
```

Если вы уверены, что опционал содержит значение, вы можете получить доступ к его значению, добавив восклицательный знак (!) в конце имени опционала.

Восклицательный знак фактически говорит: «Я знаю точно, что этот опционал содержит значение, пожалуйста, используйте его». Это выражение известно как *Принудительное извлечение* значения опционала:

```
1. if convertedNumber != nil {
2.   println("convertedNumber has an integer value
of \(convertedNumber!).")
3. }
4. // напечатает "convertedNumber has an integer value
of 123."
```

Более подробную информацию об операторе `if` можно получить в главе Управление потоком.

Заметка

Попытка использовать ! к несуществующему опциональному значению вызовет runtime ошибку. Всегда будьте уверены в том, что опционал содержит не-`nil` значение, перед тем как использовать ! чтобы принудительно извлечь это значение.

Привязка опционалов

Можно использовать *Привязку опционалов*, чтобы выяснить содержит ли опционал значение, и если да, то сделать это значение доступным в качестве временной константы или переменной. Привязка опционалов может использоваться с операторами `if` и `while`, для проверки значения внутри опционала, и извлечения этого значения в константу или переменную, в рамках одного действия.

Операторы `if` и `while` более подробно представлены в главе Управление потоком.

Привязку опционалов для оператора `if` можно писать как показано ниже:

```
1. if let constantName = someOptional {  
2.     statements  
3. }
```

Мы можем переписать пример `possibleNumber` сверху, используя привязку опционалов, а не принудительное извлечение:

```
1. if let actualNumber = possibleNumber.toInt() {  
2.     println("\'(possibleNumber)\' has an integer value  
   of \'(actualNumber)\")  
3. } else {  
4.     println("\'(possibleNumber)\' could not be  
   converted to an integer")  
5. }  
6. // напечатает "123 имеет целое значение 123"
```

Это может быть прочитано как:

«Если опциональный `Int` возвращаемый `possibleNumber.toInt` содержит значение, установи в новую константу с названием `actualNumber` значение, содержащееся в опционале»

Если преобразование прошло успешно, константа `actualNumber` становится доступной для использования внутри первого ветвления оператора `if`. Он уже инициализируется значением,

содержащимся *внутри* опционала, и поэтому нет необходимости в использовании `!` для доступа к его значению. В этом примере, `actualNumber` просто используется, чтобы напечатать результат преобразования.

Вы можете использовать и константы и переменные для привязки опционалов. Если вы хотели использовать значение `actualNumber` внутри первого ветвления оператора `if`, вы могли бы написать `if var actualNumber` вместо этого, и значение, содержащееся в опционале, будет использоваться как переменная, а не константа.

Неявно извлеченные опционалы

Как описано выше, опционалы показывают, что константам или переменным разрешено не иметь «никакого значения». Опционалы можно проверить с помощью оператора `if`, чтобы увидеть существует ли значение, и при условии, если оно существует, можно извлечь его с помощью привязки опционалов для доступа к опциональному значению.

Иногда, сразу понятно из структуры программы, что опционал *всегда* будет иметь значение, после того как это значение впервые было установлено. В этих случаях, очень полезно избавиться от проверки и извлечения значения опционала каждый раз при обращении к нему, потому что можно с уверенностью утверждать, что он постоянно имеет значение.

Эти виды опционалов называются *неявно извлеченные опционалы*. Их можно писать, используя восклицательный знак `(String!)`, вместо вопросительного знака `(String?)`, после типа, который вы хотите сделать опциональным.

Неявно извлеченные опционалы полезны, когда известно, что значение опционала существует непосредственно после первого объявления опционала, и точно будет существовать после этого. Основное использование неявно извлечённых опционалов в Swift во время инициализации класса, как описано в Отношения без владельца и свойства неявно извлеченных опционалов.

Честно говоря, неявно извлеченные опционалы - это нормальные опционалы, но они могут быть использованы как не опциональные значения, без необходимости в извлечении опционального значения каждый раз при доступе. Следующий пример показывает разницу в поведении между опциональной строкой и неявно извлеченной опциональной строкой при доступе к их внутреннему значению как к явной строке:

```
1. let possibleString: String? = "An optional string."
2. let forcedString: String = possibleString! //
   необходим восклицательный знак
3.
4. let assumedString: String! = "An implicitly
   unwrapped optional string."
5. let implicitString: String = assumedString //
   восклицательный знак не нужен
```

Можно представлять неявно извлеченный опционал как передачу прав опционалу для автоматического извлечения всякий раз, когда он используется. Вместо размещения восклицательного знака после имени опционала каждый раз, когда вы его используете, ставьте восклицательный знак после типа опционала в момент его объявления.

Заметка

Если вы попытаетесь получить доступ к неявно извлеченному опционалу когда он не содержит значения - вы получите runtime ошибку. Результат будет абсолютно тот же, если бы

вы разместили восклицательный знак после нормального опционала, который не содержит значения.

Вы по-прежнему можете обращаться к неявно извлеченному опционалу как к нормальному опционалу, чтобы проверить, содержит ли он значение:

```
1. if assumedString != nil {
2.   println(assumedString)
3. }
4. // напечатает "An implicitly unwrapped optional
   string."
```

Вы также можете использовать неявно извлеченный опционал с привязкой опционалов, чтобы проверить и извлечь его значение в одном выражении:

```
1. if let definiteString = assumedString {
2.   println(definiteString)
3. }
4. // напечатает "An implicitly unwrapped optional
   string."
```

Заметка

Не используйте неявно извлеченный опционал, если существует вероятность, что в будущем переменная может стать `nil`. Всегда используйте нормальный тип опционала, если вам нужно проверять на `nil` значение в течение срока службы переменной.

Утверждение

Опционалы позволяют проверить существует ли значение, или нет, и писать код, который корректно относится к отсутствию значения. В некоторых случаях, однако, это просто невозможно для вашего кода продолжать выполняться, если значение не существует, или если данное значение не удовлетворяет определенным условиям. В таких ситуациях, вы можете вызвать утверждение (assertion) в коде для остановки выполнения и предоставления возможности отладки причины отсутствующего или недопустимого значения.

Отладка с помощью утверждений

Утверждение представляет собой runtime проверку того, что логическое условие определенно вычисляется как истина. Буквально говоря, утверждение «утверждает», что условие истинно. Используйте утверждения, чтобы убедиться, что главное условие выполняется перед тем как выполнить дальнейший код. Если условие истинно, выполнение кода продолжается как обычно. Если условие вычисляется как ложное, выполнение кода прекращается, а ваше приложение завершается.

Если ваш код вызывает утверждение во время работы в отладочной среде, например, когда вы запускаете приложение в Xcode, вы можете увидеть где именно появится неверное состояние и запросить состояние вашего приложения в момент, когда утверждение было вызвано. Утверждение также позволяет обеспечить подходящим отладочным сообщением, отражающим суть утверждения.

Утверждения можно писать через вызов глобальной функции `assert`. Вы передаете функции `assert` некое

выражение, которое вычисляется в `true` или `false`, а также сообщение, которое должно отображаться в случае, если результат условия `false`:

```
1. let age = -3
2. assert(age >= 0, "A person's age cannot be less
   than zero")
3. // это приведет к вызову утверждения, потому что
   age >= 0
```

В этом примере, выполнение кода продолжится, только если `age >= 0` вычислится в `true`, что может случиться, если значение `age` не отрицательное. Если значение `age` отрицательное, как в коде выше, тогда `age >= 0` вычислится как `false`, и запустится утверждение. завершив за собой приложение.

Сообщение утверждения можно пропускать по желанию, как в следующем примере:

```
assert(age >= 0)
```

Когда использовать утверждения

Используйте утверждения, когда условие вероятно может стать `false`, но, для продолжения выполнения кода, он определенно должен быть `true`. Подходящие сценарии для проверки утверждения включают:

- Индекс целого числа передается для работы над пользовательским индексом, но значение индекса может быть слишком маленьким или слишком большим.
- Значение передается функции, но если значение недопустимо, это будет означать, что функция не сможет выполнить свою задачу.

- Опциональное значение в текущем состоянии `nil`, но значение `не-nil` необходимо для последующего успешного выполнения кода.

Смотрите также главы Индексы и Функции.

Заметка

Утверждения приводят к завершению вашего приложения и не являются заменой проектирования кода в надежде, что недействительные условия вряд ли возникнут. Тем не менее, в ситуациях, когда недопустимые условия возможны, утверждение является эффективным способом для того, чтобы такие условия были выделены и замечены во время разработки, перед публикацией вашего приложения.

Базовые операторы

Оператор — это специальный символ или выражение для проверки, изменения или сложения величин. Например, оператор сложения (+) суммирует два числа (`let i = 1 + 2`). Более сложными операторами являются И `&&` (`if enteredDoorCode && passedRetinaScan`), а также инкремент `++i`, который сокращенно обозначает увеличение `i` на `1`.

Язык Swift поддерживает большинство стандартных операторов C, а также ряд возможностей для устранения типичных ошибок в коде. Оператор присваивания (=) не возвращает значение, что позволяет избежать путаницы с оператором проверки на равенство (==). Арифметические операторы (+, -, *, /, % и т. д.) могут обнаруживать и предотвращать переполнение типа, чтобы числовой переменной нельзя было присвоить слишком большое или слишком маленькое значение. Контроль переполнения типа включается в Swift специальными операторами, которые описаны в разделе Операторы переполнения.

В отличие от C язык Swift позволяет делить с остатком (%) числа с плавающей точкой. Также в Swift имеются два сокращенных оператора интервала (`a..b` и `a...b`), которых нет в C.

В этой главе описываются стандартные операторы Swift. Более сложные операторы Swift рассмотрены в главе Дополнительные операторы, где описано, как объявить пользовательские операторы и реализовать стандартные операторы для пользовательских типов.

Терминология

Операторы делятся на унарные, бинарные и тернарные:

- *Унарные* операторы применяются к одной величине (например, `-a`). Унарные *префиксные* операторы ставятся непосредственно перед величиной (например, `!b`), а унарные *постфиксные* операторы — сразу за ней (например, `i++`).
- *Бинарные* операторы применяются к двум величинам (например, `2 + 3`) и являются *инфиксными*, так как ставятся между этими величинами.
- *Тернарные* операторы применяются к трем величинам. Как и в языке C, в Swift есть только один такой оператор, а именно — тернарный условный оператор (`a ? b : c`).

Величины, к которым применяются операторы, называются *операндами*. В выражении `1 + 2` символ `+` является бинарным оператором, а его операндами служат `1` и `2`.

Оператор присваивания

Оператор присваивания (`a = b`) инициализирует или изменяет значение переменной `a` на значение `b`:

```
1. let b = 10
2. var a = 5
3. a = b
4. // теперь a равно 10
```

Если правая часть выражения является кортежем с несколькими значениями, его элементам можно присвоить сразу несколько констант или переменных:

```
1. let (x, y) = (1, 2)
2. // x равно 1, а y равно 2
```

В отличие от C и Objective-C оператор присваивания в Swift не может возвращать значение. К примеру, следующее выражение недопустимо:

```
1. if x = y {
2.     // это неверно, так как x = y не возвращает
   никакого значения
3. }
```

Эта особенность не позволяет разработчику спутать оператор присваивания (`=`) с оператором проверки на равенство (`==`). Благодаря тому, что выражения типа `if x = y` некорректны, подобные ошибки при программировании на Swift не произойдут.

Арифметические операторы

Язык Swift поддерживает четыре стандартных *арифметических оператора* для всех числовых типов:

- сложение (+)
- вычитание (-)
- умножение (*)
- деление (/)

```
1. 1 + 2 // равно 3
2. 5 - 3 // равно 2
3. 2 * 3 // равно 6
4. 10.0 / 2.5 // равно 4.0
```

В отличие от C и Objective-C арифметические операторы Swift по умолчанию не допускают переполнения типа. Контроль переполнения типа включается в Swift специальными операторами (например, `a &+ b`). Подробнее см. в главе Операторы переполнения.

Оператор сложения служит также для конкатенации, или же склейки, строковых значений (тип `String`):

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

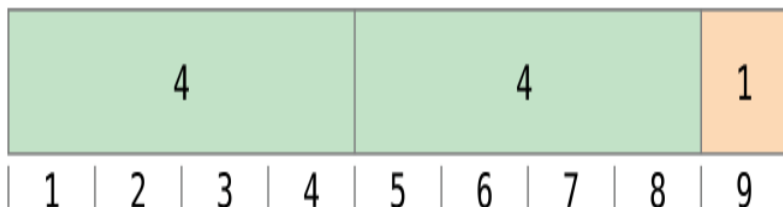
Оператор целочисленного деления

Оператор целочисленного деления ($a \% b$) показывает, сколько целых раз значение b содержится в a , возвращая при этом разницу (остаток от деления).

Заметка

Оператор целочисленного деления ($\%$) в некоторых языках называется *оператором деления по модулю*. Однако учитывая его действие над отрицательными числами в Swift, этот оператор, строго говоря, выполняет деление с остатком, а не по модулю.

Оператор целочисленного деления работает следующим образом. Для вычисления выражения $9 \% 4$ сначала определяется, сколько **четверок** содержится в **девятке**:



В одной **девятке** содержатся две **четверки**, а остатком будет 1 (выделено оранжевым цветом).

На языке Swift это записывается так:

9 % 4 // равно 1

Чтобы получить результат деления $a \% b$, оператор `%` вычисляет следующее выражение и возвращает остаток:

$$a = (b \times \text{множитель}) + \text{остаток}$$

где `множитель` показывает, сколько целых раз `b` содержится в `a`.

Подставляя в это выражение 9 и 4, получим:

$$9 = (4 \times 2) + 1$$

Точно так же рассчитывается остаток, когда `a` отрицательно:

-9 % 4 // равно -1

Подставляя в наше выражение -9 и 4, получим:

$$-9 = (4 \times -2) + -1$$

причем остаток будет равен -1.

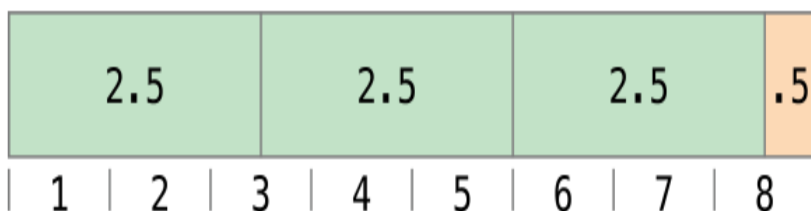
Если `b` отрицательно, его знак отбрасывается. Это означает, что выражения $a \% b$ и $a \% -b$ всегда будут давать одинаковый результат.

Вычисления целочисленного деления с числами с плавающей точкой

В отличие от C и Objective-C оператор целочисленного деления в Swift поддерживает числа с плавающей точкой:

```
8 % 2.5 // равно 0.5
```

В этом примере деление 8 на 2.5 дает 3 с остатком 0.5, поэтому оператор целочисленного деления возвратит значение 0.5 типа `Double`.



Операторы инкремента и декремента

Как и в C, в языке Swift имеются *оператор инкремента* (`++`) и *оператор декремента* (`--`), которые сокращенно обозначают увеличение и уменьшение переменной на 1 соответственно. Эти операторы можно применять как к целым, так и к вещественным переменным.

```
1. var i = 0
2. ++i // теперь i равно 1
```

При каждом вызове оператора `++i` значение `i` увеличивается на 1. По сути, `++i` является краткой формой выражения `i = i + 1`. Аналогично, `--i` служит краткой формой выражения `i = i - 1`.

Символы `++` и `--` могут быть как префиксными операторами, так и постфиксными. Выражения `++i` и `i++` эквивалентны и увеличивают значение `i` на 1. Точно так же эквивалентны выражения `--i` и `i--`, которые уменьшают значение `i` на 1.

Важно отметить, что эти операторы не только изменяют величину `i`, но и возвращают значение. Если требуется лишь увеличить или уменьшить `i` на единицу, возвращенное значение можно проигнорировать. Однако если возвращенное значение действительно нужно использовать, то в зависимости от префиксной или постфиксной формы оператора будут применяться следующие правила:

- Если оператор стоит *перед* переменной, он увеличит ее значение до того, как возвратит результат.
- Если оператор стоит *после* переменной, он увеличит ее значение после того, как возвратит результат.

Пример:

```
1. var a = 0
2. let b = ++a
3. // теперь a и b равны 1
4. let c = a++
5. // теперь a равно 2, но c снова стало 1, как до
   применения инкремента
```

В примере выше выражение `let b = ++a` увеличивает `a` на единицу *до того*, как возвращает значение. Именно поэтому `a` и `b` получают новое значение 1.

Однако выражение `let c = a++` увеличивает `a` на единицу *после того*, как возвращает значение. В результате `c` снова становится равной 1, в то время как `a` приравнивается к 2.

В случаях, когда не требуется использовать именно `i++`, лучше писать `++i` и `--i`, так как эти конструкции работают привычнее — сначала изменяют `i` и только потом возвращают результат.

Оператор унарного минуса

Для изменения знака числового значения служит префиксный минус (`-`), который называется *оператором унарного минуса*:

```
1. let three = 3
2. let minusThree = -three // minusThree равно -3
3. let plusThree = -minusThree // plusThree равно 3,
   т. е. "минус минус три"
```

Оператор унарного минуса (`-`) ставится непосредственно перед значением, без пробела.

Оператор унарного плюса

Оператор *унарного плюса* (+) просто возвращает исходное значение без каких-либо изменений:

```
1. let minusSix = -6
2. let alsoMinusSix = +minusSix // alsoMinusSix равно -6
```

Хотя оператор унарного плюса не выполняет никаких действий, он придает коду единообразие, позволяя зрительно отличать положительные значения от отрицательных.

Составные операторы присваивания

Как и в языке C, в Swift имеется *составные операторы присваивания*, совмещающий простое присваивание (=) с другой операцией. Одним из примером может служить *оператор присваивания со сложением* (+=):

```
1. var a = 1
2. a += 2
3. // теперь a равно 3
```

Выражение `a += 2` является краткой формой записи `a = a + 2`. Таким образом, один и тот же оператор выполняет одновременно операцию сложения и присваивания.

Заметка

Составные операторы присваивания не возвращают значение. К примеру, нельзя написать так: `let b = a +=`
2. Принцип работы этих операторов отличается от рассмотренных выше операторов инкремента и декремента.

Полный список составных операторов присваивания приведен в главе Выражения.

Операторы сравнения

Язык Swift поддерживает все стандартные *операторы сравнения* из C:

- равно (`a == b`)
- не равно (`a != b`)
- больше (`a > b`)
- меньше (`a < b`)
- больше или равно (`a >= b`)
- меньше или равно (`a <= b`)

Заметка

В языке Swift есть также два *оператора проверки на равенство* (`===` и `!==`), определяющие, ссылаются ли два указателя на один и тот же экземпляр объекта. Дополнительную информацию см. в главе Классы и структуры.

Каждый оператор сравнения возвращает значение типа `Bool`, указывающее, является ли выражение истинным:

```
1. 1 == 1 // истина, так как 1 равно 1
2. 2 != 1 // истина, так как 2 не равно 1
3. 2 > 1 // истина, так как 2 больше чем 1
4. 1 < 2 // истина, так как 1 меньше 2
5. 1 >= 1 // истина, так как 1 больше либо равно 1
6. 2 <= 1 // ложь, так как 2 не меньше либо равно 1
```

Операторы сравнения часто используются в условных выражениях, включая конструкцию `if`:

```
1. let name = "world"
2. if name == "world" {
3.   println("hello, world")
4. } else {
5.   println("I'm sorry \(name), but I don't recognize you")
6. }
7. // напечатает "hello, world", так как name очевидно
   равно "world"
```

Подробнее о конструкции `if` см. в главе Поток выполнения.

Тернарный условный оператор

Тернарный условный оператор — это специальный оператор из трех частей, имеющий следующий синтаксис: **выражение ? действие1 : действие2**. Он выполняет одно из двух действий в зависимости от того, является ли выражение истинной или ложью. Если **выражение** истинно, оператор выполняет **действие1** и возвращает его результат; в противном случае оператор выполняет **действие2** и возвращает его результат.

Тернарный условный оператор является краткой записью следующего кода:

```
1. if выражение {  
2.   действие1  
3. } else {  
4.   действие2  
5. }
```

Ниже приведен пример расчета высоты строки в таблице. Если у строки есть заголовок, то она должна быть выше своего содержимого на 50 точек, а если заголовка нет, то на 20 точек:

```
1. let contentHeight = 40  
2. let hasHeader = true  
3. let rowHeight = contentHeight +  
   (hasHeader ? 50 : 20)  
4. // rowHeight равно 90
```

В развернутом виде этот код можно записать так:

```
1. let contentHeight = 40
2. let hasHeader = true
3. var rowHeight = contentHeight
4. if hasHeader {
5.   rowHeight = rowHeight + 50
6. } else {
7.   rowHeight = rowHeight + 20
8. }
9. // rowHeight равно 90
```

В первом примере с помощью тернарного условного оператора величине `rowHeight` в одну строку присваивается правильное значение. Этот вариант не только короче второго примера, но и позволяет объявить величину `rowHeight` константой, так как в отличие от конструкции `if` ее значение не нужно изменять.

Тернарный условный оператор — это короткая и удобная конструкция для выбора между двумя выражениями. Однако тернарный условный оператор следует применять с осторожностью. Избыток таких коротких конструкций иногда делает код трудным для понимания. В частности, лучше не использовать несколько тернарных условных операторов в одном составном операторе присваивания.

Оператор объединения по нулевому указателю

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

Оператор объединения по нулевому указателю является краткой записью следующего кода:

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

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

Заметка

Если `a` не равно `nil`, выражение `b` не анализируется. Такой подход называется *краткой проверкой условия* (**short-circuit evaluation**).

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

```

1. let defaultColorName = "red"
2. var userDefinedColorName: String? // по умолчанию
   равно nil
3.
4. var colorNameToUse = userDefinedColorName ?? default
   ColorName
5. // userDefinedColorName равен nil, поэтому
   colorNameToUse получит значение по умолчанию —
   "red"

```

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

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

```

1. userDefinedColorName = "green"
2. colorNameToUse = userDefinedColorName ?? defaultCol
   orName
3. // userDefinedColorName не равно nil, поэтому
   colorNameToUse получит значение "green"

```

Операторы диапазона

В языке Swift есть два *оператора диапазона*, которые в короткой форме задают диапазон значений.

Оператор закрытого диапазона

Оператор закрытого диапазона (`a...b`) задает диапазон от `a` до `b`, включая сами `a` и `b`. При этом значение `a` не должно превышать `b`.

Оператор закрытого диапазона удобно использовать при последовательном переборе значений из некоторого диапазона, как, например, в цикле `for-in`:

```
1. for index in 1...5 {
2.     println("\(index) times 5 is \(index * 5)")
3. }
4. // 1 умножить на 5 будет 5
5. // 2 умножить на 5 будет 10
6. // 3 умножить на 5 будет 15
7. // 4 умножить на 5 будет 20
8. // 5 умножить на 5 будет 25
```

Подробнее о циклах `for-in` см. в главе Поток выполнения.

Оператор полуоткрытого диапазона

Оператор полуоткрытого диапазона (`a..<b`) задает диапазон от `a` до `b`, исключая значение `b`. Такой диапазон называется полуоткрытым, потому что он включает первое значение, но исключает последнее. Так же, как и для

оператора закрытого диапазона, значение **a** не должно превышать **b**.

Оператор полуоткрытого диапазона особенно удобны при работе с массивами и другими последовательностями, пронумерованными с нуля, когда нужно перебрать элементы от первого до последнего:

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. let count = names.count
3. for i in 0..
```

Хотя в массиве четыре элемента, диапазон `0.. доходит только до 3 (т. е. до номера последнего элемента в массиве), так как это оператор полуоткрытого диапазона. Подробнее о массивах см. в главе Массивы.`

Логические операторы

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

- логическое НЕ (`!a`)
- логическое И (`a && b`)
- логическое ИЛИ (`a || b`)

Оператор логического НЕ

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

Оператор логического НЕ является префиксным и ставится непосредственно перед значением, без пробела. Как видно из следующего примера, его можно воспринимать как "**не** `a`":

```
1. let allowedEntry = false
2. if !allowedEntry {
3.     println("ACCESS DENIED")
4. }
5. // напечатает "ACCESS DENIED"
```

Конструкция `if !allowedEntry` означает "если не `allowedEntry`". Идущая за ней строка будет выполнена, только если "**не** `allowedEntry`" является истиной, т. е. если `allowedEntry` равно `false`.

Как видно из этого примера, удачный выбор булевой константы и имен переменных делает код коротким и понятным, без двойных отрицаний и громоздких логических выражений.

Оператор логического И

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

Если хотя бы один из них равен `false`, результатом всего выражения тоже будет `false`. На самом деле, если *первое* значение равно `false`, то второе даже не будет анализироваться, так как оно все равно не изменит общий результат на `true`. Такой подход называется краткой проверкой условия (*short-circuit evaluation*).

В следующем примере проверяются два значения типа `Bool`, и если они оба равны `true`, программа разрешает доступ:

```
1. let enteredDoorCode = true
2. let passedRetinaScan = false
3. if enteredDoorCode && passedRetinaScan {
4.   println("Welcome!")
5. } else {
6.   println("ACCESS DENIED")
7. }
8. // напечатает "ACCESS DENIED"
```

Оператор логического ИЛИ

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

Как и описанный выше оператор логического И, оператор логического ИЛИ использует краткую проверку условия. Если левая часть выражения с логическим ИЛИ равна `true`, то

правая не анализируется, так как ее значение не повлияет на общий результат.

В приведенном ниже примере первое значение типа `Bool` (`hasDoorKey`) равно `false`, а второе (`knowsOverridePassword`) равно `true`. Поскольку одно из значений равно `true`, результат всего выражения тоже становится `true` и доступ разрешается:

```
1. let hasDoorKey = false
2. let knowsOverridePassword = true
3. if hasDoorKey || knowsOverridePassword {
4.   println("Welcome!")
5. } else {
6.   println("ACCESS DENIED")
7. }
8. // напечатает "Welcome!"
```

Комбинирование логических операторов

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

```
1. if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
2.   println("Welcome!")
3. } else {
4.   println("ACCESS DENIED")
5. }
6. /// напечатает "Welcome!"
```

В этом примере с помощью нескольких операторов `&&` и `||` составляется более длинное и сложное выражение. Однако операторы `&&` и `||` по-прежнему применяются только к двум величинам, поэтому все выражение можно разбить на три простых условия. Алгоритм работы будет следующим:

если пользователь правильно ввел код дверного замка и прошел сканирование сетчатки или если он использовал действующую ключ-карту или если он ввел код экстренного доступа, то дверь открывается.

Исходя из значений `enteredDoorCode`, `passedRetinaScan` и `hasDoorKey` первые два подусловия дают `false`. Однако был введен код экстренного доступа, поэтому все составное выражение по-прежнему равно `true`.

Явное указание круглых скобок

Иногда имеет смысл использовать дополнительные круглые скобки, чтобы сложное логическое выражение стало проще для восприятия. В примере с открытием двери можно заключить в круглые скобки первую часть составного выражения, что сделает его нагляднее:

```
1. if (enteredDoorCode && passedRetinaScan)
   || hasDoorKey || knowsOverridePassword {
2.   println("Welcome!")
3. } else {
4.   println("ACCESS DENIED")
5. }
6. ///напечатает "Welcome!"
```

Круглые скобки показывают, что первые две величины составляют одно из возможных значений всего логического выражения. Хотя результат составного выражения не изменится, такая запись сделает код понятнее. Читаемость кода всегда важнее краткости, поэтому желательно ставить круглые скобки везде, где они облегчают понимание.

Строки и символы

Строка представляет собой упорядоченную совокупность символов, например "hello, world" или "albatross".

Строки в Swift представлены типом `String`, который, в свою очередь, представляет собой коллекцию значений типа `Character`.

Типы `String` и `Character` в Swift предусматривают быстрый, Unicode-совместимый способ работы с текстом в вашем коде. Синтаксис для создания и манипулирования строками легкий и читабельный, он включает синтаксис строковых литералов похожий на C. Конкатенация строк так же проста, как сложение двух строк с помощью оператора `+`, а изменчивость строки можно управлять выбирая к чему присваивать значение, константе или переменной, также как в случае с любым другим значением в Swift.

Несмотря на эту простоту синтаксиса, тип `String` в Swift имеет быструю и современную реализацию. Каждая строка состоит из независимых от кодировки символов Unicode, и обеспечивает поддержку доступа к этим символам в различных Unicode представлениях.

Строки также можно использовать для вставки констант, переменных, литералов, и выражений в более длинные строки через процесс называемый интерполяция строк. Это позволяет легко создавать собственные строковые значения для отображения, хранения и печати.

Заметка

Тип `String` в Swift бесшовно шит с классом `NSString` из Foundation. Если вы работаете с фреймворком Foundation в Cocoa или Cocoa Touch, то весь API `NSString` доступен для каждого значения типа `String` создаваемого вами в Swift,

включая все возможности String, которые описываются в этой главе. Вы также можете использовать значение с типом String для любых API, в которых используется NSString. Для получения дополнительной информации об использовании String с Foundation и Cocoa, обратитесь к книге "Использование Swift с Cocoa и Objective-C".

Строковые литералы

Вы можете включить predefined `String` значения в вашем коде как строковые литералы. Строковый литерал - это фиксированная последовательность текстовых символов, окруженная парой двойных кавычек (`"``"`).

Используйте строковый литерал как начальное значение для константы или переменной:

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

Заметьте, что Swift вывел тип `String` для константы `someString`, потому что он был инициализирован строковым литеральным значением.

Заметка

Для получения информации об использовании специальных символов в строковых литералах, смотрите главу Специальные Unicode символы в строковых литералах.

Инициализация пустых строк

Чтобы создать пустое `String` значение в качестве отправной точки для создания более длинных строк, либо присвойте литерал пустой строки к переменной, либо инициализируйте объект `String` с помощью синтаксиса инициализации:

```
1. var emptyString = "" // empty string literal
2. var anotherEmptyString = String() // initializer
   syntax
3. // обе строки пусты и эквиваленты друг другу
```

Можно узнать пустое ли `String` значение, через его Boolean свойство `isEmpty`:

```
1. if emptyString.isEmpty {
2.   println("Nothing to see here")
3. }
4. // напечатает "Nothing to see here"
```

Изменчивость строк

Вы можете указать, может ли конкретный `String` быть модифицирован, путем присвоения его переменной (в этом случае он может быть модифицирован), или присвоения его константе (в этом случае он не может быть модифицирован):

```
1. var variableString = "Horse"
2. variableString += " and carriage"
3. // variableString теперь "Horse and carriage"
4.
5. let constantString = "Highlander"
6. constantString += " and another Highlander"
7. // это выдаст ошибку компиляции: строковая
   константа не может быть модифицирована
```

Заметка

Этот подход отличается от изменчивости строк в Objective-C и Сосоа, где мы выбираем между двумя классами (`NSString` и `NSMutableString`), чтобы указать, может ли строка быть изменена.

Строка является типом значения

ип `String` в Swift является типом значения. Когда вы создаёте новое `String` значение, это значение копируется когда оно передается функции или методу, или когда оно присваивается константе или переменной. В каждом случае создается новая копия существующего `String` значения, и передаётся либо присваивается новая копия, а не исходная версия. Типы значений описаны в главе Структуры и перечисления являются типами значений.

Заметка

Это поведение отличается от `NSString` в Сосоа. Когда вы создаете объект `NSString` в Сосоа, и передаете его функции или методу, либо присваиваете переменной, вы всегда передаете либо присваиваете ссылку на один и тот же `NSString`. Никакого копирования строки не происходит, если вы специально не просите его.

Подход "копировать по умолчанию" для `String` в Swift позволяет быть уверенным в том, что когда вы передаете функции либо методу `String` значение, то очевидно, что вы имеете точно то же `String` значение, независимо от того, откуда она пришла. Вы можете быть уверены, что строка, которая вам передана, не будет модифицирована, если вы не модифицируете его сами.

Компилятор Swift оптимизирует использование строк, так что фактическое копирование строк происходит только тогда, когда оно действительно необходимо. Это означает, что вы всегда получаете высокую производительность, при работе со строками, как с типами значений.

Работа с символами

Тип `String` в Swift представляет собой коллекцию значений `Character` в указанном порядке. Вы можете получить доступ к отдельным значениям `Character` в строке с помощью итерации по этой строке в `for-in` цикле:

```
1. for character in "Dog! (смайл пса)" {
2.     println(character)
3. }
4. // D
5. // o
6. // g
7. // !
8. // (смайл пса)
```

Цикл `for-in` описан в главе Циклы `for`.

Кроме того, можно создать отдельную `Character` константу или переменную из односимвольного строкового литерала с помощью присвоения типа `Character`:

```
let yenSign: Character = "¥"
```

Конкатенация строк и СИМВОЛОВ

Значения типа `String` могут быть добавлены или конкатенированы с помощью оператора сложения (+):

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
// welcome равен "hello there"
```

Вы можете добавить значение типа `String` к другому, уже существующему значению `String`, с помощью комбинированного оператора сложения и присвоения (`+=`):

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

Вы можете добавить значение типа `Character` к переменной типа `String`, используя метод `String.append`:

```
let exclamationMark: Character = "!"
welcome.append(exclamationMark)
// welcome равен "hello there!"
```

Заметка

Вы не можете добавить `String` или `Character` к уже существующей `Character`, потому что тип `Character` предназначен только для одиночного символа.

Интерполяция строк

Интерполяция строк - способ создать новое значение типа `String` из разных констант, переменных, литералов и выражений, включая их значения в строковый литерал. Каждый элемент, который вы вставляете в строковый литерал, должен быть помещен в двойные кавычки, и перед открывающей скобкой должен стоять знак обратного слэша.

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is
\ (Double(multiplier) * 2.5)"
// message равен "3 times 2.5 is 7.5"
```

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

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

Заметка

Выражение, которое вы пишете внутри скобок в пределах интерполируемой строки, не может содержать одиночные кавычки (`' '`) или обратный слэш (`\`), так же как и не может содержать символ начала новой строки (`\n`) или символ возврат каретки (`\r`).

Типы коллекций

В Swift есть два типа коллекций для хранения наборов значений - это Массивы и Словари. Массивы хранят упорядоченный набор значений одинакового типа. Словари хранят неупорядоченный набор значений одинакового типа, на которые можно сослаться и найти через уникальный идентификатор (также известный как ключ).

В Swift, всегда понятно какие значения и ключи можно хранить в массивах и словарях. Это означает то вы не можете по ошибке вставить значение в массив или словарь. Это также означает, что вы можете быть уверены в типах значений, которые вы получите из массива или словаря. Использование явно типизированных коллекций в Swift гарантирует, что ваш код всегда понимает с какими данными он может работать и позволяет устранить несоответствия типов на ранних стадиях разработки.

Заметка

Забегая вперед скажем, что типы Массив и Словарь в Swift реализованы как универсальные коллекции. Более подробную информацию об универсальных типах и коллекциях можно получить в главе "Универсальные типы".

Изменчивость коллекций

Когда вы создаете массив или словарь и присваиваете его переменной, то созданная коллекция будет изменяемой. Это означает, что вы можете изменить коллекцию после его создания путем добавления, удаления, или изменения элементов этой коллекции. И наоборот, когда вы присвоите массив или словарь константе, то он будет неизменяемым, а его размер и содержимое не может быть изменено.

Заметка

Хорошей практикой является создание неизменяемых коллекций во всех случаях, когда коллекцию не нужно менять. Делая это, мы позволяем компилятору Swift оптимизировать производительность наших коллекций.

Массивы

Массивы хранят много значений одинакового типа в упорядоченном списке. Одно и то же значение в массиве может появиться несколько раз, в разных позициях.

Массивы в Swift точны по поводу типов значений которые они могут хранить. Они различаются от классов `NSArray` и `NSMutableArray` из Objective-C, которые могут хранить любые значения и не дают информацию о типах объектов, которые они могут вернуть. В Swift, всегда ясно какие типы значений может хранить определенный массив, либо через явное присвоение типа, либо через вывод типов, при условии что значение не будет пользовательским классом. Например, когда вы создаете массив из `Int` значений, вы не сможете вставить в этот массив никакое значение отличное от `Int`. Массивы в Swift строго типизированы, и всегда точны по поводу типов которые они могут хранить.

Сокращённый синтаксис массивов

Полная форма записи массива в Swift пишется `Array<SomeType>`, где `SomeType` это тип который может хранить массив.

Вы можете также написать массив в сокращенной форме как `[SomeType]`.

Хотя две формы функционально идентичны, краткая форма является предпочтительной и используется в данном руководстве при обращении к типу массива.

Литералы массива

Вы можете инициализировать массив с помощью литерала массива, который является быстрым способом писать одно или несколько значений как набор значений массива.

Литерал массива пишется в виде списка значений, разделенных запятыми и окруженными парами скобок:

```
[ значение 1, значение 2, значение 3 ]
```

В приведенном ниже примере создается массив под названием `shoppingList` для хранения `String` значений:

1. `var shoppingList: [String] = ["Eggs", "Milk"]`
2. `// shoppingList был инициализирован двумя начальными элементами`

Переменная `shoppingList` объявлена как "массив из `String` значений", который записывается как `[String]`.

Поскольку для этого массива указан тип значения `String`, ему разрешено хранить только `String` значения. Здесь, массив `shoppingList` инициализирован двумя `String` значениями ("`Eggs`" и "`Milk`"), написанными внутри литерала массива.

Заметка

Массив `shoppingList` объявлен как переменная (с помощью `var`), а не константа (с помощью `let`), поскольку много элементов добавляются в список покупок в примерах ниже.

В данном случае литерал массива содержит два `String` значения и больше ничего. Это подходит типу, который мы присвоили при объявлении переменной `shoppingList` (массив который может хранить только `String` значения), и поэтому присвоение литерала массива разрешено как способ инициализации `shoppingList` двумя начальными элементами.

Благодаря выводу типов Swift, вы можете не писать тип для массива, который вы инициализируете с помощью литерала массива, хранящего значения того же типа. Вместо этого, инициализация `shoppingList` может быть записана в сокращенной форме:

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

Поскольку все значения внутри литерала массива одинакового типа, Swift может вывести, что `[String]` является правильным типом для переменной `shoppingList`.

Доступ и изменение массива

Вы можете получить доступ к массиву и изменять его либо через его методы и свойства, либо используя синтаксис сабскриптов.

Чтобы узнать количество элементов в массиве, проверьте его **read-only** свойство `count`:

```
1. println("The shopping list
   contains \(shoppingList.count) items.")
2. // напечатает "The shopping list contains 2 items."
```

Логическое свойство `isEmpty` можно использовать в качестве быстрого способа узнать, является ли свойство `count` равным 0:

```
1. if shoppingList.isEmpty {
2.   println("The shopping list is empty.")
3. } else {
4.   println("The shopping list is not empty.")
5. }
6. // напечатает "The shopping list is not empty."
```

Вы можете добавить новый элемент в конец массива через вызов метода **append**:

```
1. shoppingList.append("Flour")
2. // shoppingList теперь содержит 3 элемента, а кое-
   кто делает блины
```

Кроме того, добавить массив с одним или несколькими совместимыми элементами можно с помощью оператора сложения и присвоения **(+=)**:

```
1. shoppingList += ["Baking Powder"]
2. // shoppingList теперь хранит 4 элемента
3. shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
4. // shoppingList теперь хранит 7 элементов
```

Можно извлечь значение из массива с помощью синтаксиса сабскриптов, поместив индекс значения, который вы хотите получить, внутри квадратных скобок сразу после имени массива.

```
1. var firstItem = shoppingList[0]
2. // firstItem равен "Eggs"
```

Заметьте, что первый элемент в этом массиве имеет индекс 0, а не 1. Массивы в Swift всегда начинаются с 0.

Вы можете использовать синтаксис сабскриптов для изменения существующего значения данного индекса:

```
1. shoppingList[0] = "Six eggs"
2. // первый элемент в списке теперь равен "Six eggs",
   а не "Eggs"
```

Вы также можете использовать синтаксис сабскриптов для изменения диапазона значений за раз, даже если набор изменяющихся значений имеет разную длину, по сравнению с диапазоном который требуется заменить. Следующий пример заменяет "Chocolate Spread", "Cheese", и "Butter" на "Bananas" и "Apples":

```
1. shoppingList[4...6] = ["Bananas", "Apples"]
2. // shoppingList теперь содержит 6 элементов
```

Заметка

Вы не можете использовать синтаксис сабскриптов для добавления нового элемента в конец массива. Если вы попытаетесь использовать синтаксис сабскриптов для получения или установки значения для индекса, который выходит за существующие границы массива, вы получите runtime ошибку. Тем не менее, вы можете проверить существует ли индекс, прежде чем его использовать, путем сравнения его со свойством `count` массива. За исключением тех случаев, когда `count` равен 0 (то есть массив пуст), наибольший допустимый индекс в массиве должен быть 1, поскольку массивы индексируются с нуля.

Для вставки элемента по заданному индексу внутрь массива, вызовите его метод `insert(atIndex:)` :

```
1. shoppingList.insert("Maple Syrup", atIndex: 0)
2. // shoppingList теперь содержит 7 элементов
3. // "Maple Syrup" теперь первый элемент списка
```

Вызвав этот `insert` метод, мы вставили новый элемент со значением `"Maple Syrup"` в самое начало списка покупок, то есть в элемент с индексом 0.

Аналогичным образом можно удалить элемент из массива с помощью метода `removeAtIndex`. Этот метод удаляет элемент с указанным индексом и возвращает удалённый элемент (хотя вы можете игнорировать возвращаемое значение если оно вам не нужно):

```
1. let mapleSyrup = shoppingList.removeAtIndex(0)
2. // элемент который имел индекс 0 был удален
3. // shoppingList теперь содержит 6 элементов, и нет Maple Syrup
4. // константа mapleSyrup теперь равна удаленной строке "Maple Syrup"
```

Любые пробелы внутри массива закрываются когда удаляется элемент, и поэтому значение с индексом 0 опять равно "Six eggs":

```
1. firstItem = shoppingList[0]
2. // firstItem теперь равен "Six eggs"
```

Если вы хотите удалить последний элемент из массива, то можно использовать метод `removeLast` вместо `removeAtIndex`, чтобы избежать необходимости запроса свойства `count` для массива. Также как и метод `removeAtIndex`, `removeLast` возвращает удаленный элемент:

```
1. let apples = shoppingList.removeLast()
2. // последний элемент массива был удален
3. // shoppingList теперь содержит 5 элементов, и нет
  // яблочек
4. // константа apples теперь равна удаленной строке
  "Apples"
```

Итерация по массиву

Вы можете выполнить итерацию по всему набору значений внутри массива с помощью цикла `for-in`:

```
1. for item in shoppingList {
2.   println(item)
3. }
4. // Six eggs
5. // Milk
6. // Flour
7. // Baking Powder
8. // Bananas
```

Если вам нужен целочисленный индекс каждого значения так же как и самое значение, используйте вместо этого глобальную функцию `enumerate` для итерации по массиву. Функция `enumerate` возвращает кортеж для каждого элемента массива, собрав вместе индекс и значение для этого элемента. Вы можете разложить кортеж во временные константы или переменные в рамках итерации:

```
1. for (index, value) in enumerate(shoppingList) {
2.     println("Item \ (index + 1): \ (value)")
3. }
4. // Item 1: Six eggs
5. // Item 2: Milk
6. // Item 3: Flour
7. // Item 4: Baking Powder
8. // Item 5: Bananas
```

Чтобы получить подробную информацию про цикл `for-in`, смотрите главу "Циклы For".

Создание и инициализация массива

Вы можете создать пустой массив определенного типа (без присвоения какого-либо начального значения) используя синтаксис инициализатора:

```
1. var someInts = [Int]()
2. println("someInts is of type [Int]
   with \ (someInts.count) items.")
3. // напечатает "someInts is of type [Int] with 0
   items."
```

Заметьте, что тип переменной `someInts` выведен как `[Int]`, поскольку ему присваивается результат инициализатора `[Int]`.

С другой стороны, если по контексту можно получить информацию о типе, например если это аргумент функции, либо типизированная переменная или константа, вы можете создать пустой массив с помощью пустого литерала массива, который пишется как `[]` (пара пустых квадратных скобок):

```
1. someInts.append(3)
2. // someInts теперь содержит 1 значение типа Int
3. someInts = []
4. // someInts теперь пустой массив, но по-прежнему
   остается типом [Int]
```

Тип `Array` в Swift имеет также инициализатор для создания массива определенного размера, со всеми значениями установленными в указанное значение по умолчанию. Вы передаете этому инициализатору количество элементов для добавления в новый массив (параметр `count`) и значение по умолчанию определенного типа (параметр `repeatedValue`):

```
1. var threeDoubles =
   [Double](count: 3, repeatedValue: 0.0)
2. // threeDoubles имеет тип [Double], и равен [0.0,
   0.0, 0.0]
```

Вы можете создать новый массив путем сложения вместе двух существующих массивов соответствующего типа, используя оператор сложения (`+`). Тип нового массива выведен из типа двух массивов, которые вы сложили вместе:

```
1. var anotherThreeDoubles =
   [Double](count: 3, repeatedValue: 2.5)
2. // anotherThreeDoubles выведен тип [Double], и
   равен [2.5, 2.5, 2.5]
3.
4. var sixDoubles = threeDoubles + anotherThreeDoubles
5. // sixDoubles выведен тип [Double], и равен [0.0,
   0.0, 0.0, 2.5, 2.5, 2.5]
```

Множества

Множество хранит различные значения одного типа в виде коллекции в неупорядоченной форме. Вы можете использовать множества как альтернативы массиву, когда порядок для вас значения не имеет или когда вам нужны быть уверенным в том, что значения внутри коллекции не повторяются.

Заметка

Тип Swift `Set` связан с классом Foundation `NSSet`.

Хеш значения для типа Set

Тип должен быть *хешируемым* для того, чтобы мог храниться в множестве, таким образом тип должен предоставлять возможность для вычисления собственного *значения хеша*. Тип значения хеша `Int` должен быть для всех объектов одинаковым, чтобы можно было провести сравнение что если `a == b`, то и `a.hashValue == b.hashValue`.

Все базовые типы Swift (`Int`, `String`, `Double`, `Bool`) являются хешируемыми типами по умолчанию и могут быть использованы в качестве типов значений множества или в качестве типов ключей словаря. Значения членов перечисления без каких-либо связанных значений (что описано в главе "Перечисления") также являются хешируемыми по умолчанию.

Заметка

Вы можете использовать ваш собственный тип в качестве типа значения множества или типа ключа словаря, подписав его под протокол `Hashable` из стандартной библиотеки Swift. Типы, которые подписаны под протокол `Hashable` должны обеспечивать `gettable` свойство `hashValue`. Значение, которое

возвращает `hashValue` не обязательно должно иметь одно и то же значение при выполнении одной и той же программы или разных программ. Так как протокол `Hashable` подписан под протокол `Equatable`, то подписанные под него типы так же должны предоставлять реализацию оператора равенства `==`. Протоколе `quatable` требует любую реализацию оператора равенства для реализации возможности сравнения. Таким образом, реализация оператора `==` должна удовлетворять следующим трем условиям, для всех трех значений `a, b, c`.

1. `a == a` (Рефлексивность)
2. `a == b`, значит `b == a` (Симметрия)
3. `b == a && b == c`, значит `a == c` (Транзитивность)

Для более подробной информации читайте главу ["Протоколы"](#).

Синтаксис типа множества

Тип множества Swift записывается как `Set<T>`, `T` является типом, который храниться в множестве. В отличии от массивов множества не имеют сокращенной формы записи.

Создание и инициализация пустого множества

Вы можете создать пустое множество конкретного типа, используя синтаксис инициализатора:

```
var letters = Set<Character>()
print("letters имеет тип Set<Character> с
\ (letters.count) элементами.")
// напечатает "letters имеет тип
Set<Character> с 0 элементов."
```


Заметка

Тип переменной `letters` выведен из типа инициализатора как `Set<Character>`.

Альтернативно, если контекст предоставляет информацию о типе, например как аргумент функции или просто явное указание типа переменной или константы, то вы можете создать пустое множество при помощи пустого литерала массива:

```
letters.insert("a")
// letters now contains 1 value of type
Character
letters = []
// letters теперь является пустым множеством,
но все еще имеет тип Set<Character>
```

Создание множества при помощи литерала массива

Вы так же можете инициализировать множество при помощи литерала массива, чтобы использовать его в качестве сокращенной записи нескольких элементов в качестве коллекции множества.

Пример ниже создает множество `favoriteGenres` для хранения `String`.

```
var favoriteGenres: Set<String> = ["Rock",
"Classical", "Hip hop"]

// favoriteGenres был инициализирован при
помощи трех начальных элементов
```

Переменная `favoriteGenres` объявлена как множество значений типа `String`, который записывается как `Set<String>`. Так как это множество имеет определенный тип `String`, то этому множеству позволено хранить только значения типа `String`. Поэтому здесь мы инициализируем `favoriteGenres` тремя значениями типа `String`, записанными в виде литерала массива.

Заметка

Множество `favoriteGenres` объявлен как переменная (ключевое слово `var`), но не константа (ключевое слово `let`), так как мы добавляем и удаляем элементы в примере ниже.

Так как тип множества не может быть выведен только из литерала, то его тип должен быть указан явно. Однако из-за вывода типа в Swift вы не должны писать тип множества, если вы инициализируете его при помощи литерала массива, который содержит элементы одного типа. Вместо этого инициализация `favoriteGenres` может быть записана и в более короткой форме:

```
var favoriteGenres: Set = ["Rock",  
"Classical", "Hip hop"]
```

Так как все элементы литерала массива одного типа, то Swift может вывести, что `Set<String>` является корректным типом для переменной `favoriteGenres`.

Доступ и изменение множества

Получить доступ и модифицировать множества можно через свойства и методы.

Для того, чтобы выяснить количество элементов в множестве вам нужно использовать свойство `count`:

```
print("У меня есть \(favoriteGenres.count)  
любимых музыкальных жанра.")  
// prints "У меня есть 3 любимых музыкальных  
жанра."
```

Используйте булево свойство `isEmpty` в качестве сокращенной проверки наличия элементов во множестве или другими словами равно ли свойство `count` 0:

```
if favoriteGenres.isEmpty {  
    print("Мне все равно какая музыка играет.  
Я не придирчив.")  
} else {  
    print("У меня есть свои музыкальные  
предпочтения.")  
}  
// prints "У меня есть свои музыкальные  
предпочтения."
```

Вы можете добавить новый элемент во множество, используя метод `insert(_):`

```
favoriteGenres.insert("Jazz")  
// теперь в favoriteGenres находится 4  
элемента
```

Вы так же можете удалить элемент из множества, используя метод `remove(_:)`, который удаляет элемент, который является членом множества и возвращает удаленное значение или `nil`, если удаляемого элемента нет. Так же все объекты множества могут быть удалены одновременно при помощи метода `removeAll()`.

```
if let removedGenre =
  favoriteGenres.remove("Rock") {
    print("\(removedGenre)? С меня хватит.")
} else {
    print("Меня это не сильно заботит.")
}
// prints "Rock? С меня хватит."
```

Можно проверить наличие определенного элемента во множестве, используя метод `contains(_:)`:

```
if favoriteGenres.contains("Funk") {
    print("О! Да я встал с правильной ноги!")
} else {
    print("Слишком много Funk'a тут.")
}
// prints "Слишком много Funk'a тут."
```

Итерация по множеству

Вы можете совершать итерации по множеству при помощи цикла `for-in`.

```
for genre in favoriteGenres {  
    print("\(genre)")  
}  
// Classical  
// Jazz  
// Hip hop
```

Для более подробной информации по циклу `for-in` читайте в главе "[Циклы For](#)".

Множества в Swift не имеют определенного порядка. Для того, чтобы провести итерацию по множеству в определенном порядке вам нужно использовать функцию `sort()`, которая возвращает вам коллекцию определенной последовательности.

```
for genre in favoriteGenres.sort() {  
    print("\(genre)")  
}  
// Classical  
// Hip hop  
// Jazz
```

Выполнение операций Set

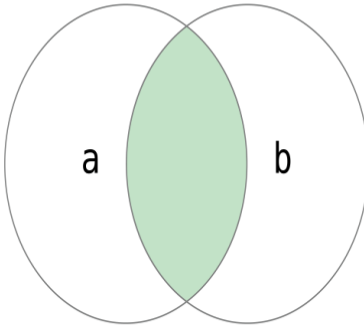
Вы можете очень эффективно использовать базовые операции множества, например, комбинирование двух множеств, определение общих значений двух множеств, определять содержат ли множества несколько, все или ни одного одинаковых значения.

Базовые операции set

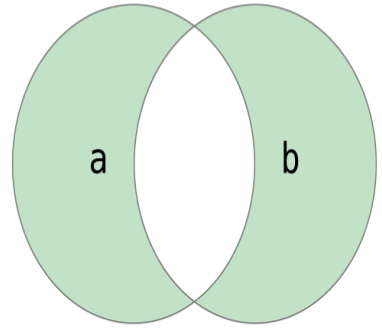
Иллюстрации внизу изображают два множества **a** и **b** в результате применения различных методов.



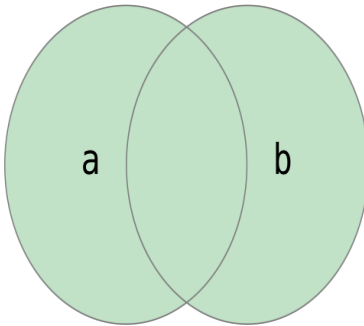
`a.intersects(b)`



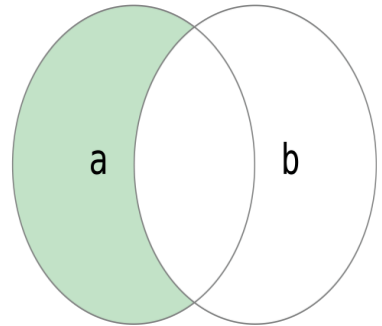
`a.exclusiveOr(b)`



`a.union(b)`



`a.subtract(b)`



- Используйте метод `intersect(_:)` для создания нового множества из общих значений двух входных множеств.
- Используйте метод `exclusiveOr(_:)` для создания нового множества из значений, которые не повторяются в двух входных множествах.
- Используйте метод `union(_:)` для создания нового множества состоящего из всех значений обоих множеств.
- Используйте метод `subtract(_:)` для создания множества со значениями исключительно принадлежащих только одному из множеств.

```

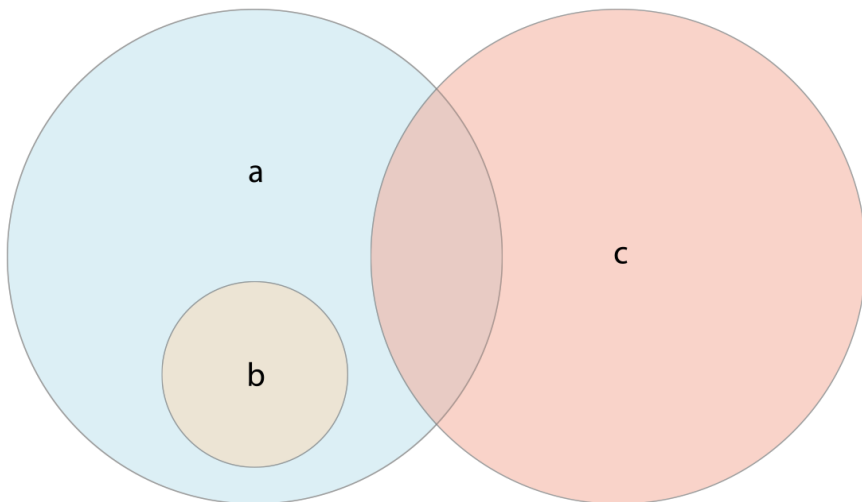
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sort()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersect(evenDigits).sort()
// []
oddDigits.subtract(singleDigitPrimeNumbers).sort()
// [1, 9]
oddDigits.exclusiveOr(singleDigitPrimeNumbers).sort()
// [1, 2, 9]

```

Взаимосвязь и равенство множеств

Иллюстрация ниже отображает три множества **a**, **b** и **c**. Множество **a** является надмножеством множества **b**, так как содержит все его элементы, соответственно множество **b** является подмножеством множества **a**, опять таки потому, что все его элементы находятся в **a**. Множества **b** и **c** называются *разделенными*, так как у них нет общих элементов.



[OBJ]

- Используйте оператор равенства (`==`) для определения все ли значения двух множеств одинаковы.
- Используйте метод `isSubsetOf(_:)` для определения является ли множество подмножеством другого множества.
- Используйте метод `isSupersetOf(_:)` для определения является ли множество надмножеством другого множества.
- Используйте методы `isStrictSubsetOf(_:)` или `isStrictSupersetOf(_:)` для определения является ли множество подмножеством или надмножеством, но не равным указанному множеству.
- Используйте метод `isDisjointWith(_:)` для определения наличия общих значений в двух множествах.

```
let houseAnimals: Set = [" ", " "]
let farmAnimals: Set = [" ", " ", " ", " ", " ", " "]
let cityAnimals: Set = [" ", " "]

houseAnimals.isSubsetOf(farmAnimals)
// true
farmAnimals.isSupersetOf(houseAnimals)
// true
farmAnimals.isDisjointWith(cityAnimals)
// true
```

Словари

Словарь представляет собой контейнер, который хранит несколько значений одного и того же типа. Каждое значение связано с уникальным ключом, который выступает в качестве идентификатора этого значения внутри словаря. В отличие от элементов в массиве, элементы в словаре не имеют определенного порядка. Используйте словарь, когда вам нужно искать значения на основе их идентификатора, так же как в реальном мире словарь используется для поиска определения конкретного слова.

Заметка

Словари в Swift конкретны по отношению к типам ключей и значений которые они могут хранить. Они отличаются от классов `NSDictionary` и `NSMutableDictionary` из Objective-C, которые могут использовать любой объект в качестве ключей и значений, и не предоставляют никакой информации о характере этого объекта. В Swift, всегда понятно какие типы ключей и значений может хранить определенной словарь, либо через явное объявление типа, либо через вывод типов.

Сокращенный синтаксис словаря

В Swift тип словаря в полной форме пишется как `Dictionary<Key, Value>`, где `Key` это тип значения который используется как ключ словаря, а `Value` это тип значения который словарь может хранить для этих ключей.

Вы можете также написать словарь в сокращенной форме как `[Key: Value]`. Хотя две формы функционально идентичны, краткая форма является предпочтительной и используется в данном руководстве при обращении к типу словаря.

Литералы словаря

Вы можете инициализировать словарь с помощью литерала словаря, который имеет схожий синтаксис с литералом массива, рассмотренного ранее. Литерал словаря это краткий способ написать одну или несколько пар ключ-значение в виде коллекций словаря.

Пара ключ-значение является комбинацией ключа и значения. В литерале словаря, ключ и значение в каждой паре ключ-значение разделено двоеточием. Пары ключ-значение написаны как список, разделенный запятыми и окруженный парой квадратных скобок:

```
[ [ключ 1]: значение 1, [ключ 2]: значение 2,
  [ключ 3]: значение 3 ]
```

В примере ниже создается словарь, который хранит имена международных аэропортов. В этом словаре ключи являются трехбуквенным кодом международной ассоциации воздушного транспорта, а значения - названия аэропортов:

```
1. var airports: [String: String] = ["YYZ": "Toronto  
Pearson", "DUB": "Dublin"]
```

Словарь `airports` объявлен с типом `[String: String]`, что означает "словарь ключи которого имеют тип `String` и значения которого также имеют тип `String`".

Заметка

Словарь `airports` объявлен как переменная (с помощью `var`), а не константа (с помощью `let`), поскольку много аэропортов будет добавляться к словарю в примерах ниже.

Словарь `airports` инициализирован с помощью литерала словаря, содержащего две пары ключ-значение. Первая пара имеет ключ `"YYZ"` и значение `"Toronto Pearson"`. Вторая пара имеет ключ `"DUB"` и значение `"Dublin"`.

Этот словарь содержит две пары `String: String`. Этот тип ключ-значение подходит типу который мы присвоили переменной `airports` (словарь содержащий только `String` ключи, и только `String` значения), и поэтому присвоение литерала словаря допустимо в качестве способа инициализации словаря `airports` двумя начальным элементами.

Подобно к массивам, вы можете не писать тип словаря если вы инициализируете его с помощью литерала словаря, чьи ключи и значения имеют соответствующие типы. Инициализация `airports` может быть записана в более краткой форме:

```
1. var airports = ["YYZ": "Toronto  
Pearson", "DUB": "Dublin"]
```

Поскольку все ключи в литерале имеют одинаковый тип, и точно так же все значения имеют одинаковый тип, то Swift может вывести, что `[String: String]` является правильным типом для использования в словаре `airports`.

Доступ и изменение словаря

Вы можете получить доступ к словарю и изменять его либо через его методы и свойства, либо используя синтаксис индексов. Подобно массивам, вы можете узнать количество элементов в словаре через его read-only свойство `count`:

```
1. println("The airports dictionary  
contains \(airports.count) items.")  
2. // напечатает "The airports dictionary contains 2  
items."
```

Логическое свойство `isEmpty` можно использовать в качестве быстрого способа узнать, является ли свойство `count` равным 0:

```
1. if airports.isEmpty {  
2.     println("The airports dictionary is empty.")  
3. } else {  
4.     println("The airports dictionary is not empty.")  
5. }  
6. // напечатает "The airports dictionary is not  
empty."
```

Вы можете добавить новый элемент в словарь с помощью синтаксиса индексов. Используйте новый ключ

соответствующего типа в качестве индекса, а затем присвойте новое значение соответствующего типа:

```
1. airports["LHR"] = "London"
2. // словарь airports теперь содержит 3 элемента
```

Вы также можете использовать синтаксис индексов для изменения значения связанного с определенным ключом:

```
1. airports["LHR"] = "London Heathrow"
2. // значение для "LHR" поменялось на "London Heathrow"
```

В качестве альтернативы индексам, можно использовать метод словаря `updateValue(forKey:)`, чтобы установить или обновить значение для определенного ключа. Подобно примерам с индексами вверху, метод `updateValue(forKey:)` устанавливает значение для ключа если оно не существует, или обновляет значение, если этот ключ уже существует. Однако, в отличие от индексов, метод `updateValue(forKey:)` возвращает старое значение после выполнения обновления. Это позволяет вам проверить состоялось ли обновление или нет.

Метод `updateValue(forKey:)` возвращает опциональное значение соответствующее типу значения словаря. Например, для словаря, который хранит `String` значения, метод возвратит `String?` тип, или " опциональный `String`". Это опциональное значение содержит старое значение для этого ключа, если оно существовало до обновления, либо `nil` если значение не существовало.

```

1. if let oldValue = airports.updateValue("Dublin
   Airport", forKey: "DUB") {
2. println("The old value for DUB was \(oldValue).")
3. }
4. // напечатает "The old value for DUB was Dublin."

```

Вы также можете использовать синтаксис индексов чтобы получить значение из словаря для конкретного ключа. Поскольку есть вероятность запросить ключ для несуществующего значения, индекс словаря возвращает опциональное значение соответствующее типу значений словаря. Если словарь содержит значение для запрошенного ключа, индекс возвращает опциональное значение, содержащее существующее значение для этого ключа. В противном случае индекс возвращает `nil`:

```

1. if let airportName = airports["DUB"] {
2. println("The name of the airport is \(airportName).")
3. } else {
4. println("That airport is not in the airports
   dictionary.")
5. }
6. // напечатает "The name of the airport is Dublin
   Airport."

```

Вы можете использовать синтаксис индексов для удаления пары ключ-значение из словаря путем присвоения `nil` значению для этого ключа:

```

1. airports["APL"] = "Apple International"
2. // "Apple International" несуществующий аэропорт
   для APL, так что удалим его
3. airports["APL"] = nil
4. // APL теперь был удален из словаря"

```

Кроме того, можно удалить пару ключ-значение из словаря с помощью метода `removeValueForKey`. Этот метод удаляет пару ключ-значение если она существует и затем возвращает значение, либо возвращает `nil` если значения не существует:

```
1. if let removedValue = airports.removeValueForKey("DUB")
2. {
3.     println("The removed airport's name
4.     is \(removedValue).")
5. } else {
6.     println("The airports dictionary does not contain a
7.     value for DUB.")
8. }
9. // напечатает "The removed airport's name is Dublin
10. Airport."
```

Итерация по словарю

Вы можете сделать итерацию по парам ключ-значение в словаре с помощью `for-in` цикла. Каждое значение в словаре возвращается как кортеж (*ключ, значение*), и вы можете разложить части кортежа по временным константам или переменным в рамках итерации:

```
1. for (airportCode, airportName) in airports {
2.     println("\(airportCode) : \(airportName)")
3. }
4. // LHR: London Heathrow
5. // YYZ: Toronto Pearson
```

Чтобы подробнее узнать про цикл `for-in`, смотрите главу "Циклы for"

Вы также можете получить коллекцию ключей или значений словаря через обращение к его свойствам `keys` и `values`:


```

1. for airportCode in airports.keys {
2.     println("Airport code: \(airportCode)")
3. }
4. // Airport code: LHR
5. // Airport code: YYZ
6.
7. for airportName in airports.values {
8.     println("Airport name: \(airportName)")
9. }
10. // Airport name: London Heathrow
11. // Airport name: Toronto Pearson

```

Если вам нужно использовать ключи или значения словаря вместе с каким-либо API, которое принимает объект `Array`, то можно инициализировать новый массив с помощью свойств `keys` и `values`:

```

1. let airportCodes = [String](airports.keys)
2. // airportCodes теперь ["YYZ", "LHR"]
3.
4. let airportNames = [String](airports.values)
5. // airportNames теперь ["Toronto Pearson", "London Heathrow"]

```

Заметка

Тип словарь в Swift является неупорядоченной коллекцией. Порядок получения ключей, значений и пар ключ-значение при итерации по словарю, не уточняется.

Создание пустого словаря

Подобно массивам вы можете создать пустой словарь определенного типа с помощью синтаксиса инициализатора:

```
1. var namesOfIntegers = [Int: String]()
2. // namesOfIntegers является пустым [Int: String]
   словарем
```

В этом примере создается пустой словарь с типом `[Int: String]` для хранения удобных для восприятия имен числовых значений. Его ключи имеют тип `Int`, а значения - `String`.

Если контекст уже предоставляет информацию о типе, вы можете создать пустой словарь с помощью литерала пустого словаря, который пишется как `[:]` (двоеточие внутри пары квадратных скобок):

```
1. namesOfIntegers[16] = "sixteen"
2. // namesOfIntegers теперь содержит 1 пару ключ-
   значение
3. namesOfIntegers = [:]
4. // namesOfIntegers теперь опять пустой словарь с
   типом [Int: String]
```

Хеш значения для типов ключей словаря

Тип должен быть хешируемым, чтобы можно было использовать его как тип ключа словаря, то есть тип должен предоставить возможность вычислять хеш-значение для самого себя. Хеш-значение - это значение типа `Int`, которое является одинаковым для всех объектов, которые проверяют равенство, такое, что если `a == b`, то и `a.hashCode == b.hashCode`.

В Swift все простые типы (такие как `String`, `Int`, `Double`, и `Bool`) являются хешируемыми по умолчанию, так что все эти типы могут быть использованы как ключи для словаря. Элементы перечислений без связанных значений (как описано в главе "Перечисления"), также хешируемы по умолчанию.

Заметка

Вы можете использовать свои собственные пользовательские типы в качестве типов для ключей словаря путем приведения их в соответствие протоколу `Hashable` из стандартной библиотеки Swift. Типы которые соответствуют протоколу `Hashable` должны предусматривать `gettable Int` свойство с именем `hashValue`, и должны предусматривать реализацию оператора "равно" (`==`). Значение возвращаемое от свойства `hashValue` не обязательно должно быть одинаковое при разных запусках одной и той же программы, и даже при запусках различных программ.

Для получения дополнительной информации о соответствии с протоколами, смотрите главу "Протоколы".

Управление потоком

В Swift есть все знакомые нам операторы управления потоком из C-подобных языков. К ним относятся: циклы `for` и `while` для многократного выполнения задач, операторы `if` и `switch` для выполнения различных ветвлений кода в зависимости от определенных условий, а также такие операторы, как `break` и `continue` для перемещения потока выполнения в другую точку вашего кода.

В дополнение к традиционному циклу `for` из C, Swift добавляет цикл `for-in`, который упрощает итерацию по массивам, словарям, диапазонам, строкам и другим последовательностям.

В Swift оператор `switch` также намного мощнее, чем его аналог из языка C. В Swift не происходит проваливания к следующему кейсу, что позволяет избежать распространенную в C ошибку, связанную с пропуском оператора `break`. Кейсы могут сопоставлять различные типы шаблонов, включая сопоставление диапазонов, кортежей, а также выполнять приведение к определенному типу. Совпавшие значения в кейсе оператора `switch` могут быть привязаны к временной константе или переменной для использования в теле кейса, а сложные условия сравнения могут быть выражены с помощью `where` для каждого кейса.

Циклы For

Swift предлагает два вида цикла, которые выполняют набор инструкций определенное количество раз:

- Цикл `for-in` выполняет набор инструкций для каждого элемента диапазона, последовательности, коллекции или прогрессии.
- Цикл `for` выполняет набор инструкций до тех пор, пока не будет выполнено определенное условие. Как правило, это достигается путем инкрементирования счетчика в конце каждой итерации цикла.

Цикл For-in

Цикл `for-in` используется для итерации по коллекциям элементов, таких как диапазоны чисел, элементы массива или символы в строке.

Этот пример напечатает несколько записей таблицы умножения на 5:

```
1. for index in 1...5 {
2.     println("\(index) times 5 is \(index * 5)")
3. }
4. // 1 умножить на 5 будет 5
5. // 2 умножить на 5 будет 10
6. // 3 умножить на 5 будет 15
7. // 4 умножить на 5 будет 20
8. // 5 умножить на 5 будет 25
```

Коллекция элементов по которой происходит итерация, является закрытым диапазоном чисел от 1 до 5 включительно, так как используется оператор закрытого диапазона (`...`). Значение `index` устанавливается в первое число из

диапазона (1), и выражение внутри цикла выполняются. В данном случае, цикл содержит только одно выражение, которое печатает запись из таблицы умножения на пять для текущего значения `index`. После того как выражение выполнено, значение `index` обновляется до следующего значения диапазона (2), и функция `println` снова вызывается. Этот процесс будет продолжаться до тех пор, пока не будет достигнут конец диапазона.

В примере выше `index` является константой, значение которой автоматически устанавливается в начале каждой итерации цикла. Как таковую, ее не нужно объявлять перед использованием. Ее объявление неявно происходит в объявлении цикла, без необходимости использования зарезервированного слова `let`.

Если Вам не нужно каждое значение из диапазона, то вы можете игнорировать их, используя символ подчёркивания вместо имени переменной:

```
1. let base = 3
2. let power = 10
3. var answer = 1
4. for _ in 1...power {
5.     answer *= base
6. }
7. println("\ (base) to the power
   of \ (power) is \ (answer)")
8. // напечатает "3 to the power of 10 is 59049"
```

В этом примере вычисляется значение одного числа возведенное в степень другим (в данном случае 3 в степени 10). Начальное значение 1 (то есть 3 в степени 0) умножается на 3 десять раз, используя закрытый диапазон значений, который начинается с 1, и заканчивается 10. В данном случае нет необходимости знать значения счётчика во время каждой итерации цикла - он просто должен

выполниться необходимое количество раз. Символ подчёркивания `"_"` (который используется вместо переменной цикла) игнорирует ее отдельные значения и не предоставляет доступ к текущему значению во время каждой итерации цикла.

Можно использовать цикл `for-in` вместе с массивом для итерации по его элементам:

```
1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. for name in names {
3.   println("Hello, \(name)!")
4. }
5. // Hello, Anna!
6. // Hello, Alex!
7. // Hello, Brian!
8. // Hello, Jack!
```

Таким же образом вы можете производить итерацию по словарю, чтобы получить доступ к его паре ключ-значение. Когда происходит итерация словаря, каждый его элемент возвращается как кортеж (`ключ, значение`). Вы можете разложить члены кортежа на отдельные константы для того, чтобы использовать их в теле цикла `for-in`. Здесь ключи словаря разлагаются в константу `animalName`, а его значения — в константу `legCount`:

```
1. let numberOfLegs =
   ["spider": 8, "ant": 6, "cat": 4]
2. for (animalName, legCount) in numberOfLegs {
3.   println("\(animalName)s have \(legCount) legs")
4. }
5. // ants have 6 legs
6. // cats have 4 legs
7. // spiders have 8 legs
```

Итерация по элементам словаря не обязательно будет происходить в том же порядке, в котором они были вставлены. Содержимое словаря по сути своей не является упорядоченным, поэтому и извлекаемые из него значения во время итерации тоже могут быть не упорядочены. Более подробно о массивах и словарях смотрите в главе Типы Коллекций.

Помимо массивов и словарей, вы также можете использовать цикл `for-in` для перебора `Character` символов в строке:

```
1. for character in "Hello" {
2.     println(character)
3. }
4. // H
5. // e
6. // l
7. // l
8. // o
```

Цикл For

В дополнение к циклу `for-in`, Swift поддерживает традиционный цикл `for` с условием и инкрементом:

```
1. for var index = 0; index < 3; ++index {
2.     println("index is \(index)")
3. }
4. // index is 0
5. // index is 1
6. // index is 2
```


В общем виде цикл `for` выглядит следующим образом:

```
1. for [инициализация] ; [условие] ; [инкремент] {  
2.   [выражения]  
3. }
```

Как и в языке C, цикл разделяется на три части с помощью точки с запятой. Однако, в отличие от C, Swift не требует обрамлять круглыми скобками блок "инициализация; условие; инкремент".

Выполнение цикла происходит следующим образом:

1. Во время первого входа в цикл происходит однократное определение инициализатора для того, чтобы установить все переменные и константы, необходимые при выполнении цикла.
2. Происходит проверка выполнения условия. Если условие не выполняется (результат равен `false`), то работа цикла прерывается и происходит выполнение кода, находящегося за закрывающей фигурной скобкой (`}`). Если же результат условия равен `true`, то исполняется код, расположенный внутри фигурных скобок.
3. После того как все операции в теле цикла будут завершены, происходит выполнение выражения инкремента. Оно может увеличить или уменьшить значение счетчика, либо изменить значение одной из инициализированных переменных на основании результата выражений, выполненных в теле цикла. После того, как выражение инкремента исполнится, выполнение цикла возвращается к шагу 2, где снова проверяется условное выражение.

Формат цикла и процесс его выполнения, описанный выше, является краткой формой записи:

```
1.  инициализация
2.  while условие {
3.  операторы
4.  инкремент
5.  }
```

Константы и переменные, объявленные в выражении инициализации (такие как `var index = 0`) действительны только в пределах самого цикла `for`. Чтобы получить доступ к последнему значению переменной `index`, вы должны объявить ее до начала цикла.

```
1.  var index: Int
2.  for index = 0; index < 3; ++index {
3.  println("index is \(index)")
4.  }
5.  // index is 0
6.  // index is 1
7.  // index is 2
8.  println("The loop statements were
   executed \(index) times")
9.  // напечатает "The loop statements were executed 3
   times"
```

Обратите внимание, что последнее значение переменной `index` после завершения цикла равно 3, а не 2. Во время последней итерации цикла, выражение `++index` присваивает переменной `index` значение, равное 3, после чего условие `index < 3` не выполняется и цикл завершается.

Циклы While

Цикл `while` выполняет набор инструкций до тех пор, пока его условие не станет ложным. Этот вид циклов лучше всего использовать в тех случаях, когда количество итераций до первого входа в цикл неизвестно. Swift предлагает два вида циклов `while`:

- `while` - вычисляет условие выполнения в начале каждой итерации цикла.
- `do-while` - вычисляет условие выполнения в конце каждой итерации цикла.

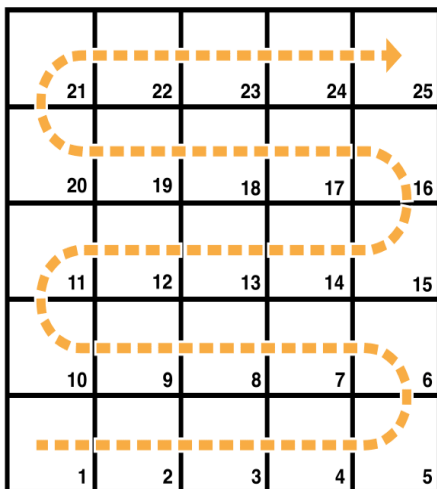
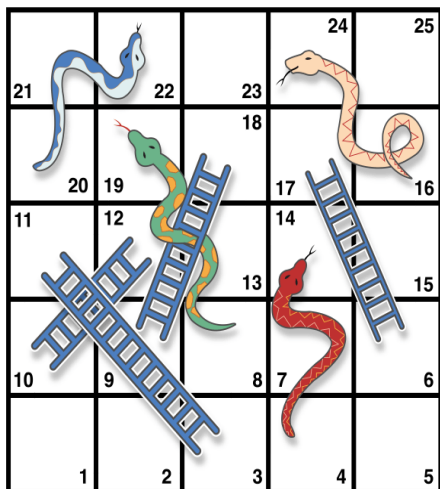
While

Цикл `while` начинается с вычисления условия. Если условие истинно, то инструкции в теле цикла будут выполняться до тех пор, пока оно не станет ложным.

Общий вид цикла `while` выглядит следующим образом:

```
1. while условие {  
2. инструкции  
3. }
```

В этом примере показана простая игра Змеи и Лестницы (также известная, как Горы и Лестницы):



Игра проходит по следующим правилам:

- Доска разделена на 25 квадратов и цель состоит в том, чтобы стать на 25-ый квадрат или за его пределами.
- В начале каждого хода Вы бросаете игральную кость и перемещаетесь на то число шагов, которое выпало после броска, в направлении, которое указывает пунктирная стрелка.
- Если ваш ход заканчивается на основании лестницы, то вы поднимаетесь по ней вверх.
- Если ваш ход заканчивается на голове змеи, то вы спускаетесь вниз по этой змее.

Игровая доска в примере представлена массивом значений типа `Int`. Его размер хранится в константе `finalSquare`, которая используется как для инициализации массива, так и для проверки условия победы. Игровое поле инициализируется 26-ю, а не 25-ю целочисленными нулевыми значениями(каждое с индексом от 0 до 25 включительно):

```

1. let finalSquare = 25
2. var board =
    [Int](count: finalSquare + 1, repeatedValue: 0)

```

Затем, для обозначения лестниц и змей, некоторым квадратам присваиваются специальные значения. Квадраты с основанием лестницы, перемещающие вас вверх по доске, имеют положительные значения, тогда как квадраты с головой змеи, спускающие вас вниз - отрицательное.

```

1. board[03] = +08; board[06] = +11; board[09] =
  +09; board[10] = +02
2. board[14] = -10; board[19] = -11; board[22] = -
  02; board[24] = -08

```

Квадрат 3 с основанием лестницы перемещает вас вверх на 11 квадрат. Чтобы это сделать, элементу массива `board[03]` присваивается `+08`, что эквивалентно значению 8 типа `Int` (разница между 3 и 11). Для того чтобы уточнить формулировку игрового поля, оператор унарного плюса (`+`) уравнивает оператор унарного минуса (`-`), а числа ниже 10 приписаны нули. (В этих двух стилистических надстройках нет прямой необходимости, но они делают код более читаемым).

Все участники начинают игру с нулевого квадрата, который находится у левого нижнего края доски. Первый бросок кубика всегда перемещает участника на игровую доску:

```

1. var square = 0
2. var diceRoll = 0
3. while square < finalSquare {
4.   // бросок кубика
5.   if ++diceRoll == 7 { diceRoll = 1 }
6.   // начать ходить на выпавшее количество шагов
7.   square += diceRoll
8.   if square < board.count {

```

```

9. // если мы все еще на поле, идти вверх или вниз по
   змеям или лестницам
10.     square += board[square]
11.     }
12.     }
13.     println("Game over!")

```

Данный пример использует самый простой подход к реализации броска кубика. Вместо использования генератора случайных чисел, значение `diceRoll` начинается с 0. Каждую итерацию цикла переменная `diceRoll` увеличивается на 1 с помощью префиксного оператора инкремента (`++i`), после чего проверяется не стало ли её значение слишком большим. Возвращаемое значение `++diceRoll` равно значению переменной `diceRoll` после её инкрементирования. Когда это значение становится равным 7, оно сбрасывается на 1. В итоге мы получаем последовательность значений `diceRoll`, которая всегда будет выглядеть следующим образом: 1, 2, 3, 4, 5, 6, 1, 2 и так далее.

После броска кубика игрок перемещается вперед на количество клеток, равное значению переменной `diceRoll`. Возможен случай, когда бросок кубика может переместить игрока за пределы квадрата 25. В таком случае игра заканчивается. Для того чтобы справиться с таким сценарием, код проверяет что значение `square` меньше чем свойство `count` массива `board` перед прибавлением значения, хранящегося в `board[square]` к текущему значению `square` для перемещения игрока вверх или вниз по змеям или лестницам.

Если бы этой проверки не было, могла бы произойти попытка обращения к значению `board[square]`, находящемуся за границами массива `board`, что привело бы к вызову ошибки. Если `square` равно 26, код попытается проверить значение `board[26]`, которое выходит за границы массива.

Текущая итерация цикла заканчивается, после чего проверяется условие цикла, для того чтобы понять нужно ли переходить к следующей итерации. Если игрок переместился на квадрат 25 или за его пределы, значение условия будет вычислено как `false` и игра закончится.

В данном случае использование `while` является наиболее подходящим, так как продолжительность игры неизвестна перед началом цикла. Цикл просто выполняется до тех пор, пока не будет выполнено конкретное условие.

Цикл `do-while`

Другой вариант цикла `while`, известный как цикл `do-while`, выполняет одну итерацию до того, как происходит проверка условия. Затем цикл продолжает повторяться до тех пор, пока условие не станет ложным.

Общий вид цикла `do-while` выглядит следующим образом:

```
1. do {  
2.   инструкции  
3. } while условие
```

Ниже снова представлен пример игры Змеи и Лестницы, написанный с использованием цикла `do-while`. Значения переменных `finalSquare`, `board`, `square` и `diceRoll` инициализированы точно таким же образом, как и в случае с циклом `while`:

```

1. let finalSquare = 25
2. var board =
   [Int](count: finalSquare + 1, repeatedValue: 0)
3. board[03] = +08; board[06] = +11; board[09] =
   +09; board[10] = +02
4. board[14] = -10; board[19] = -11; board[22] = -
   02; board[24] = -08
5. var square = 0
6. var diceRoll = 0

```

В этой версии игры в начале цикла происходит проверка на наличие змей или лестниц на квадрате. Ни одна лестница на поле не приведет игрока на квадрат 25. Таким образом невозможно победить в игре, переместившись вверх по лестнице. Следовательно, такая проверка в самом начале цикла является абсолютно безопасной.

В начале игры игрок находится на квадрате 0. `board[0]` всегда равняется 0 и не оказывает никакого влияния:

```

1. do {
2. // идти вверх или вниз по змеям или лестницам
3. square += board[square]
4. // бросить кубик
5. if ++diceRoll == 7 { diceRoll = 1 }
6. // начать ходить на выпавшее количество шагов
7. square += diceRoll
8. } while square < finalSquare
9. println("Game over!")

```

После проверки на наличие змей и лестниц происходит бросок кубика и игрок продвигается вперед на количество квадратов, равное `diceRoll`. После этого текущая итерация цикла заканчивается.

Условие цикла (`while square < finalSquare`) такое же, как раньше, но в этот раз оно не вычисляется до окончания первого запуска цикла. Структура цикла `do-while` лучше подходит для этой игры, чем цикл `while` в предыдущем примере. В цикле `do-while` выше `square += board[square]` всегда выполняется сразу, в то время как в цикле `while` происходит проверка того, что `square` все еще находится на поле. Такой принцип работы цикла `do-while` снимает необходимость проверки выхода за границы массива, которую мы видели в предыдущей версии игры.

Условные операторы

Иногда бывает полезным исполнять различные куски кода в зависимости от условий. А может быть вы хотите запустить исполнение дополнительного кода, в случае возникновения ошибки или просто показать сообщение, когда значение какой-либо величины становится слишком большим. Чтобы сделать это, вы делаете ваш код условным.

Swift предоставляет нам два варианта добавить условные ответвления кода - это при помощи оператора `if` и при помощи оператора `switch`. Обычно мы используем оператор `if`, если наше условие достаточно простое и предусматривает всего несколько вариантов. А вот оператор `switch` подходит для более сложных условий, с множественными перестановками, и очень полезна в ситуациях, где по найденному совпадению с условием и выбирается соответствующая ветка кода для исполнения.

Оператор if

В самой простой своей форме оператор `if` имеет всего одно условие `if`. Этот оператор выполняет установленные инструкции только в случае, когда условие `true`:

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    println ("It's very cold. Consider
wearing a scarf.")
}
// выведет "It's very cold. Consider wearing a
scarf."
```

В приведенном примере проверяется значение температуры, которая может быть ниже или 32 (0 по Цельсию) градусов по Фаренгейту либо равна или выше. Если она ниже, но выведется сообщение. В противном случае никакого сообщения не будет, и код продолжит свое выполнение после закрывающей фигурной скобки оператора `if`.

Оператор `if` может предусматривать еще один дополнительный набор инструкций в ветке известной как оговорка `else`, которая нужна в случае, если условие `false`. Эти инструкции указываются через ключевое слово `else`:

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    println ("It's very cold. Consider wearing
a scarf.")
} else {
    println ("It's not that cold. Wear a t-
shirt.")
}
```

```
// выведет "It's not that cold. Wear a t-shirt."
```

В этом коде всегда будет выполняться код либо в первом, либо во втором ответвлении. Из-за того что температура выросла до 40 градусов Фаренгейта, значит больше не обязательно носить шарф, таким образом ответвление `else` выполняется.

Вы можете соединять операторы `if` между собой, чтобы создать более сложные условия:

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing
a scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to
wear sunscreen.")
} else {
    println("It's not that cold. Wear a t-
shirt.")
}
// выведет "It's really warm. Don't forget to
wear sunscreen."
```

В приведенном коде было добавлено дополнительное условие `if`, для соответствия определенным температурам. Конечное условие `else` соответствует всем температурам, не соответствующим первым двум условиям.

Последняя `else` опциональна, однако она может быть удалена только в случае, если в ней нет необходимости:

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing
a scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to
wear sunscreen.")
}
```

В этом примере температура ни высокая, ни низкая, и вообще она не соответствует ни одному условию, так что никакого сообщения мы не увидим.

Оператор `switch`

Оператор `switch` подразумевает наличие какого-то значения, которое сравнивается с несколькими возможными шаблонами. После того как значение совпало с каким-либо шаблоном, выполняется код, соответствующий ответвлению этого шаблона, и больше сравнения уже не происходит. `Switch` представляет собой альтернативу оператору `if`, отвечающей нескольким потенциальным значениям.

В самой простой форме в оператор `switch` значение сравнивается с одним или более значений того же типа:

```
switch значение для сопоставления {  
    case значение 1:  
        инструкция для значения 1  
    case значение 2, значение 3:  
        инструкция для значения 2 или  
        значения 3  
    default:  
        инструкция, если совпадений с  
        шаблонами не найдено  
}
```

Каждый оператор `switch` состоит из нескольких возможных случаев или `cases`, каждый из которых начинается с ключевого слова `case`. Помимо сравнения с конкретными значениями, Swift предлагает еще несколько опций для каждого случая для создания более сложных шаблонных сравнений. Об этих опциях мы поговорим далее в этой главе.

Тела каждого отдельного блока `case` в `switch` - это отдельная ветка исполнительного кода, что делает `switch` похожим на оператор `if`. Оператор `switch` определяет какое ответвление должно быть выбрано. Это известно как переключение на значение, которое в настоящее время рассматривается.

Каждый оператор `switch` должен быть исчерпывающим. То есть это значит, что каждое значение обязательно должно находить совпадение с шаблоном в каком-либо случае (`case`). Если неудобно вписывать все возможные варианты случаев, то вы можете определить случай по умолчанию, который включает в себя все значения, которые не были включены в остальные случаи. Такой случай по умолчанию

называется `default`, и он всегда идет после всех остальных случаев.

В следующем примере `switch` рассматривает единственный символ в нижнем регистре, который называется `someCharacter`:

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    println("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k",
    "l", "m",
    "n", "p", "q", "r", "s", "t", "v", "w", "x",
    "y", "z":
    println("\(someCharacter) is a consonant")
default:
    println("\(someCharacter) is not a vowel
or a consonant")
}
// "e is a vowel"
```

Первый случай в `switch` включает в себя все гласные английского алфавита в нижнем регистре. Аналогично, второй случай включает в себя все согласные английского алфавита в нижнем регистре.

Вообще не практично описывать все возможные варианты символов как часть оператора `switch`, поэтому и используют `default`, который включает в себя все остальные варианты, которые не относятся к гласным или согласным английского алфавита. Именно такое положение и гарантирует, что оператор `switch` исчерпывающий.

Отсутствие case-провалов

Большое отличие оператора `switch` в языке Swift от оператора `switch` в C и Objective-C составляет отсутствие провалов через условия. Вместо этого оператор `switch` прекращает выполнение после нахождения первого соответствия с `case` и выполнения соответствующего кода в ветке, без необходимости явного вызова `break`. Это делает оператор `switch` более безопасным и простым для использования, чем в C, и исключает исполнение кода более чем одного случая.

Заметка

Хотя `break` не требуются в Swift, вы все равно можете его использовать для соответствия и для игнорирования конкретного случая или просто для выхода из конкретного случая, еще до того, как исполнится код.

Тело каждого случая должно включать в себя хотя бы одно исполняемое выражение. Код не будет исполнен и выдаст ошибку компиляции, если написать его следующим образом:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a":
case "A":
    println("The letter A")
default:
    println("Not the letter A")
}
// ошибка компиляции
```

В отличие от оператора `switch` в языке C, `switch` в Swift не соответствует ни "a", ни "A". Но зато вы получите ошибку компиляции о том, что `case "a":` не содержит ни одного

исполняемого выражения. Такой подход исключает случайные "проваливания" из одного случая в другой, что делает код безопаснее и чище своей краткостью.

Множественные совпадения с одним случаем `switch` могут быть разграничены запятыми и могут быть записаны в несколько строк, если перечень шаблонов достаточно длинный:

```
switch значение для сопоставления {  
    case значение 1,  
        значение 2:  
        инструкции  
}
```

Заметка

Для того, чтобы у вас появилась возможность проваливаться через случай в конкретном `switch` случае, вы можете использовать ключевое слово `fallthrough`, которое описано в следующих главах.

Соответствие диапазону

Значения в случаях `switch` могут быть проверены на их вхождение в диапазон. Пример ниже использует целочисленные диапазоны для описания любых значений художественным языком:

```
let count = 3000000000000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
case 100...999:
    naturalCount = "hundreds of"
case 1000...999999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
println("There are \(naturalCount)
\ (countedThings).")
// выведет "There are millions and millions
of stars in the Milky Way."
```

Заметка

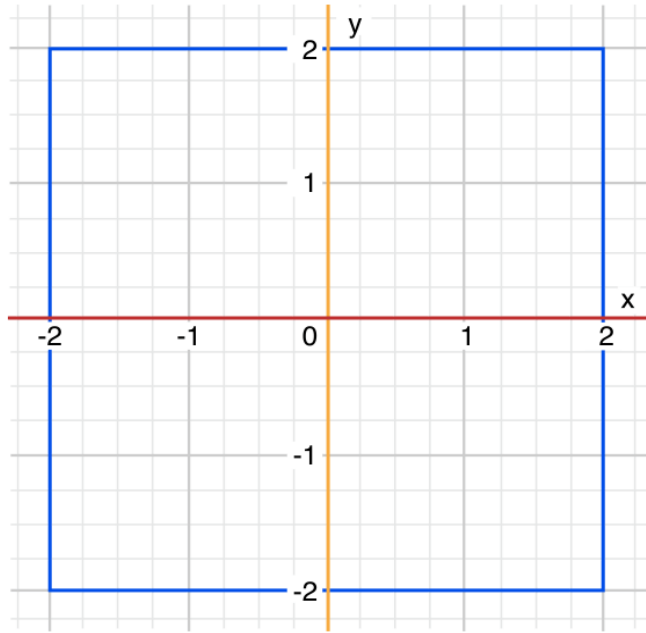
Функции и оператора закрытого диапазона (`...`) и оператор полу-открытого диапазона (`..<`) перегружены для возвращения типы `IntervalType` или `Range`. Диапазон может определить, содержит ли он определенный элемент,

например при сопоставлении `case` оператору `switch`. Диапазон является коллекцией последовательных значений, которые могут итерироваться в операторе `for-in`.

Кортежи

Вы можете использовать кортежи для тестирования нескольких значений в одном и том же операторе `switch`. Каждый элемент кортежа может быть протестирован с любой величиной или с диапазоном величин. Так же вы можете использовать идентификатор подчеркивания (`_`) для соответствия любой возможной величине. Пример ниже берет точку с координатами `(x, y)`, выраженную в виде кортежа `(Int, Int)` и относит к соответствующей категории как следует из примера ниже:

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    println("(0, 0) is at the origin")
case (_, 0):
    println("(\"(somePoint.0)\", 0) is on the x-
axis")
case (0, _):
    println("(0, \"(somePoint.1)\") is on the y-
axis")
case (-2...2, -2...2):
    println("(\"(somePoint.0)\", \"(somePoint.1)\")
is inside the box")
default:
    println("(\"(somePoint.0)\", \"(somePoint.1)\")
is outside of the box")
} // выведет "(1, 1) is inside the box"
```



Операторы **switch** определяет:

- Находится ли точка в начале отсчета.
- Находится ли она на оси x (красная)
- Находится ли она на оси y (оранжевая)
- Находится ли она внутри квадрата 4×4 клетки, в котором точка отсчета находится в центре или находится вне этого квадрата

В отличие от C, оператор `switch` в Swift позволяет множественное совпадение или пересечение значений нескольких случаев. Это факт, что точка $(0, 0)$ соответствует всем четырем условиям в этом примере. Однако, если возможно совпадение сразу с несколькими шаблонами, то в расчет принимается только первое из них. То есть точка $(0, 0)$

будет удовлетворять случаю `case (0, 0)`; а остальные случаи будут проигнорированы.

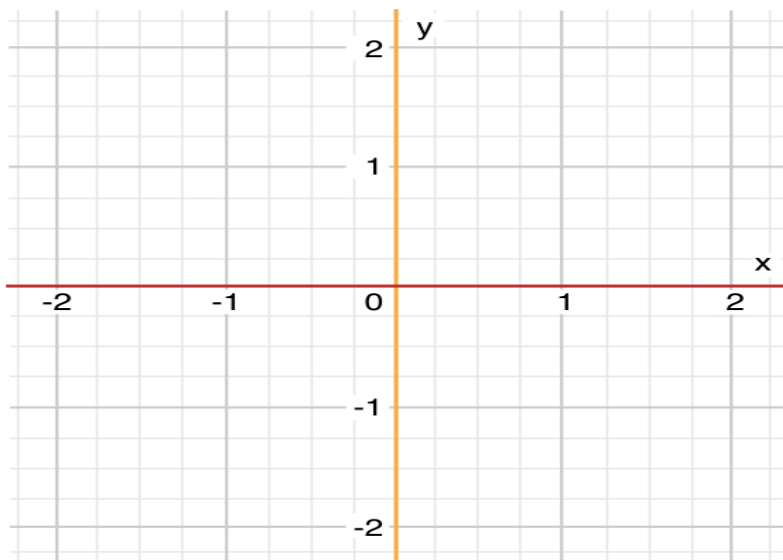
Привязка значений

Случай (`case`) в операторе `switch` может связывать значение или значения, с которыми сравнивается, с временными константами или переменными. Это известно как связывание величин, потому что значения "**связаны**" с временными константами или переменным внутри тела случая (`case`).

Пример ниже берет точку с координатами `(x, y)`, представленной в виде кортежа `(Int, Int)` и определяет ее позицию на графике, который представлен ниже:

```
let anotherPoint = (2, 0)

switch anotherPoint {
case (let x, 0):
    println("on the x-axis with an x value of
    \(x)")
case (0, let y):
    println("on the y-axis with a y value of
    \(y)")
case let (x, y):
    println("somewhere else at (\(x), \(y))")
}
// выведет "on the x-axis with an x value of 2"
```



Оператор `switch` определяет лежит ли точка на красной оси `x` или оранжевой оси `y`, а может быть она не будет ни на одной из осей.

Три случая в операторе `switch` объявляют константы `x`, `y`, которым временно присваиваются значения одного или обоих элементов из кортежа `anotherPoint`. В первом случае `case (let x, 0)`: подойдет любая точка со значением `y` равным `0`, а в константу `x` запишется значение координаты `x` нашей точки. Аналогично и во втором случае, когда `case (0, let y)`, этот случай включает все точки при значении их координаты `x` равной `0`, и происходит присваивание значения координаты `y` в временную константу `y`.

Объявленная константа может быть использована внутри блока случая (`case`). Здесь мы их используем как сокращенный вариант для вывода сообщения с помощью функции `println`.

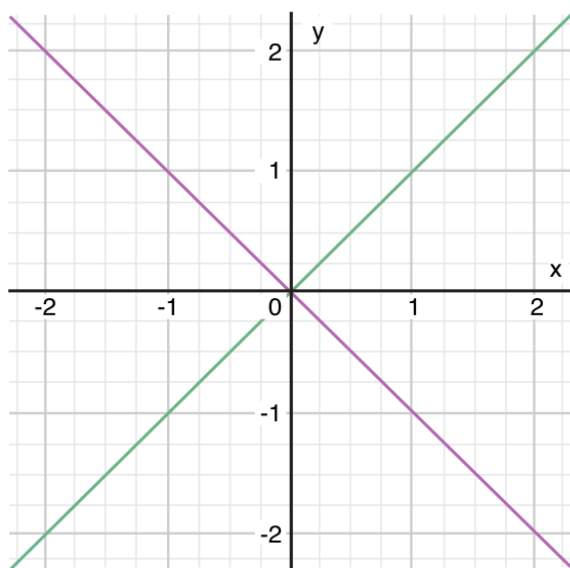
Заметьте, что оператор `switch` не имеет случая `default`. Последний шаблон `case let (x, y):`, где объявляются две временные константы, которые соответствуют абсолютно любой точке. И в качестве результата мы получаем все возможные оставшиеся значения, таким образом отпадает необходимость использования `default` в операторе `switch`.

В примере выше константы `x, y` объявлены с ключевым словом `let`, потому что у нас нет необходимости менять их значения в теле блока. Однако ничто нам не мешает объявить их и в качестве переменных, используя слово `var` вместо `let`. Если бы мы сделали так, то была бы создана временная переменная, которая была бы инициализирована соответствующим значением. Все изменения этой переменной имели бы силу так же только внутри тела блока случая (`case`).

Where в switch

В операторе **switch** мы также можем использовать дополнительное условие с помощью ключевого слова **where**.

Пример ниже размещает точку (x, y) на приведенном рисунке:



```

let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    println("(x), (y) is on the line x ==
y")
case let (x, y) where x == -y:
    println("(x), (y) is on the line x == -
y")
case let (x, y):
    println("(x), (y) is just some
arbitrary point")
}
// выведет "(1; -1) is on the line x == -y"

```

Оператор switch определяет лежит ли точка на зеленой диагонали, где $x == y$, или фиолетовой диагонали, где $x == -y$, или ни на одной и ни на другой.

Три случая объявляют константы x, y , которые временно берут значения из кортежа `yetAnotherPoint`. Эти константы используются как часть условия `where`, для создания динамического фильтра. Теперь при сравнении с шаблоном случая, так же учитывается и условие с `where`, удовлетворение которому и ведет значение сравнения к `true`.

Как и в предыдущем примере, последний случай включает в себя все возможные оставшиеся варианты, так что `default` тут так же не нужен, так как и без него оператор switch является исчерпывающим.

Операторы передачи управления

Операторы передачи управления меняют последовательность исполнения вашего кода, передавая управление от одного фрагмента кода другому. В Swift есть четыре оператора передачи управления:

- `continue`
- `break`
- `fallthrough`
- `return`

Операторы `continue`, `break`, `fallthrough` будут описаны в следующих главах. А оператор `return` будет описан в главе "Функции".

Оператор Continue

Оператор `continue` говорит циклу прекратить текущую итерацию и начать новую. Он как бы говорит: "Я закончил с текущей итерацией", но полностью из цикла не выходит.

Заметка

В цикле `for` с условием и инкрементом все продолжает работать как и работало, а именно условие проверяется, инкрементор меняется как и обычно, в общем цикл работает как обычно, только код внутри цикла пропускается.

Следующий пример убирает все гласные и пробелы из строки с нижним регистром, для того чтобы создать загадочную фразу-головоломку:

```

let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {switch character
{
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput.append(character)
}
}
print(puzzleOutput)
// напечатает "grtmndsthnlk"

```

Пример выше вызывает оператор `continue`, когда он находит соответствие с гласными звуками или пробелом, вызывая тем самым прекращение текущей итерации и начало новой. Такой подход позволяет блоку `switch` находить соответствие(и игнорировать) только гласные звуки и пробел, а не проверять каждую букву, которая может быть напечатана.

Оператор Break

Оператор `break` останавливает выполнение всей конструкции управления потоком. Оператор `break` может быть использован внутри конструкции `switch` или внутри цикла, когда вы хотите остановить дальнейшее выполнение `switch` или цикла раньше, чем он должен закончиться сам по себе.

Оператор Break в цикле

Когда оператор `break` используется внутри цикла, то `break` прекращает работу всего цикла немедленно, и выполнение кода продолжается с первой строки после закрывающей скобки цикла `{}`.

Оператор Break в конструкции Switch

Когда оператор `break` используется внутри конструкции `switch`, то он прекращает исполнение кода конкретного случая и перекидывает исполнение на первую строку после закрывающей скобки `}` конструкции `switch`.

Так же оператор `break` может использоваться для сопоставления или игнорирования случаев в конструкции `switch`. Так как конструкция `switch` исчерпывающая и не допускает пустых случаев, то иногда бывает необходимо умышленно соответствовать или игнорировать случаи для того, чтобы сделать ваши намерения ясными. Вы делаете это когда пишете слово `break` в теле случая, который вы хотите пропустить. Когда этот случай попадает под сравнение, то `break` сразу завершает работу всей конструкции `switch`.

Заметка

Случай (`case`) в конструкции `switch`, который содержит только комментарий, при компиляции выдаст ошибку компиляции. Комментарии - это не утверждения, и они не дают возможности игнорировать случаи. Если вы хотите игнорировать случай, то используйте `break`.

Следующий пример переключается на символьные значение `Character` и определяет, является ли символ целым числом на одном из четырех языков. Несколько языков включены в каждый случай (`case`) для краткости:

```
let numberSymbol: Character = "三" // Цифра 3
в упрощенном Китайском языке
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "\", "一", "๑":
    possibleIntegerValue = 1
case "2", "๒", "二", "๒":
    possibleIntegerValue = 2
case "3", "๓", "三", "๓":
    possibleIntegerValue = 3
case "4", "๔", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of
\ (numberSymbol) is \ (integerValue).")
} else {
    print("An integer value could not be found
for \ (numberSymbol).")
}
// напечатает "The integer value of 三 is 3."
```

Этот пример проверяет `numberSymbol` на наличие в нем целого числа от 1 до 4 на арабском, латинском, китайском или тайском языках. Если совпадение найдено, то один из случаев `switch` устанавливает опциональную

переменную `Int?`, названную `possibleIntegerValue` в подходящее целочисленное значение.

После того как конструкция `switch` выполнена, пример использует опциональную привязку для определения наличия величины. Переменная `possibleIntegerValue` имеет неявное начальное значение равное `nil` в силу того, что она опциональный тип, таким образом опциональная привязка пройдет успешно только в том случае, если `possibleIntegerValue` будет иметь актуальное значение одного из четырех первых случаев конструкции `switch`.

Это не практично перечислять каждое возможное значение `Character` в примере выше, таким образом случай `default` улавливает все остальные варианты символов, которые не соответствуют первым четырем случаям. Случаю `default` не надо предпринимать каких-либо действий, так что там прописан только оператор `break`. После того как срабатывает случай `default`, срабатывает `break`, что прекращает действие конструкции `switch` и код продолжает свою работу с `if let`.

Оператор `Fallthrough`

Конструкция `switch` в Swift не проваливается из каждого случая (`case`) в следующий. Напротив, как только находится соответствие с первым случаем, так сразу и прекращается работа всей конструкции. А в языке C, работа конструкции `switch` немного сложнее, так как требует явного прекращения работы при нахождении соответствия словом `break` в конце случая, в противном случае при соответствии мы провалимся в следующий случай и так далее пока не встретим слово `break`. Избежание провалов значит что конструкция `switch` в Swift более краткая и

предсказуемая, чем она же в C, так как она предотвращает срабатывание нескольких случаев по ошибке.

Если вам по какой-то причине нужно аналогичное проваливание как в C, то вы можете использовать оператор `fallthrough` в конкретном случае. Пример ниже использует `fallthrough` для текстового описания целого числа:

```
let integerToDescribe = 5
var description = "The number
\ (integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// выводит "The number 5 is a prime number,
and also an integer."
```

В примере мы объявляем новую переменную типу `String`, названную `description` и присваиваем ей исходное значение. Потом мы определяем величину `integerToDescribe`, используя конструкцию `switch`. Если значение `integerToDescribe` одно из значений списка случая, то мы получаем текстовое описание значение, которое дополняется значением, которое находится `default`, так как на уровень выше в сработавшем случае стоит ключевое слово `fallthrough`, после чего завершается работа конструкции `switch`.

Если значение `integerToDescribe` не принадлежит списку значений нашего единственного случая, то срабатывает

случай по умолчанию, который имеет все оставшиеся значения, не вошедшие в первый случай, и `integerToDescribe` получает значение только то, что есть в `default`.

После того как сработала конструкция `switch`, мы получаем описание числа, используя функцию `println()`. В нашем примере мы получаем `5`, что корректно определено как простое число.

Заметка

Ключевое слово `fallthrough` не проверяет условие случая, оно позволяет провалиться из конкретного случая в следующий или в `default`, что совпадает со стандартным поведением конструкции `switch` в языке C.

Маркированные конструкции

Вы можете размещать циклы или конструкции `switch` внутри других циклов или `switch` конструкций, создавая тем самым сложное течение исполнения кода. Однако циклы и конструкции `switch` могут иметь `break`, что может прервать выполнение кода преждевременно. В общем иногда полезно явно указывать какой цикл или какую конструкцию `switch` вы хотите прервать оператором `break`. Так же, если у вас есть несколько вложенных циклов, то может быть полезным явное указание того, на какой цикл именно будет действовать оператор `continue`.

Для всех этих целей мы можем маркировать цикл или конструкцию `switch` маркером конструкций и использовать его вместе с оператором `break` или оператором `continue` для предотвращения или продолжения исполнения маркированной конструкции.

Маркированные конструкции обозначаются маркером, идущим на той же строке как и ключевое слово конструкции, сопровождаемым двоеточием. Ниже приведен пример синтаксиса цикла `while`, хотя принцип работы маркера такой же со всеми конструкциями:

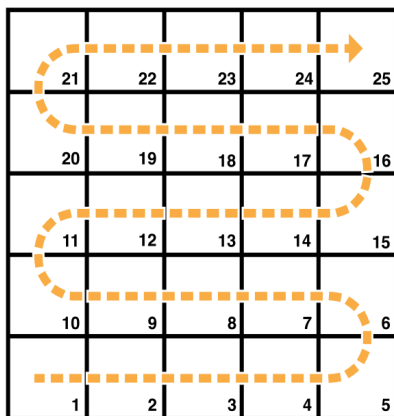
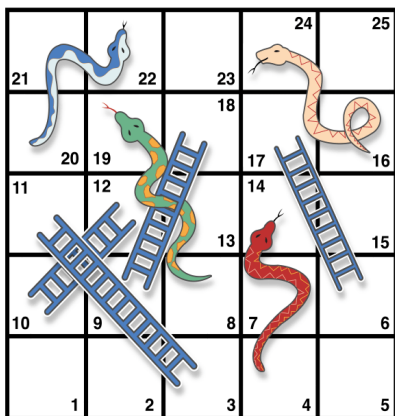
```
имя маркера : while условие {  
    исполняемый код  
}
```

В дальнейшем примере мы будем использовать `break`, `continue` с маркированным циклом `while` для адаптированной версии *Змеи и Лестницы*, которую вы видели ранее в "Циклы While". В этот раз у нас появилось новое правило:

- Чтобы победить вы должны попасть точно на клетку 25.

Если результат броска кубика дает вам ходов более чем на 25 клетку, то вы должны бросить еще раз, до тех пор пока не попадете точно на клетку 25.

Игровая зона доски осталась такой же как и была:



Величины `finalSquare`, `board`, `square` и `diceRoll` инициализируются точно так же как и в прошлых примерах игры:

```
let finalSquare = 25
var board = [Int](count: finalSquare + 1,
  repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] =
+09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] =
-02; board[24] = -08
var square = 0
var diceRoll = 0
```

В этой версии игры используется цикл `while` и конструкция `switch` для воплощения логики игры. Цикл `while` маркер названный `gameLoop`, для индикации главного цикла игры.

Условие цикла `while square != finalSquare` показывает, что теперь нам нужно попасть строго на клетку 25:

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        //после броска кубика мы попадаем на
        клетку 25, игра окончена
        break gameLoop
    case let newSquare where newSquare >
finalSquare:
        //мы кинули кубик на слишком большое
        значение, значит нам нужно кинуть снова
        continue gameLoop
    default:
        //допустимое движение по игровому
        полю, двигаемся
        square += diceRoll
        square += board[square]
    }
}
print("Game over!")
```

Игральная кость бросается в начале каждого цикла. Прежде чем двигаться по доске идет проверка в конструкции `switch` на валидность хода, потом обрабатывает его, если такое движение допустимо:

- Если игральная кость двигает игрока на последнюю клетку, игра заканчивается. Оператор `break` с маркером `gameLoop` перекидывает исполнение кода на первую строку кода после цикла `while`, которая и завершает игру.
- Если игральная кость двигает игрока далее чем на последнюю клетку, то такое движение считается

некорректным, и игроку приходится кидать кость еще раз. Оператор `continue` с маркером `gameLoop` заканчивает итерацию и начинает новую.

- Во всех случаях движение игрока на `diceRoll` клеток допустимо и каждый ход идет проверка логики игры на наличие лестниц и змей. Когда кончается итерация, мы возвращаемся на начало цикла `while`, где проверяется условие на необходимость дальнейших ходов.

Заметка

Если оператор `break` не использует маркер `gameLoop`, то он будет прерывать выполнение конструкции `switch`, а не всего цикла `while`. Но используя маркер `gameLoop` мы можем указать какое исполнение конструкции нужно прервать. Обратите внимание так же и на то, что нет необходимости использовать маркер `gameLoop`, когда мы обращаемся к `continue gameLoop` для того, чтобы перейти к следующей итерации. В этой игре всего один цикл, так что нет никакой двусмысленности на какой цикл этот оператор может вообще воздействовать. Однако и никакого вреда нет в том, что мы явно указали `gameLoop` маркер в строке с оператором `continue`. Более того, делая так мы делаем наш код нагляднее и восприятие логики игры становится более ясным.

Функции

Функции – это самостоятельные фрагменты кода, решающие определенную задачу. Каждой функции присваивается уникальное имя, по которому ее можно идентифицировать и "вызвать" в нужный момент.

Язык Swift предлагает достаточно гибкий единый синтаксис функций – от простых C-подобных функций без параметров до сложных методов в стиле Objective-C с локальными и внешними параметрами. Параметры могут служить как для простой инициализации значений внутри функции, так и для изменения внешних переменных после выполнения функции.

Каждая функция в Swift имеет тип, описывающий тип параметров функции и тип возвращаемого значения. Тип функции можно использовать аналогично любым другим типам в Swift, т. е. одна функция может быть параметром другой функции либо ее результирующим значением. Функции также могут вкладываться друг в друга, что позволяет инкапсулировать определенный алгоритм внутри локального контекста.

Объявление и вызов функции

При объявлении функции можно задать одно или несколько именованных типизированных значений, которые будут ее входными данными (или *параметрами*), а также тип значения, которое функция будет передавать в качестве результата (или *возвращаемый тип*).

У каждой функции должно быть *имя*, которое отражает решаемую задачу. Чтобы воспользоваться функцией, ее нужно "вызвать", указав имя и входные значения (*аргументы*), соответствующие типам параметров этой функции. Аргументы функции всегда должны идти в том же порядке, в каком они были указаны при объявлении функции.

В приведенном ниже примере функция называется `sayHello`, потому что это отражает ее задачу – получить имя пользователя и вежливо поздороваться. Для этого задается один входной параметр типа `String` под названием `personName`, а возвращается тоже значение типа `String`, но уже содержащее приветствие:

```
1. func sayHello(personName: String) -> String {  
2. let greeting = "Hello, " + personName + "!"  
3. return greeting  
4. }
```

Вся эта информация указана в *объявлении функции*, перед которым стоит ключевое слово `func`. Тип возвращаемого значения функции ставится после *результатирующей стрелки* `->` (это дефис и правая угловая скобка).

Из объявления функции можно узнать, что она делает, какие у нее входные данные и какой результат она возвращает. Объявленную функцию можно однозначно вызывать из любого участка кода:

```
1. println(sayHello("Anna"))
2. // напечатает "Hello, Anna!"
3. println(sayHello("Brian"))
4. // напечатает "Hello, Brian!"
```

Так, функцию `sayHello` можно вызвать с аргументом типа `String`, взятым в кавычки: `sayHello("Anna")`. Поскольку функция возвращает значение типа `String`, вызов функции `sayHello` можно указать в качестве параметра функции `println`, чтобы напечатать полученную строку (см. выше).

Тело функции `sayHello` начинается с объявления новой константы типа `String` под названием `greeting`, в которую записывается простое приветствие для пользователя `personName`. Затем это приветствие возвращается в точку вызова функции с помощью ключевого слова `return`. После выполнения оператора `return greeting` функция завершает свою работу и возвращает текущее значение `greeting`.

Функцию `sayHello` можно вызывать многократно и с разными входными значениями. В примере выше показано, что будет, если функцию вызвать с аргументом `"Anna"`, а затем с аргументом `"Brian"`. В каждом случае функция возвратит персональное приветствие.

Чтобы упростить код этой функции, можно записать создание сообщения и его возврат в одну строку:

```
1. func sayHelloAgain(personName: String) -> String {
2.   return "Hello again, " + personName + "!"
3. }
4. println(sayHelloAgain("Anna"))
5. // напечатает "Hello again, Anna!"
```

Параметры функции и возвращаемые значения

В языке Swift параметры функций и возвращаемые значения реализованы очень гибко. Разработчик может объявлять любые функции – от простейших, с одним безымянным параметром, до сложных, со множеством параметров и составными именами.

Функции с несколькими входными параметрами

У функции может быть несколько параметров, которые указываются через запятую в скобках.

Следующая функция получает номера начального и конечного значений полуинтервала и подсчитывает, сколько в нем элементов:

```
func halfOpenRangeLength(start: Int, end: Int)
-> Int {
    return end - start
}
println(halfOpenRangeLength(1, 10))
// напечатает "9"
```

Функции без параметров

В некоторых случаях функции могут не иметь входных параметров. Вот пример функции без входных параметров, которая при вызове всегда возвращает одно и то же значение типа `String`:

```
func sayHelloWorld() -> String {  
    return "hello, world"  
}  
println(sayHelloWorld())  
// напечатает "hello, world"
```

Обратите внимание, что несмотря на отсутствие параметров, в объявлении функции все равно нужно ставить скобки после имени. При вызове после имени функции также указываются пустые скобки.

Функции, не возвращающие значения

В некоторых случаях функции могут не иметь возвращаемого типа. Вот другая реализация функции `sayHello` под названием `sayGoodbye`, которая печатает собственное значение типа `String`, но не возвращает его:

```
func sayGoodbye(personName: String) {  
    println("Goodbye, \(personName)!")  
}  
sayGoodbye("Dave")  
// напечатает "Goodbye, Dave!"
```

Так как у функции нет выходного значения, в ее объявлении отсутствует результирующая стрелка (`->`) и возвращаемый тип.

Заметка

Строго говоря, функция `sayGoodbye` все же возвращает значение, хотя оно нигде и не указано. Функции, для которых не задан возвращаемый тип, получают специальный тип `Void`. По сути, это просто пустой кортеж, т. е. кортеж с нулем элементов, который записывается как `()`.

Выходное значение функции может быть игнорировано:

```
func printAndCount(stringToPrint: String) ->
Int {
    println(stringToPrint)
    return count(stringToPrint)
}
func printWithoutCounting(stringToPrint:
String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// напечатает "hello, world" и вернет значение
12
printWithoutCounting("hello, world")
// напечатает "hello, world", но не вернет
значение
```

Первая функция, `printAndCount`, печатает строку, а затем возвращает ее длину в виде целого (`Int`). Вторая функция, `printWithoutCounting`, вызывает первую, но не обрабатывает ее результат. При вызове второй функции первая функция по-прежнему печатает сообщение, но ее результат не используется.

Заметка

Хотя возвращаемые значения можно игнорировать, функция все же должна вернуть то, что задано в ее объявлении. Функция, для которой указан возвращаемый тип, не может

заканчиваться оператором, который ничего не возвращает, иначе произойдет ошибка во время компиляции.

Функции, возвращающие несколько значений

Чтобы возвращать из функции несколько значений в виде составного параметра, нужно объявить функцию типа кортеж.

В следующем примере объявлена функция `minMax`, которая ищет минимальный и максимальный элементы в массиве типа `Int`:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  
  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..  
        array.count] {  
        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```

Функция `minMax` возвращает кортеж из двух значений типа `Int`. Этим значениям присвоены имена `min` и `max`, чтобы к ним можно было обращаться при запросе возвращаемого типа функции.

Тело функции `minMax` начинается с инициализации двух рабочих переменных `currentMin` и `currentMax` значением первого целого элемента в массиве. Затем функция последовательно проходит по всем остальным значениям в массиве и сравнивает их со значениями `currentMin` и

`currentMax` соответственно. И наконец, самое маленькое и самое большое значения возвращаются внутри кортежа типа `Int`.

Так как имена элементов кортежа указаны в возвращаемом типе функции, к ним можно обращаться через точку и считывать значения:

```
let bounds = minMax([8, -6, 2, 109, 3, 71])
println("min is \"(bounds.min)\" and max is \"(bounds.max)\"")
// напечатает "min is -6 and max is 109"
```

Обратите внимание, что к элементам кортежа не нужно обращаться через точку при вызове функции, так как их имена уже указаны в ее возвращаемом типе.

Кортеж-опционал как возвращаемый тип

Если возвращаемый из функции кортеж может иметь "пустое значение", то его следует объявить как *кортеж-опционал*, т. е. кортеж, который может равняться `nil`. Чтобы сделать возвращаемый кортеж опционалом, нужно поставить вопросительный знак после закрывающей скобки: `(Int, Int)?` или `(String, Int, Bool)?`.

Заметка

Кортеж-опционал вида `(Int, Int)?` это не то же самое, что кортеж, содержащий опционалы: `(Int?, Int?)`. Кортеж-опционал сам является опционалом, но не обязан состоять из опциональных значений.

Функция `minMax` выше возвращает кортеж из двух значений типа `Int`, однако не проверяет корректность передаваемого

массива. Если аргумент `array` содержит пустой массив, для которого `count` равно 0, функция `minMax` в том виде, в каком она приведена выше, выдаст ошибку времени выполнения, когда попытается обратиться к элементу `array[0]`.

Для устранения этого недочета перепишем функцию `minMax` так, чтобы она возвращала кортеж-опционал, который в случае пустого массива примет значение `nil`:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

Чтобы проверить, возвращает ли эта версия функции `minMax` фактическое значение кортежа или `nil`, можно использовать привязку опционала:

```
1. if let bounds = minMax([8, -6, 2, 109, 3, 71]) {
2.     println("min is \(bounds.min) and max
   is \(bounds.max)")
3. }
4. // напечатает "min is -6 and max is 109"
```

Имена параметров

функции

Во всех рассмотренных выше функциях определены *имена параметров*:

```
func someFunction(parameterName: Int) {  
    // тело функции, в котором можно  
    использовать parameterName  
    // для обращения к значению этого  
    аргумента  
}
```

Тем не менее эти имена параметров используются только в теле функции и недоступны при ее вызове. Такие имена называются *локальными именами параметров*, поскольку они доступны только в пределах тела функции.

Внешние имена параметров

Иногда удобно присваивать параметрам имена *во время вызова* функции, чтобы указать их назначение.

Чтобы при вызове функции можно было присвоить имена ее параметрам, объявите для них *внешние имена* в дополнение к локальным. Внешнее имя параметра записывается перед соответствующим локальным через пробел:

```
func someFunction(externalParameterName  
localParameterName: Int) {
```

```
    // тело функции, в котором можно
использовать localParameterName
    // для обращения к значению этого
аргумента
}
```

Заметка

Если для параметра объявлено внешнее имя, то при вызове функции для него всегда нужно указывать внешнее имя.

В качестве примера рассмотрим следующую функцию, которая объединяет две строки, вставляя между ними "строку-соединитель":

```
func join(s1: String, s2: String, joiner:
String) -> String {
    return s1 + joiner + s2
}
```

В момент вызова функции назначение этих трех входных строк еще не вполне понятно:

```
join("hello", "world", ", ")
// возвратит "hello, world"
```

Чтобы прояснить эти значения типа `String`, объявим внешние имена для каждого параметра функции `join`:

```
func join(string s1: String, toString s2:
String, withJoiner joiner: String)
-> String {
    return s1 + joiner + s2
}
```

В этой версии функции `join` у первого параметра есть внешнее имя `string` и локальное имя `s1`; у второго параметра есть внешнее имя `toString` и локальное имя `s2`; и наконец, у третьего параметра есть внешнее имя `withJoiner` и локальное имя `joiner`.

Теперь с помощью этих внешних имен параметров можно однозначно вызывать функцию:

```
join(string: "hello", toString: "world",  
withJoiner: ", ")  
// возвратит "hello, world"
```

Благодаря использованию внешних имен параметров вторую версию функции `join` можно вызывать внутри выражений без ущерба для читаемости кода и понимания ее алгоритма.

Заметка

Внешние имена параметров лучше всего использовать, когда аргументы функции не очевидны для тех, кто читает код впервые. Если же назначение каждого входного параметра интуитивно ясно при вызове функции, то внешние имена не нужны.

Сокращенные внешние имена параметров

Если необходимо задать внешнее имя для параметра функции, у которого уже есть подходящее локальное имя, то дублировать это имя не требуется. Вместо этого можно написать имя один раз и поставить перед ним решетку (`#`). На языке Swift это будет означать, что имя параметра является одновременно и локальным, и внешним.

В следующем примере объявлена функция `containsCharacter`, в которой внешние имена заданы для

```
func containsCharacter(#string: String,
#characterToFind: Character) -> Bool {
    for character in string {
        if character == characterToFind {
            return true
        }
    }
    return false
}
```

Такой выбор имен не только делает код функции проще и понятнее, но и устраняет двусмысленность при ее вызове:

```
let containsAVee = containsCharacter(string:
"aardvark", characterToFind: "v")
// containsAVee равно true, так как "aardvark"
содержит "v"
```

Значения по умолчанию для параметров

При объявлении функции любому из ее параметров можно присвоить *значение по умолчанию*. Если у параметра есть значение по умолчанию, то при вызове функции этот параметр можно опустить.

Заметка

Параметры со значениями по умолчанию лучше всего указывать в конце списка параметров функции. Это гарантирует, что при каждом вызове функции порядок аргументов будет одинаковым, а также подсказывает тем, кто читает код, что вызывается одна и та же функция.

Вот еще одна реализация рассмотренной выше функции `join`, в которой параметру `joiner` задано значение по умолчанию:

```
func join(string s1: String, toString s2: String,

        withJoiner joiner: String = " ") -> String
{
    return s1 + joiner + s2
}
```

Если при вызове функции `join` параметру `joiner` присваивается значение, то оно, как и раньше, будет использовано в качестве строки-соединителя:

```
join(string: "hello", toString: "world",
withJoiner: "-")
// возвратит "hello-world"
```

Однако если при вызове функции параметр `joiner` пропущен, будет использоваться символ пробела (" ") как значение по умолчанию:

```
join(string: "hello", toString: "world")
// возвратит "hello world"
```

Внешние имена для параметров со значениями по умолчанию

В большинстве случаев для каждого параметра со значением по умолчанию лучше указывать внешнее имя. Это гарантирует, что разработчики поймут назначение такого

параметра, если при вызове функции ему присвоено значение.

Для удобства язык Swift автоматически присваивает внешние имена всем параметрам, имеющим значения по умолчанию. Автоматически созданное внешнее имя совпадает с локальным, как если бы перед локальным именем стоял символ решетки.

Вот еще одна версия рассмотренной выше функции `join`, в которой нет внешних имен для параметров, но по-прежнему есть значение по умолчанию для параметра `joiner`:

```
func join(s1: String, s2: String, joiner:
String = " ") -> String {
    return s1 + joiner + s2
}
```

В этом случае Swift автоматически присваивает параметру `joiner` внешнее имя. Соответственно, при вызове функции нужно указывать это внешнее имя, чтобы смысл параметра был понятным и однозначным:

```
join("hello", "world", joiner: "-")
// возвратит "hello-world"
```

Заметка

Это поведение можно отключить, поставив знак подчеркивания (`_`) вместо явного указания внешнего имени при объявлении параметра. Однако лучше все-таки использовать внешние имена.

Вариативные параметры

Вариативным называется параметр, который может иметь сразу несколько значений или не иметь ни одного. С помощью вариативного параметра можно передать в функцию произвольное число входных значений. Чтобы объявить параметр как вариативный, нужно поставить три точки (...) после его типа.

Значения, переданные через вариативный параметр, доступны внутри функции в виде массива соответствующего типа. Например, вариативный параметр `numbers` типа `Double...` доступен внутри функции в виде массива-константы `numbers` типа `[Double]`.

В приведенном ниже примере вычисляется *среднее арифметическое* (или же *среднее*) последовательности чисел, имеющей произвольную длину:

```
func arithmeticMean(numbers: Double...) ->
Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// возвратит 3.0, т. е. среднее арифметическое
этих пяти чисел
arithmeticMean(3, 8.25, 18.75)
// возвратит 10.0, т. е. среднее
арифметическое этих трех чисел
```

Заметка

У функции может быть только один вариативный параметр, который всегда должен стоять последним в списке параметров, чтобы при вызове функции, имеющей несколько параметров, не возникало двусмысленности.

Если у функции, помимо вариативного параметра, есть еще один или несколько параметров со значениями по умолчанию, то вариативный параметр нужно ставить после них в самом конце списка.

Параметры-константы и параметры-переменные

По умолчанию параметры функции являются константами, поэтому попытка изменить значение параметра функции из тела этой же функции приведет к ошибке времени компиляции. Это защищает параметры от случайного изменения.

Однако иногда внутри функции удобнее работать с *переменной* копией параметра. Чтобы не создавать вручную новую переменную в теле функции, можно объявить один или несколько параметров *переменными*. Переменные параметры намного ближе к обычным переменным, чем к константам, и позволяют работать с изменяемыми копиями соответствующих параметров функции.

Чтобы объявить параметр как переменный, нужно перед его именем поставить ключевое слово `var`:

```
func alignRight(var string: String,
totalLength: Int, pad: Character) -> String {
    let amountToPad = totalLength -
count(string)
    if amountToPad < 1 {
        return string
    }
    let padString = String(pad)
    for _ in 1...amountToPad {
        string = padString + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString,
10, "-")
// paddedString равно "-----hello"
// originalString по-прежнему равно "hello"
```

В этом примере объявлена новая функция `alignRight`, которая выравнивает входную строку по правому краю более длинной выходной строки. Все свободное пространство слева заполняется специальным символом. В данном случае строка `"hello"` преобразуется в строку `"-----hello"`.

В функции `alignRight` входной параметр `string` объявлен как переменный параметр. Это означает, что `string` теперь является локальной переменной, инициализированной значением входной строки, поэтому с ней можно работать прямо в теле функции.

Сначала функция определяет, сколько символов нужно добавить слева от `string`, чтобы выровнять ее по правому

краю относительно общей строки. Результат записывается в локальную константу `amountToPad`. Если отбивка не требуется (т. е. `amountToPad` меньше 1), функция просто возвращает входное значение `string` без заполнителей.

В противном случае функция создает временную константу типа `String` под названием `padString`, в которую записывается символ `pad`, и добавляет `amountToPad` копий строки `padString` слева от существующей строки. (Значение типа `String` нельзя добавить к значению типа `Character`, поэтому была введена константа `padString`, которая гарантирует, что оба слагаемых оператора `+` имеют тип `String`.)

Заметка

Изменения переменных параметров не сохраняются после завершения работы функции и не видны за ее пределами. Переменные параметры существуют только во время работы функции.

Сквозные параметры

Переменные параметры, как описано выше, могут изменяться только в теле функции. Чтобы после завершения работы функции ее измененные параметры сохранялись, нужно объявить их не как переменные, а как *сквозные параметры*.

Для создания сквозного параметра нужно поставить ключевое слово `inout` перед объявлением параметра. Сквозной параметр передает значение *в функцию*, которое затем изменяется в ней и возвращается *из функции*, заменяя исходное значение.

Аргументом для сквозного параметра может быть только переменная. Константы или литералы нельзя передать в сквозной параметр, так как это неизменяемые элементы. Непосредственно перед именем переменной ставится амперсанд (&), обозначающий, что аргумент, переданный как входной параметр, можно изменять в теле функции.

Заметка

Сквозные параметры не могут иметь значения по умолчанию и не могут быть вариативными параметрами с ключевым словом `inout`. Если параметр объявлен как `inout`, для него уже нельзя использовать `var` или `let`.

Вот пример функции под названием `swapTwoInts`, у которой есть два сквозных целочисленных параметра – `a` и `b`:

```
func swapTwoInts(inout a: Int, inout b: Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Функция `swapTwoInts` просто присваивает переменной `b` значение `a`, а переменной `a` – значение `b`. Для этого функция сохраняет значение `a` во временной переменной `temporaryA`, присваивает значение `b` переменной `a`, а затем присваивает значение `temporaryA` переменной `b`.

С помощью функции `swapTwoInts` можно поменять местами значения двух переменных типа `Int`. Обратите внимание, что перед именами переменных `someInt` и `anotherInt` при их передаче в функцию `swapTwoInts` стоит амперсанд:

```
var someInt = 3  
var anotherInt = 107
```

```
swapTwoInts(&someInt, &anotherInt)
println("someInt is now \(someInt), and
anotherInt is now \(anotherInt)")
// напечатает "someInt is now 107, and
anotherInt is now 3"
```

В вышеприведенном примере видно, что исходные значения переменных `someInt` и `anotherInt` изменены функцией `swapTwoInts`, несмотря на то, что изначально они были объявлены за ее пределами.

Заметка

Сквозные параметры – это не то же самое, что возвращаемые функцией значения. В примере с функцией `swapTwoInts` нет ни возвращаемого типа, ни возвращаемого значения, но параметры `someInt` и `anotherInt` все равно изменяются. Сквозные параметры – это альтернативный способ передачи изменений, сделанных внутри функции, за пределы тела этой функции.

Функциональные типы

У каждой функции есть специальный *функциональный тип*, состоящий из типов параметров и типа возвращаемого значения.

Пример:

```
1. func addTwoInts(a: Int, b: Int) -> Int {
2.   return a + b
3. }
4. func multiplyTwoInts(a: Int, b: Int) -> Int {
5.   return a * b
6. }
```


В данном примере объявлены две простые математические функции – `addTwoInts` и `multiplyTwoInts`. Каждая из этих функций принимает два значения типа `Int` и возвращает одно значение типа `Int`, содержащее результат математической операции.

Обе функции имеют тип `(Int, Int) -> Int`. Эта запись означает следующее:

"функция с двумя параметрами типа `Int`, возвращающая значение типа `Int`". Вот еще один пример функции без параметров и возвращаемого значения:

```
1. func printHelloWorld() {  
2.     println("hello, world")  
3. }
```

Эта функция имеет тип `() -> ()`, т. е. "функция без параметров, которая возвращает `Void`". Функции, в которых не указано возвращаемое значение, всегда возвращают значение типа `Void`, эквивалентное в Swift пустому кортежу, который обозначается как `()`.

Использование функциональных типов

В Swift с функциональными типами можно работать так же, как и с другими типами. Например, можно объявить константу или переменную функционального типа и присвоить ей функцию соответствующего типа:

```
1. var mathFunction: (Int, Int) -> Int = addTwoInts
```

Эта запись означает следующее:

"Объявить переменную `mathFunction`, имеющую тип "функция, принимающая два значения типа `Int`, и возвращающая одно значение типа `Int`". Присвоить этой новой переменной указатель на функцию `addTwoInts`".

Функция `addTwoInts` имеет тот же тип, что и переменная `mathFunction`, поэтому с точки зрения языка Swift такое присваивание корректно.

Теперь функцию можно вызывать с помощью переменной `mathFunction`:

```
1. println("Result: \(mathFunction(2, 3))")
2. // напечатает "Result: 5"
```

Той же переменной можно присвоить и другую функцию такого же типа – аналогично нефункциональным типам:

```
1. mathFunction = multiplyTwoInts
2. println("Result: \(mathFunction(2, 3))")
3. // напечатает "Result: 6"
```

Как и для любого другого типа в Swift, тип функции, присвоенной константе или переменной, проверяется автоматически:

```
1. let anotherMathFunction = addTwoInts
2. // для переменной anotherMathFunction выведен тип (Int, Int) ->
   Int
```

Функциональные типы как типы параметров

Функциональные типы наподобие `(Int, Int) -> Int` могут быть типами параметров другой функции. Это позволяет определять некоторые аспекты реализации функции непосредственно во время ее вызова.

Следующий код печатает на экране результаты работы приведенных выше математических функций:

```
1. func printMathResult(mathFunction: (Int, Int) -  
  > Int, a: Int, b: Int) {  
2. println("Result: \(mathFunction(a, b))")  
3. }  
4. printMathResult(addTwoInts, 3, 5)  
5. // напечатает "Result: 8"
```

В этом примере объявлена функция `printMathResult`, у которой есть три параметра. Первый параметр под названием `mathFunction` имеет тип `(Int, Int) -> Int`. Соответственно, аргументом этого параметра может быть любая функция такого же типа. Второй и третий параметры называются `a` и `b` и относятся к типу `Int`. Они служат для передачи двух входных значений для математической функции.

При вызове `printMathResult` получает в качестве входных данных функцию `addTwoInts` и два целочисленных значения `3` и `5`. Затем она вызывает переданную функцию со значениями `3` и `5`, а также выводит на экран результат `8`.

Задача функции `printMathResult` заключается в том, чтобы печатать результат работы математической функции соответствующего типа. При этом конкретные детали этой математической функции не имеют значения – главное,

чтобы она была подходящего типа. Все это позволяет безопасно управлять работой функции `printMathResult` непосредственно во время вызова.

Функциональные типы как возвращаемые типы

Функциональный тип можно сделать возвращаемым типом другой функции. Для этого нужно записать полный функциональный тип сразу же после результирующей стрелки (`->`) в возвращаемой функции.

В следующем примере объявлены две простые функции – `stepForward` и `stepBackward`. Функция `stepForward` возвращает входное значение, увеличенное на единицу, а функция `stepBackward` – уменьшенное на единицу. Обе функции имеют тип `(Int) -> Int`:

```
1. func stepForward(input: Int) -> Int {
2.   return input + 1
3. }
4. func stepBackward(input: Int) -> Int {
5.   return input - 1
6. }
```

Следующая функция под названием `chooseStepFunction`, имеет тип "функция типа `(Int) -> Int`". Функция `chooseStepFunction` возвращает функцию `stepForward` или функцию `stepBackward` в зависимости от значения логического параметра `backwards`:

```
1. func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
2.   return backwards ? stepBackward : stepForward
3. }
```

Теперь с помощью `chooseStepFunction` можно получать функцию, которая будет сдвигать значение влево или вправо:

```
1. var currentValue = 3
2. let moveNearerToZero = chooseStepFunction(currentValue > 0)
3. // moveNearerToZero теперь указывает на функцию stepBackward()
```

В предыдущем примере мы определяли, нужно ли прибавить или отнять единицу, чтобы последовательно приблизить переменную `currentValue` к нулю. Изначально `currentValue` имеет значение `3`, т. е. сравнение `currentValue > 0` даст `true`, а функция `chooseStepFunction`, соответственно, возвратит функцию `stepBackward`. Указатель на возвращаемую функцию хранится в константе `moveNearerToZero`.

Так как `moveNearerToZero` теперь ссылается на нужную функцию, можно использовать эту константу для отсчета до нуля:

```
1. println("Counting to zero:")
2. // Отсчет до нуля:
3. while currentValue != 0 {
4.   println("\(currentValue) ... ")
5.   currentValue = moveNearerToZero(currentValue)
6. }
7. println("zero!")
8. // 3...
9. // 2...
10.    // 1...
11.    // zero!
```

Вложенные функции

Все ранее рассмотренные в этом разделе функции являются *глобальными*, т. е. определенными в глобальном контексте. Но помимо глобальных можно объявлять и функции, находящиеся внутри других функций, или же *вложенные*.

Вложенные функции по умолчанию недоступны извне, а вызываются и используются только родительской функцией. Родительская функция может также возвращать одну из вложенных, чтобы вложенную функцию можно было использовать за ее пределами.

Приведенный выше пример с функцией `chooseStepFunction` можно переписать со вложенными функциями:

```
1. func chooseStepFunction(backwards: Bool) -> (Int) -  
  > Int {  
2.   func stepForward(input: Int) -  
     > Int { return input + 1 }  
3.   func stepBackward(input: Int) -  
     > Int { return input - 1 }  
4.   return backwards ? stepBackward : stepForward  
5. }  
6. var currentValue = -4  
7. let moveNearerToZero = chooseStepFunction(currentVa  
   lue > 0)  
8. // moveNearerToZero теперь указывает на вложенную  
   функцию stepForward()  
9. while currentValue != 0 {  
10.   println("\ (currentValue) ... ")  
11.   currentValue = moveNearerToZero(currentValue)  
12. }  
13. println("zero!")  
14. // -4...
```

```
15.      // -3...
16.      // -2...
17.      // -1...
18.      // zero!
```

Замыкания

Замыкания - это самоорганизованные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде. Замыкания в Swift похожи на блоки в C и Objective-C, и лямбды в других языках программирования.

Замыкания могут захватывать и хранить отношения для любых констант и переменных из контекста, в котором они объявлены. Эта процедура известна как *заклучение* этих констант и переменных, отсюда и название "*замыкание*". Swift выполняет всю работу с управлением памятью при захвате за вас.

Заметка

Не волнуйтесь, если вы не знакомы с понятием "*захвата*" (*capturing*). Это объясняется более подробно ниже в главе *Захват значений*.

Глобальные и вложенные функции, которые были представлены в главе *Функции*, являются частным случаем замыканий. Замыкания принимают одно из трех форм:

- Глобальные функции являются замыканиями, у которых есть имя и которые не захватывают никакие значения.
- Вложенные функции являются замыканиями, у которых есть имя и которые могут захватывать значения из замыкающей функции.
- Выражения замыкания являются безымянными замыканиями, написанные в облегченном синтаксисе, которые могут захватывать значения из их окружающего контекста.

Выражения замыкания в Swift имеют четкий, ясный, оптимизированный синтаксис в распространенных сценариях. Эти оптимизации включают:

- Вывод типа параметра и возврат типа значения из контекста
- Неявные возвращающиеся значения однострочных замыканий
- Сокращенные имена параметров
- Синтаксис последующих замыканий

Замыкающие выражения

Вложенные функции, которые были представлены в главе [Вложенные функции](#), являются удобным способом для обозначения и объявления самоорганизованных блоков кода, которые являются частью более крупной функции. Тем не менее, иногда полезно писать короткие версии функциональных конструкций, без полного объявления и указания имени. Это особенно верно, когда вы работаете с функциями, которые принимают другие функции в виде одного из своих параметров.

Замыкающие выражения, являются способом написания встроенных замыканий через краткий и специализированный синтаксис. Замыкающие выражения обеспечивают несколько синтаксических оптимизаций для написания замыканий в краткой форме, без потери ясности и умысла. Примеры замыкающих выражений ниже, показывают эти оптимизации путем рассмотрения функции `sorted` при нескольких итерациях, каждая из которых изображает ту же функциональность в более сжатой форме.

Функция `sorted`

В стандартной библиотеке Swift есть функция называемая `sorted`, которая сортирует массив значений определенных типов, основываясь на результате сортирующего замыкания, которого вы ему передадите. После завершения процесса сортировки, функция `sorted` возвращает новый массив того же типа и размера как старый, с элементами в правильном порядке сортировки. Исходный массив не изменяется функцией `sorted`.

Примеры замыкающих выражений ниже используют функцию `sorted` для сортировки массива из `String` значений в обратном алфавитном порядке. Вот исходный массив для сортировки:

```
let names = ["Chris", "Alex", "Ewa",  
"Barry", "Daniella"]
```

Функция `sorted` принимает два параметра:

- Массив из значений определенного типа
- Замыкание, которое принимает два параметра того же типа, что и содержимое массива, и возвращает `Bool` значение, которое решает поставить ли первое значение перед вторым, или после второго. Замыкание сортировки должно вернуть `true`, если первое значение должно быть до второго значения, и `false` в противном случае.

Этот пример сортирует массив из `String` значений, так что сортирующее замыкание должно быть функцией с типом `(String, String) -> Bool`.

Один из способов обеспечить сортирующее замыкание, это написать нормальную функцию нужного типа, и передать его в качестве второго параметра функции `sorted`:

```
1. func backwards(s1: String, s2: String) -> Bool {  
2.   return s1 > s2  
3. }  
4. var reversed = sorted(names, backwards)  
5. // reversed равен ["Ewa", "Daniella", "Chris",  
   "Barry", "Alex"]
```

Если первая строка (`s1`) больше чем вторая строка (`s2`), функция `backwards` возвращает `true`, что указывает, что `s1` должна быть перед `s2` в сортированном массиве. Для символов в строках, "больше чем" означает "появляется в алфавите позже, чем". Это означает что буква "В" "больше чем" буква "А", а строка "Tom" больше чем строка "Tim". Это делает обратную алфавитную сортировку, с "Barry" поставленным перед "Alex", и так далее.

Тем не менее, это довольно скучный способ написать то, что по сути, является функцией с одним выражением (`a > b`). В этом примере, было бы предпочтительнее написать сортирующее замыкание в одну строку, используя синтаксис замыкающего выражения.

Синтаксис замыкающего выражения

Синтаксис замыкающего выражения имеет следующую общую форму:

```
1. { ( параметры ) -> тип результата in
2. выражения
3. }
```

Синтаксис замыкающего выражения может использовать параметры константы, параметры переменные, и сквозные параметры. Значения по умолчанию не могут быть переданы. Вариативные параметры могут быть использованы если вы назовете вариативные параметр и поместите его последним в списке параметров. Кортежи также могут быть использованы как типы параметров и как типы возвращаемого значения.

Пример ниже показывает версию функции `backwards` с использованием замыкающего выражения:

```
1. reversed = sorted(names, { (s1: String, s2: String) -
> Bool in
2. return s1 > s2
3. })
```

Обратите внимание, что объявление типов параметров и типа возвращаемого значения для этого однострочного замыкания идентично объявлению из функции `backwards`. В обоих случаях, оно пишется в виде `(s1: String, s2: String) -> Bool`. Тем не менее, для однострочных замыкающих выражений, параметры и тип возвращаемого значения пишутся внутри фигурных скобок, а не вне их.

Начало тела замыкания содержит ключевое слово `in`. Это ключевое слово указывает, что объявление параметров и возвращаемого значения замыкания закончено, и тело замыкания вот-вот начнется.

Поскольку тело замыкания настолько короткое, оно может быть записано в одну строку:

```
reversed = sorted(names, { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

Это показывает, что общий вызов функции `sorted` остался прежним. Пара скобок по-прежнему обособляют весь набор параметров для функции. Тем не менее, один из этих параметров является теперь однострочным замыканием.

Вывод типа из контекста

Поскольку сортирующее замыкание передается как аргумент функции, Swift может вывести типы его параметров и тип возвращаемого значения, через тип второго параметра функции `sorted`. Этот параметр ожидает функцию имеющую тип `(String, String) -> Bool`. Это означает что типы `(String, String)` и `Bool` не нужно писать в объявлении замыкающего выражения. Поскольку все типы могут быть выведены, стрелка результата `(->)` и скобки вокруг имен параметров также могут быть опущены:

```
reversed = sorted(names, { s1, s2 in return s1 > s2 } )
```

Всегда можно вывести типы параметров и тип возвращаемого значения, когда мы передаем замыкание функции в виде однострочного замыкающего выражения. В результате, когда

замыкание используется как параметр функции, вам никогда не нужно писать однострочное замыкание в его полном виде.

Тем не менее, вы всё равно можете явно указать типы, если хотите. И делать это предполагается, если это поможет избежать двусмысленности для читателей вашего кода. В случае с функцией `sorted`, цель замыкания понятна из того факта, что сортировка происходит, и она безопасна для читателя, который может предположить, что замыкание, вероятно, будет работать со значениями `String`, поскольку оно помогает сортировать массив из строк.

Неявные возвращаемые значения из замыканий с одним выражением

Замыкания с одним выражением могут неявно возвращать результат своего выражения через опускание ключевого слова `return` из их объявления, как показано в этой версии предыдущего примера:

```
reversed = sorted(names, { s1, s2 in s1 > s2  
} )
```

Здесь, функциональный тип второго аргумента функции `sorted` дает понять, что замыкание вернет `Bool` значение. Поскольку тело замыкания содержит одно выражение (`s1 > s2`), которое возвращает `Bool` значение, то нет никакой двусмысленности, и ключевое слово `return` можно опустить.

Сокращенные имена параметров

Swift автоматически предоставляет сокращённые имена для однострочных замыканий, которые могут быть использованы

для обращения к значениям параметров замыкания через имена `$0`, `$1`, `$2`, и так далее.

Если вы используете эти сокращенные имена параметров с вашим замыкающим выражением, вы можете пропустить список параметров замыкания из его объявления, а количество и тип сокращенных имен параметров будет выведено из ожидаемого типа функции. Ключевое слово `in` также может быть опущено, поскольку замыкающее выражение полностью состоит из его тела:

```
reversed = sorted(names, { $0 > $1 } )
```

Здесь, `$0` и `$1` обращаются к первому и второму `String` параметру замыкания.

Операторы-функции

Здесь есть на самом деле более короткий способ написать замыкающее выражение выше. Тип `String` в Swift содержит специфичные для строк реализации оператора больше (`>`) как функции, которая имеет два `String` параметра, и возвращает `Bool` значение. Это точно соответствует типу функции, для второго параметра функции `sorted`. Таким образом, вы можете просто написать оператор больше, а Swift будет считать, что вы хотите использовать специфичную для строк реализацию:

```
reversed = sorted(names, > )
```

Более подробную информацию о операторах-функциях смотрите в разделе Операторы-функции.

Последующее замыкание

Если вам нужно передать выражение замыкания функции в качестве последнего аргумента функции и само выражение замыкания длинное, то оно может быть записано в виде *последующего замыкания*. Последующее замыкание - замыкание, которое записано в виде замыкающего выражения вне (и после) круглых скобок вызова функции, которое она содержит:

```
func someFunctionThatTakesAClosure(closure: ()
-> Void) {
    //тело функции
}
//вот пример того, как можно вызвать функцию
без последующих замыканий
someFunctionThatTakesAClosure( {
    //тело замыкания
} )
//теперь как мы можем вызвать функцию с
последующим замыканием
someFunctionThatTakesAClosure() {
    //тело последующего замыкания
}
```

Заметка

Если замыкающее выражение - единственный аргумент функции, то вы можете использовать это выражение как последующее замыкание, вам не нужно писать пару () после имени функции при ее вызове.

Сортирующее строки замыкание из главы "Синтаксис замыкающего выражения" может быть записано вне круглых скобок функции `sort(_:)`, как последующее замыкание:

```
reversed = names.sort { $0 > $1 }
```


Последующие замыкания полезны в случаях, когда само замыкание достаточно длинное, и его невозможно записать в одну строку. В качестве примера приведем вам метод `map(_:)` типа `Array` в языке Swift, который принимает выражение замыкания как его единственный аргумент. Замыкание вызывается по одному разу для каждого элемента массива и возвращает альтернативную отображаемую величину (возможно другого типа) для этого элемента. Природа отображения и тип возвращаемого значения определяется замыканием.

После применения замыкания к каждому элементу массива, метод `map(_:)` возвращает новый массив, содержащий новые преобразованные величины, в том же порядке, что и в исходном массиве.

Вот как вы можете использовать метод `map(_:)` вместе с преследующим замыканием для превращения массива значений типа `Int` в массив типа `String`. Массив `[16, 58, 510]` используется для создания нового массива `["OneSix", "FiveEight", "FiveOneZero"]`:

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3:
    "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8:
    "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

Код выше создает словарь отображающий цифры и их английская версия имен. Так же он объявляет массив целых значений для преобразования в массив строк.

Вы можете использовать массив `numbers` для создания значений типа `String`, передав замыкающее выражение в метод `map(_:)` массива в качестве последующего замыкания. Обратите внимание, что вызов `number.map` не включает в себя скобки после `map`, потому что метод `map(_:)` имеет только один параметр, который мы имеем в виде последующего замыкания:

```
let strings = numbers.map {  
    (var number) -> String in  
    var output = ""  
    while number > 0 {  
        output = digitNames[number % 10]! +  
output  
        number /= 10  
    }  
    return output  
}  
//тип строк был выведен как [String]  
//значения ["OneSix", "FiveEight",  
"FiveOneZero"]
```

Метод `map(_:)` вызывает замыкание один раз для каждого элемента массива. Вам не нужно указывать тип входного параметра замыкания, `number`, так как тип может быть выведен из значений массива, который применяет метод `map`.

В этом примере параметр замыкания `number` объявлен как параметр-переменная, которые были описаны в главе "Параметры-константы и параметры-переменные", так что значение параметра может быть изменено внутри тела

замыкания, вместо того, чтобы создавать новую переменную и присваивать ей значение `number`. Замыкающее выражение определяет возвращаемый тип как `String`, тот тип, который будет храниться в выходном массиве.

Замыкающее выражение каждый раз создает переменную `output` при его вызове. Оно считает последнюю цифру `number`, используя оператор остатка (`number % 10`) и используя эти цифры для того, чтобы посмотреть и строковый эквивалент в словаре `digitNames`. Это замыкание может быть использовано для представления любого числа, которое больше 0, в строковом эквиваленте.

Заметка

Вызов словаря `digitNames` синтаксисом сабскрипта сопровождается знаком (`!`), потому что сабскрипт словаря возвращает опциональное значение, так как есть такая вероятность, что такого ключа в словаре может и не быть. В примере выше мы точно знаем, что `number % 10` всегда вернет существующий ключ словаря `digitNames`, так что восклицательный знак используется для принудительного извлечения значения типа `String` в возвращаемом опциональном значении сабскрипта.

Строка, полученная из словаря `digitNames`, добавляется в начало переменной `output`, путем правильного формирования строковой версии числа наоборот. (Выражение `number % 10` дает нам 6 для 16, 8 для 58 и 0 для 510).

Переменная `number` после вычисления остатка делится на 10. Так как тип значения `Int`, то наше число округляется вниз, таким образом 16 превращается в 1, 58 в 5, 510 в 51.

Процесс повторяется пока `number != 10` не станет равным 0, после чего строка `output` возвращается замыканием и добавляется к выходному массиву функции `map(_:_:)`.

Использование синтаксиса последующих замыканий в примере выше аккуратно инкапсулирует функциональность замыкания сразу после функции `map(_:_:)`, которой замыкание помогает, без необходимости заворачивания всего замыкания внутрь внешних круглых скобок функции `map(_:_:)`.

Захват значений

Замыкания могут *захватывать* константы и переменные из окружающего контекста, в котором оно объявлено. После захвата замыкание может ссылаться или модифицировать значения этих констант и переменных внутри своего тела, даже если область, в которой были объявлены эти константы и переменные уже больше не существует.

В Swift самая простая форма замыкания может захватывать значения из вложенных функций, написанных внутри тела других функций. Вложенная функция может захватить любые значения из аргументов окружающей ее функции, а также константы и переменные, объявленные внутри тела внешней функции.

Вот пример функции `makeIncrementer`, которая содержит вложенную функцию `incrementer`. Вложенная функция `incrementer()` захватывает два значения `runningTotal` и `amount` и окружающего контекста. После захвата этих значений `incrementer` возвращается функцией `makeIncrementer` как замыкание, которое увеличивает `runningTotal` на `amount` каждый раз как вызывается.

```

func makeIncrementer(forIncrement amount: Int)
-> Void -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

```

Возвращаемый тип `makeIncrementer Void -> Int`. Это значит, что он возвращает *функцию*, а не простое значение. Функцию, которую она возвращает не имеет параметров и возвращает `Int` каждый раз как ее вызывают. Узнать как функции могут возвращать другие функции можно в главе ["Функциональные типы"](#).

Функция `makeIncrementer (forIncrement:)` объявляет целочисленную переменную `runningTotal`, для хранения текущего значения инкремента, которое будет возвращено. Переменная инициализируется значением 0.

Функция `makeIncrementer (forIncrement:)` имеет единственный параметр `Int` с внешним именем `forIncrement` и локальным именем `amount`. Значение аргумента передается этому параметру, определяя на сколько должно быть увеличено значение `runningTotal` каждый раз при вызове функции.

Функция `makeIncrementer` объявляет вложенную функцию `incrementer`, которая непосредственно и занимается увеличением значения. Эта функция просто добавляет `amount` к `runningTotal` и возвращает результат.

Если рассматривать функцию `incrementer()` отдельно, то она может показаться необычной:

```
func incrementer() -> Int {  
    runningTotal += amount  
    return runningTotal  
}
```

Функция `incrementer()` не имеет ни одного параметра и она ссылается на `runningTotal` и `amount` внутри тела функции. Она делает это, захватывая существующие значения от `runningTotal` и `amount` из окружающей функции и используя их внутри.

Так как она не изменяет `amount`, функция `incrementer` захватывает и хранит копию значения, хранимого в `amount`. Это значение хранится вместе с новой функцией `incrementer`.

Однако из-за того, что она изменяет значение переменной `runningTotal` каждый раз как вызывается, `incrementer` захватывает ссылку к текущему значению переменной `runningTotal`, а не копию ее исходного значения. Захват ссылки дает гарантию того, что `runningTotal` не исчезнет при окончании вызова `makeIncrementer` и гарантирует, что `runningTotal` останется переменной в следующий раз, когда будет вызвана функция `incrementer()`.

Заметка

Swift определяет, что должно быть захвачено по ссылке, а что по копии значения. Вам не нужно указывать, что `amount` или `runningTotal` будут использованы внутри вложенной функции `incrementer()`. Swift также берет на себя управление памятью по размещению `runningTotal`, когда она уже будет больше не нужна в функции `incrementer()`.

Приведем пример `makeIncrementor` в действии:

```
let incrementByTen =  
makeIncrementer(forIncrement: 10)
```

Этот пример заставляет константу `incrementByTen` ссылаться на функцию инкремента, которая добавляет `10` к значению переменной `runningTotal` каждый раз как вызывается.

Многочастный вызов функции показывает ее в действии:

```
incrementByTen()  
// возвращает 10  
incrementByTen()  
// возвращает 20  
incrementByTen()  
// возвращает 30
```

Если вы создаете второй инкрементор, он будет иметь свою собственную ссылку на новую отдельную переменную `runningTotal`:

```
let incrementBySeven =  
makeIncrementer(forIncrement: 7)  
incrementBySeven()  
//возвращает значение 7
```

Повторный вызов первоначального инкремента (`incrementByTen`) заставит увеличиваться его собственную переменную `runningTotal` и никак не повлияет на переменную, захваченную в `incrementBySeven`:

```
incrementByTen()  
//возвращает 40
```

Заметка

Если вы присваиваете замыкание как свойство экземпляра класса, и оно захватывает экземпляр по ссылке на экземпляр или его члены, вы создаете сильные обратные связи между экземпляром и замыканием. Swift использует списки захвата, для разрыва этих сильных обратных связей.

Замыкания являются ССЫЛОЧНЫМ ТИПОМ

В примере выше `incrementBySeven` и `incrementByTen` константы, но замыкания, на которые ссылаются эти константы имеют возможность увеличивать переменные `runningTotal`, которые они захватили. Это из-за того, что функции и замыкания являются *ссылочными типами*.

Когда бы вы не присваивали функцию или замыкание константе или переменной вы фактически присваиваете ссылку этой константе или переменной на эту функцию или замыкание. В примере выше выбор замыкания, на которое ссылается `incrementByTen`, константа, но не содержимое самого замыкания.

Это так же значит, что если вы присвоите замыкание двум разным константам или переменным, то оба они будут ссылаться на одно и то же замыкание:

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
//возвращает 50
```


Перечисления

Перечисления определяют общий тип для группы связанных значений и позволяют работать с этими значениями в типобезопасном режиме в вашем коде.

Если вы знакомы с C, то вы знаете, что перечисления в C присваивают соответствующие имена набору целочисленных значений. Перечисления в Swift более гибкий инструмент и не должны предоставлять значения для каждого члена перечисления. Если значение (известное как “сырое” значение) предоставляется каждому члену перечисления, то это значение может быть строкой, символом или целочисленным значением, числом с плавающей точкой.

Кроме того, членам перечисления можно задать соответствующие значения любого типа, которые должны быть сохранены вместе с каждым другим значением члена. Вы можете определить общий набор соответствующих значений как часть одного перечисления, каждый из которых будет иметь разные наборы значений соответствующих типов связанными с ними.

Перечисления в Swift - типы “первого класса”. Они принимают обладают особенностями, которые обычно поддерживаются классами, например, вычисляемые свойства, для предоставления дополнительной информации о текущем значении перечисления, методы экземпляра для дополнительной функциональности, относящейся к значениям, которые предоставляет перечисление.

Перечисления так же могут объявлять инициализаторы для предоставления начального значения элементам. Они так же могут быть расширены для расширения своей функциональности над своей начальной имплементацией. Могут соответствовать протоколам для обеспечения стандартной функциональности.

Синтаксис перечислений

Перечисления начинаются с ключевого слова `enum`, после которого идет имя перечисления и полное его определение в фигурных скобках:

```
enum SomeEnumeration {  
    //здесь будет объявление перечисления  
}
```

Ниже пример с четырьмя сторонами света:

```
enum CompassPoint {  
    case North  
    case South  
    case East  
    case West  
}
```

Значения, которые объявлены в перечислении (`North, South, East, West`) называются *членами значений* (или *членами*) этого перечисления. Ключевое слово `case` показывает, что новая строка значений члена будет сейчас объявлена.

Заметка

В отличие от C и Objective-C в Swift членам перечисления не присваиваются целочисленные значения по умолчанию при их создании. В примере выше `CompassPoint`, значения членов `North, South, East, West` неявно не равны `0, 1, 2, 3`. Вместо этого различные члены перечисления по праву полностью самостоятельны, с явно объявленным типом `CompassPoint`.

Множественные значения члена перечисления могут записываться в одну строку, разделяясь между собой запятой:

```
enum Planet {  
    case Mercury, Venus, Earth, Mars, Jupiter,  
    Saturn, Uranus, Neptune  
}
```

Каждое объявление перечисления объявляет и новый тип. Как и остальные типы в Swift, их имена (к примеру `CompassPoint` и `Planet`) должны начинаться с заглавной буквы. Имена перечислениям лучше давать особенные, а не те, которые вы можете использовать в нескольких местах, так чтобы они читались как само собой разумеющиеся:

```
var directionToHead = CompassPoint.West
```

Тип `directionToHead` выведен при инициализации одного из возможных значений `CompassPoint`.

Если `directionToHead` объявлена как `CompassPoint`, то можем использовать различные значения `CompassPoint` через сокращенный точечный синтаксис:

```
directionToHead = .East
```

Тип `directionToHead` уже известен, так что вы можете не указывать тип, присваивая значения. Так делается для хорошо читаемого кода, когда работаете с явно указанными типами значений перечисления.

Использование перечислений с оператором switch(сочетание значений с ...)

Вы можете сочетать индивидуальные значения перечисления с оператором switch:

```
directionToHead = .South
```

```
switch directionToHead {
case .North:
    print("Lots of planets have a north")
case .South:
    print("Watch out for penguins")
case .East:
    print("Where the sun rises")
case .West:
    print("Where the skies are blue")
}
//ВЫВОДИТ "Watch out for penguins"
```

Вы можете прочитать этот код как:

“Рассмотрим значение directionToHead. В случае, когда directionToHead равняется .North, выводится сообщение “Lots of planets have a north”. В случае, когда оно равняется .South, выводится сообщение “Watch out for penguins”.

...и так далее...

Как сказано в главе “[Управление потоком](#)”, оператор switch должен быть исчерпывающим, когда рассматриваются члены перечисления. Если мы пропустим случай case .West, то код не скомпилируется, так как не

рассматривается полный перечень членов `CompassPoint`. Требования к конструкции быть исчерпывающей, помогает случайно не пропустить член перечисления.

Если не удобно описывать случай для каждого члена перечисления, то вы можете использовать случай `default`, для закрытия всех остальных вариантов перечисления:

```
let somePlanet = Planet.Earth
switch somePlanet {

case .Earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// напечатает "Mostly harmless"
```

Связанные значения

Примеры в предыдущей секции показывают, как члены перечисления по праву считаются объявленными значениями. Вы можете установить `Planet.Earth` как константу или переменную и посмотреть какое значение она содержит. Однако бывает удобно хранить связанные значения других типов вместе с этими значениями членов перечисления. Это позволяет вам хранить дополнительную пользовательскую информацию вместе со значением члена и разрешает изменять эту информацию каждый раз как вы используете этот член перечисления в вашем коде.

Вы можете объявить перечисления Swift для хранения связанных значений любого необходимого типа, и типы значений могут отличаться для каждого члена перечисления, если это необходимо. Перечисления такого типа так же известны как размеченные объединения, меченные

объединения или варианты в других языках программирования.

Для примера, предположим систему инвентаризации, которая должна отслеживать товар двумя различными типами штрих-кодов. Одни товары имеют коды типа 1D формата UPC-A, которые используют цифры от 0 до 9. Каждый штрих-код имеет свою “систему цифр”, где идут пять цифр “кода производителя” и пять цифр “кода продукта”. Затем идет “проверочная” цифра, которая проверяет, что код был отсканирован корректно:



Другие продукты имеют маркировку штрих-кодом 2D формата QR, который может использовать любой символ из ISO 8859–1 и может закодировать строку длиной 2953 символа:



Было бы удобно, если бы система контроля и учета товара могла бы хранить штрих-коды формата UPC-A, как кортеж из четырех целых чисел и QR код, как строку любой длины.

В Swift перечисления для определения штрих-кода продукта одного из двух типов может выглядеть следующим образом:

```
enum Barcode {  
    case UPCA(Int, Int, Int, Int)  
    case QRCode(String)  
}
```

Читается это вот так:

“Объявление перечисления типа `Barcode`, которое берет два значения, одно из которых `UPCA`, со связанным значением типа `(Int, Int, Int, Int)` и значение `QRCode` со связанным значением типа `(String)`.”

Объявление не дает никакого значения типа `Int` или `String`, оно лишь определяет типы связанных значений, которые константы или переменные `Barcode` могут содержать, когда они равны `Barcode.UPCA`, `Barcode.QRCode`.


```
var productBarcode = Barcode.UPCA(8, 85909,
51226, 3)
```

В этом примере мы создаем новую переменную `productBarcode` и присваиваем ей значение `Barcode.UPCA` со связанным кортежем значений `(8, 85099, 3233, 1)`.

Этому же продукту может быть присвоено другое значение кода:

```
productBarcode = .QRCode("ABCDEFGH IJKLMNOP")
```

Здесь исходный `Barcode.UPCA` и его целочисленные значения заменены новым `Barcode.QRCode` и его строковым значением. Константы и переменные типа `Barcode` могут хранить или `.UPCA`, или `.QRCode` (вместе со связанными значениями), но они могут хранить только один из них в любое время.

Различные типы штрих-кодов могут быть проверены конструкцией `switch` как и раньше. В этот раз связанные значения могут быть извлечены как часть конструкции `switch`. Вы извлекаете каждое связанное значение как константу (с префиксом `let`) или как переменную (префикс `var`) для использования внутри тела оператора `switch`:

```
switch productBarcode {
case .UPCA(let numberSystem, let manufacturer,
let product, let check):
    print("UPC-A: \(numberSystem),
\ (manufacturer), \ (product), \ (check) .")
case .QRCode(let productCode):
    print("QR code: \(productCode) .")
}
//выведет "QR code: ABCDEFGH IJKLMNOP"
```

Если все связанные значения для членов перечисления извлекаются как константы или переменные, то для краткости вы можете разместить одиночное `let` или `var` перед именем члена:

```
switch productBarcode {
case let .UPCA(numberSystem, manufacturer,
product, check):
    print("UPC-A: \(numberSystem),
\ (manufacturer), \(product), \(check).")
case let .QRCode(productCode):
    print("QR code: \(productCode).")
}
//выведет "QR code: ABCDEFGHIJKLMNOP"
```

Исходные значения

В примере с `barcode` в главе “Связанные значения” можно увидеть как члены перечисления могут объявлять значения различных типов, которые они могут хранить. Как альтернатива связанным значениям, члены перечисления могут иметь начальные значения (называются “исходными значениями”), которые все одного типа.

Вот пример перечисления, члены которого хранят исходные значения ASCII, прописанные рядом:

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

Исходные значения

перечисления `ASCIIControlCharacter` определены как тип `Character`, и им присвоены распространенные контрольные символы ASCII, которые описаны в разделе [“Строки и символы”](#).

Обратите внимание на то, что исходные значения *не то же самое*, что связанные значения. Исходные значения устанавливаются при объявлении перечисления. Исходное значение для конкретного члена перечисления всегда одно и то же. Связанные значения присваиваются, когда вы создаете новую константу или переменную, основанную на одном из членов перечисления и могут быть разными каждый раз, когда вы это делаете.

Исходные значения могут быть строками, символам или любым числовым типом. Каждое исходное значение должно быть уникальным при его объявлении. Когда в качестве исходных значений используются целые числа, они автоматически инкрементируются, если никакой конкретной величины для члена не указано.

Перечисление внизу - улучшенная версия описанного ранее перечисления `Planet`, только уже с исходными значениями, для обозначения порядкового номера планеты по удаленности от солнца:

```
enum Planet: Int {  
    case Mercury = 1, Venus, Earth, Mars,  
    Jupiter, Saturn, Uranus, Neptune  
}
```

Автоматическое увеличение значения значит, что `Planet.Venus` имеет исходное значение равное 2 и так далее.

Для доступа к исходному значению члена перечисления существует свойство `rawValue`:

```
let earthsOrder = Planet.Earth.rawValue
//earthsOrder равен 3
```

Инициализация через исходное значение

Если вы объявили перечисление вместе с типом исходного значения, то перечисление автоматически получает инициализатор, который берет значение типа исходного значения (как параметр `rawValue`) и возвращает либо член перечисления либо `nil`.

В этом примере `Uranus` инициализируется через его исходное значение 7:

```
let possiblePlanet = Planet(rawValue: 7)
//possiblePlanet типа Planet? и равняется
Planet.Uranus
```

Конечно не все возможные значения `Int` найдут отклик в данном перечислении. Из-за этого инициализаторы исходных значений всегда возвращают опциональный член перечисления. В этом примере `possiblePlanet` типа `Planet?` или “опциональный `Planet`”.

Заметка

Инициализатор исходного значения - проваливающийся инициализатор, потому как не каждое исходное значение будет возвращать член перечисления.

Если вы попытаетесь найти планету с номером позиции 9, то значение опциональной `Planet` вернет значение `nil`:

```

let positionToFind = 9

if let somePlanet = Planet(rawValue:
positionToFind) {
    switch somePlanet {
    case .Earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position
\ (positionToFind) ")
}
//выведет "There isn't planet at position 9"

```

Этот пример использует привязку опционалов для попытки добраться до `Planet` с исходным значением 9. Выражение `if let somePlanet = Planet(rawValue: 9)` создает опциональную `Planet` и устанавливает значение `somePlanet` опциональной `Planet`. В этом случае невозможно добраться до планеты с позицией 9, таким образом срабатывает ветка `else`.

Классы и структуры

Классы и структуры являются универсальными и гибкими конструкциями, которые станут строительными блоками для кода вашей программы. Для добавления функциональности в классах и структурах можно объявить свойства и методы, применив то же синтаксис, как и при объявлении констант, переменных и функций.

В отличие от других языков программирования, Swift не требует создавать отдельные файлы для интерфейсов и реализаций пользовательских классов и структур. В Swift, вы объявляете структуру или класс в одном файле, и внешний интерфейс автоматически становится доступным для использования в другом коде.

Заметка

Экземпляр класса традиционно называют объектом. Тем не менее, классы и структуры в Swift гораздо ближе по функциональности, чем в других языках, и многое в этой главе описывает функциональность, которую можно применить к экземплярам и класса и структуры. В связи с этим, употребляется более общий термин - экземпляр.

Сравнение классов и структур

Классы и структуры в Swift имеют много общего. В них обоих можно:

- Объявлять свойства для хранения значений
- Объявлять методы, чтобы обеспечить функциональность
- Объявлять индексы, чтобы обеспечить доступ к их значениям, через синтаксис индексов
- Объявлять инициализаторы, чтобы установить их первоначальное состояние
- Они оба могут быть расширены, чтобы расширить их функционал за пределами стандартной реализации
- Они оба могут соответствовать протоколам, для обеспечения стандартной функциональности определенного типа
- Для получения дополнительной информации смотрите главы Свойства, Методы, Индексы, Инициализаторы, Расширения и Протоколы.

Классы имеют дополнительные возможности, которых нет у структур:

- Наследование позволяет одному классу наследовать характеристики другого
- Приведение типов позволяет проверить и интерпретировать тип экземпляра класса в процессе выполнения
- Деинициализаторы позволяют экземпляру класса освободить любые ресурсы, которые он использовал
- Подсчет ссылок допускает более чем одну ссылку на экземпляр класса
- Для получения дополнительной информации смотрите Наследование, Приведение типов, Деинициализаторы и Автоматический подсчет ссылок.

Заметка

Структуры всегда копируются, когда они передаются в вашем коде, и при этом не используют подсчета ссылок.

Синтаксис объявления

Классы и структуры имеют схожий синтаксис объявления. Для объявления классов, используйте ключевое слово `class`, а для структур - ключевое слово `struct`:

```
class SomeClass {  
    // объявление класса будет здесь  
}  
struct SomeStructure {  
    // объявление структуры будет здесь  
}
```

Заметка

Что бы вы не создавали, новый класс или структуру, вы фактически создаете новый тип в Swift. Назначайте имена типов используя ВерхнийГорбатый Регистр (**SomeClass** или **SomeStructure**), чтобы соответствовать стандартам написания имен типов в Swift (например, **String**, **Int** и **Bool**). С другой стороны, всегда назначайте свойствам и методам имена в нижнемГорбатымРегистре (например, `frameRate` и `incrementCount`), чтобы отличить их от имен типов.

Пример определения структуры и класса:

```
struct Resolution {  
    var width = 0  
    var height = 0  
}  
class VideoMode{  
var resolution = Resolution()  
    var interlaced = false  
    var frameRate = 0.0  
    var name: String?  
}
```

Пример выше объявляет новую структуру Resolution для описания разрешения монитора в пикселях. Эта структура имеет два свойства **width**, **height**. Хранимые свойства - или константы или переменные, которые сгруппированы и сохранены в рамках класса или структуры. Этим свойствам выведен тип Int, так как мы им присвоили целочисленное значение 0.

В примере мы также объявили и новый класс VideoMode, для описания видеорежима для отображения на видеодисплее. У класса есть четыре свойства в виде переменных. Первое - resolution, инициализировано с помощью экземпляра структуры Resolution, что выводит тип свойства как Resolution. Для остальных трех свойств новый экземпляр класса будет инициализирован с interlaced = false, frameRate = 0.0 и опциональным значением типа String с названием name. Это свойство name автоматически будет иметь значение nil или "нет значения для name", потому что это опциональный тип.

Экземпляры класса и структуры

Объявление структуры `Resolution` и класса `VideoMode` только описывают как `Resolution` и `VideoMode` будут выглядеть. Сами по себе они не описывают специфическое разрешение или видеорежим. Для того чтобы это сделать нам нужно создать экземпляр структуры или класса.

Синтаксис для образования экземпляра класса или структуры очень схож:

```
let someResolution = Resolution()  
let someVideoMode = VideoMode()
```

И классы и структуры используют синтаксис инициализатора для образования новых экземпляров. Самая простая форма синтаксиса инициализатора - использование имени типа и пустые круглые скобки сразу после него `Resolution()`, `VideoMode()`. Это создает новый экземпляр класса или структуры с любыми инициализированными свойствами с их значениями по умолчанию.

Доступ к свойствам

Вы можете получить доступ к свойствам экземпляра используя точечный синтаксис. В точечном синтаксисе имя свойства пишется сразу после имени экземпляра, а между ними вписывается точка `(.)` без пробелов:

```
println("The width of someResolution is  
\ (someResolution.width)")  
//выведет "The width of someResolution is 0"
```

В этом примере `someResolution.width` ссылается на свойство `width` экземпляра `someResolution`, у которого начальное значение равно 0.

Вы можете углубиться в подсвойства, например, свойство `width` свойства `resolution` класса `VideoMode`:

```
println("The width of someVideoMode is  
\ (someVideoMode.resolution.width)")  
//выведет "The width of someVideoMode is 0"
```

Вы так же можете использовать точечный синтаксис для присваивания нового значения свойству:

```
someVideoMode.resolution.width = 1280  
println("The width of someVideoMode is now  
\ (someVideoMode.resolution.width)")  
//выведет "The width of someVideoMode is now  
1280"
```

Заметка

В отличие от Objective-C, в Swift вы можете устанавливать подсвойства структуры напрямую. В последнем примере выше свойство `width` свойства `resolution` экземпляра класса `someVideoMode` устанавливается напрямую, без необходимости менять все свойство `property` на новое значение.

Позлементные инициализаторы структурных типов

Все структуры имеют автоматически сгенерированный "позлементный инициализатор", который вы можете использовать для инициализации свойств новых экземпляров структуры. Начальные значения для свойств нового экземпляра могут быть переданыazoleментному инициализатору по имени:

```
let vga = Resolution(width: 640, height: 480)
```

В отличие от структур, классы не получилиazoleментного инициализатора исходных значений.

Структуры и перечисления – типы значений

Тип значения - это тип, значение которого копируется, когда оно присваивается константе или переменной, или когда передается функции.

Вообще вы уже достаточно активно использовали типы на протяжении предыдущих глав. Но факт в том, что все базовые типы Swift - типы значений и имплементированы они как структуры.

Все структуры и перечисления - типы значений в Swift. Это значит, что любой экземпляр структуры и перечисления, который вы создаете, и любые типы значений, которые они имеют в качестве свойств, всегда копируется, когда он передается по вашему коду.

Рассмотрим пример, который использует структуру `Resolution` из предыдущего примера:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

Этот пример объявляет константу `hd` и присваивает экземпляр `Resolution`, инициализированное двумя значениями `width` и `height`.

В свою очередь объявляем переменную `cinema` и присваиваем ей текущее значение `hd`. Так как `Resolution` - структура, делается копия существующего экземпляра, и эта новая копия присваивается `cinema`. Даже не смотря на то, что `hd` и `cinema` имеют одни и те же `height`, `width`, они являются абсолютно разными экземплярами.

Следующим шагом изменим значение свойства `width` у `cinema`, мы сделаем его чуть больше 2 тысяч, что является стандартным для цифровой кинопроекции (2048 пикселей ширины на 1080 пикселей высоты):

```
cinema.width = 2048
```

Если мы проверим свойство `width` у `cinema`, то мы увидим, что оно на самом деле изменилось на `2048`:

```
print("cinema is now \(cinema.width) pixels  
wide")  
//выведет "cinema is now 2048 pixel wide"
```

Однако свойство `width` исходного `hd` экземпляра
осталось `1920`:

```
print("hd is still \(hd.width) pixels wide")  
//выведет "hd is still 1920 pixels wide"
```

Когда мы присвоили `cinema` текущее значение `hd`, то значения, которые хранились в `hd` были скопированы в новый экземпляр `cinema`. И в качестве результата мы имеем два совершенно отдельных экземпляра, которые содержат одинаковые числовые значения. Так как они являются отдельными экземплярами, то установив значение свойства `width` у `cinema` на `2048` никак не влияет на значение `width` у `hd`.

То же поведение применимо к перечислениям:

```
enum CompassPoint {  
    case North, South, East, West  
}  
var currentDirection = CompassPoint.West  
let rememberedDirection = currentDirection  
currentDirection = .East  
if rememberedDirection == .West {  
    print("The remembered direction is still  
.West")  
}  
//выведет "The remembered direction is still  
.West"
```

Когда мы присваиваем `rememberedDirection` значение `currentDirection`, мы фактически копируем это значение. Изменяя значение `currentDirection`, мы не меняем копию исходного значения, хранящейся в `rememberedDirection`.

Классы – ссылочный тип

В отличие от типа значений, *ссылочный тип не копируется*, когда его присваивают переменной или константе, или когда его передают функции. Вместо копирования используется ссылка на существующий экземпляр.

Вот пример с использованием класса `VideoMode`, который был объявлен выше:

```
let tenEighty = VideoMode()  
tenEighty.resolution = hd  
tenEighty.interlaced = true  
tenEighty.name = "1080i"  
tenEighty.frameRate = 25.0
```

В этом примере объявляем новую константу `tenEighty` и устанавливаем ссылаться на новый экземпляр класса `VideoMode`. Значения видеорежима были присвоены копией со значениями `1920` на `1080`. Мы ставим `tenEighty.interlaced = true` и даем имя `"1080i"`. Наконец то устанавливаем частоту кадров `25` кадров в секунду.

Следующее, что мы сделаем, это `tenEighty` присвоим новой константе `alsoTenEighty` и изменим частоту кадров на `30.0`:

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Так как классы ссылочного типа, то экземпляры `tenEighty` и `alsoTenEighty` ссылаются на один и тот же экземпляр `VideoMode`. Фактически получается, что у нас два разных имени для одного единственного экземпляра.

Если мы проверим свойство `frameRate` у `tenEighty`, то мы увидим, что новая частота кадров `30.0`, которая берется у экземпляра `VideoMode`:

```
print("The frameRate property of tenEighty is
now \(tenEighty.frameRate)")
//выведет "The frameRate property of tenEighty
is now 30.0"
```

Обратите внимание, что `tenEighty` и `alsoTenEighty` объявлены как константы, а не переменными. Однако вы все равно можете менять `tenEighty.frameRate` и `alsoTenEighty.frameRate`, потому что значения `tenEighty` и `alsoTenEighty` сами по себе не меняются, так как они не «содержат» значение экземпляра `VideoMode`, а напротив, они лишь ссылаются на него. Это свойство `frameRate` лежащего в основе `VideoMode`, которое меняется, а не значения константы ссылающейся на `VideoMode`.

Операторы тождественности

Так как классы ссылочного типа, то есть возможность сделать так, чтобы несколько констант и переменных ссылались на один единственный экземпляр класса. (Такое поведение не применимо к структурам и перечислениями, так как они копируют значение, когда присваиваются константам или переменным или передаются функциям.)

Иногда бывает полезно выяснить ссылаются ли две константы или переменные на один и тот же экземпляр класса. Для проверки этого в Swift есть два оператора тождественности:

- Идентичен (`===`)
- Не идентичен (`!==`)

Можно использовать эти операторы для проверки того, ссылаются ли две константы или оператора на один и тот же экземпляр:

```
if tenEighty === alsoTenEighty {  
    print("tenEighty and alsoTenEighty refer  
to the same VideoMode instance.")  
}  
// напечатает "tenEighty and alsoTenEighty  
refer to the same VideoMode instance."
```

Обратите внимание, что «идентичность» (в виде трех знаков равенства, или `===`) не имеет в виду «равенство» (в виду двух знаков равенства, или `==`):

- Идентичность или тождественность значит, что две константы или переменные ссылаются на один и тот же экземпляр класса.

- Равенство значит, что экземпляры равны или эквивалентны в значении в самом обычном понимании «равны».

Когда вы объявляете свой пользовательский класс и структуру, вы сами решаете, что значит, что две значения «равны».

Указатели

Если у вас есть опыт работы в C, C++ или Objective-C, то вы может быть знаете, что эти языки используют указатели для ссылки на адрес памяти. В Swift константы и переменные, которые ссылаются на экземпляр какого-либо ссылочного типа, аналогичны указателям C, но это не прямые указатели на адрес памяти, и они не требуют от вас написания звездочки(*) для индикации того, что вы создаете ссылку. Вместо этого такие ссылки объявляются как другие константы или переменные в Swift.

Выбираем между классом и структурой

Вы можете использовать как классы так и структуры для объявления ваших пользовательских типов данных для использования их в качестве строительных блоков вашего программного кода.

Однако экземпляр структуры всегда передается по значению, а вот экземпляр класса - по ссылке. Это значит, что они подходят под разные задачи. В зависимости от того, какие конструкции данных вы рассматриваете и какую функциональность вы преследуете, вам следует решить как должны быть объявлена конструкция, как класс или как структура.

Следуя основной линии, рассмотрим некоторые условия, при которых нам следует создавать структуру:

- Основная цель структуры - инкапсуляция нескольких сравнительно простых значений данных.
- Очень логично предположить, что инкапсулированные значения будут скорее копированы, чем переданы по ссылке, когда вы назначаете или передаете экземпляр структуры.
- Любые свойства, хранящиеся в структуре, сами по себе типы значений, которые тоже будут скопированы, а не переданы по ссылке.
- Структуры не нуждаются в наследовании свойств или поведения от других существующих типов.

Примеры хороших кандидатов структур:

- Размер геометрической фигуры, возможно инкапсулирует `width`, `height` свойства, оба свойства типа `Double`.
- Способ обратиться к диапазону внутри серии, возможно инкапсулирование свойства `start` и свойства `length`, оба свойства типа `Int`.
- Точка в 3D координатной системе, возможно инкапсулирование свойств `x`, `y` и `z`, тип всех свойств `Double`.

Во всех остальных случаях объявляйте класс, создавайте экземпляры этого класса для того, чтобы у вас была возможность ссылаться на них по ссылке. На практике это значит, что самая распространенная конструкция данных будет классом, а не структурой.

Присваивание и копирование поведения для строк, массивов и словарей

В Swift типы `String`, `Array`, `Dictionaries` объявлены как структуры. Это значит, что строки, массивы и словари копируются, когда присваиваются новой константе или переменной, или когда они передаются функции.

Такое поведение отличается от `NSString`, `NSArray` и `NSDictionary` в Foundation, где они имплементированы как классы, а не структуры. Экземпляры `NSString`, `NSArray` и `NSDictionary` всегда присваиваются и передаются как ссылки на существующий экземпляр, а не как его копии.

Заметка

Описание выше относится к "копированию" строк, массивов, словарей. Поведение, которое вы видите в вашем коде всегда будет, будто ваше значение скопировали. Однако Swift делает текущую копию, когда это абсолютно необходимо. Swift управляет всеми процессами копирования величин, для гарантии оптимальной производительности, и вам не стоит избегать присваивания для упреждения этой оптимизации.

Свойства

Свойства связывают значения с определённым классом, структурой или перечислением. Хранимые свойства хранят значения константы или переменной как часть экземпляра, в то время расчетные свойства считают значения, а не хранят их. Расчетные свойства обеспечиваются классами, структурами или перечислениями. Хранимые свойства обеспечиваются только классами или структурами.

Хранимые и расчетные свойства обычно связаны с экземплярами конкретного типа. Однако свойства также могут быть связаны и с типом самим по себе. Такие свойства известны как свойства типа.

В дополнение вы можете объявить наблюдателя свойства для отслеживания изменений по значению свойства, которые может вызывать пользовательскую действия. Наблюдатели свойства могут быть добавлены к хранимым свойствам, которые вы объявили сами, и так же могут быть добавлены к свойствам, которые субкласс наследует у суперкласса.

Хранимые свойства

В самой простой форме хранимое свойство - константа или переменная, которая хранится как часть экземпляра определенного класса или структуры. Хранимые свойства могут быть или *переменными хранимыми свойствами* (начинаются с ключевого слова `var`), или *константными хранимыми свойствами* (начинается с ключевого слова `let`).

Вы можете присвоить значение по умолчанию для хранимого свойства как часть его определения. Вы так же можете присвоить начальное значение для хранимого свойства во время его инициализации. Это даже возможно для константных свойств, но об этом в следующих главах.

Пример ниже объявляет структуру `FixedLengthRange`, которая описывает диапазон целых чисел и длина которой не может быть изменена после установки первого значения:

```
struct FixedLengthRange {  
    var firstValue: Int  
    let length: Int  
}  
  
var rangeOfThreeItems =  
FixedLengthRange(firstValue: 0, length: 3)  
//диапазон чисел 0, 1, 2  
rangeOfThreeItems.firstValue = 6  
//сейчас диапазон чисел 6, 7, 8
```

Экземпляры `FixedTnegthRange` имеют переменное хранимое свойство `firstValue` и хранимое свойство в виде константы `length`. В примере выше свойство `length` инициализировано, когда мы создали новый

диапазон, который не может быть изменен после, так как это свойство константа.

Хранимые свойства константных экземпляров структуры

Если вы создаете экземпляр структуры и присваиваете его константе, то вы не можете модифицировать его свойства, даже если они объявлены как переменные:

```
let rangeOfFourItems =  
FixedLengthRange(firstValue: 0, length: 4)  
//теперь диапазон чисел выглядит как 0, 1, 2,  
3  
rangeOfFourItems.firstValue = 6  
// это вызовет ошибку, даже несмотря на то,  
что firstValue переменная
```

Из-за того, что `rangeOfFourItems` объявлена в качестве константы (ключевое слово `let`), то невозможно поменять свойство `firstValue`, даже несмотря на то, что это свойство переменная.

Такое поведение объясняется тем, что структура является *типом значений*. Когда экземпляр типа значений отмечен как константа, то все его свойства так же считаются константами.

Такое поведение не применимо к классам, так как они являются *ссылочным типом*. Если вы присваиваете экземпляр ссылочного типа константе, то он все еще может менять переменные свойства.

Свойства ленивого хранения

Свойство ленивого хранения - свойство, начальное значение которого не рассчитывается до первого использования.

Индикатор ленивого свойства - ключевое слово `lazy`.

Заметка

Всегда объявляйте свойства ленивого хранения как переменные (с помощью ключевого слова `var`), потому что ее значение может быть не получено до окончания инициализации. Свойства-константы всегда должны иметь значение до того, как закончится инициализация, следовательно они не могут быть объявлены как свойства ленивого хранения.

Ленивые свойства полезны, когда исходное значение свойства зависит от внешних факторов, значения которых неизвестны до окончания инициализации. Так же ленивые свойства полезны, когда начальное значение требует комплексных настроек или сложных вычислений, которые не должны быть проведены до того момента, пока они не понадобятся.

Пример ниже использует ленивое хранение свойства для избежания ненужной инициализации сложного класса. Этот пример объявляет два класса `DataImporter` и `DataManager`, но ни один из них мы не показали полностью:

```

class DataImporter {
    /*DataImporter - класс для импорта данных
    с внешних источников
    Считаем, что классу требуется большое
    количество времени для инициализации
    */
    var fileName = "data.txt"
    //класс DataImporter функционал данных
    будет описан тут
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    //класс DataManager обеспечит необходимую
    функциональность тут
}
let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
//экземпляр класса DataImporter для свойства
importer еще не создано

```

Класс `DataManager` хранит свойство `data`, которое инициализируется с новым пустым массивом значений типа `String`. Несмотря на то, что большая часть функциональности недоступна, цель класса `DataManager` - это управление и обеспечение доступа к этому массиву данных типа `String`.

Часть функциональности класса `DataManager` - это импорт данных из файла. Эта функциональность обеспечивается классом `DataImporter`, который, как мы подразумеваем, требует уйму времени для инициализации. Это может происходить из-за того, что экземпляр класса `DataImporter` должен открыть файл и прочитать его содержимое в памяти, когда `DataImporter` уже инициализирован.

Это возможно для экземпляра `DataManager` управлять своими данными даже не импортируя их из файла, так что нет такой надобности, как создавать экземпляр `DataImporter`, когда сам `DataManager` создан. Вместо этого логичнее создать экземпляр `DataImporter` тогда, когда он будет впервые востребован.

Так как он создан как `lazy` модификатор, экземпляр `DataImporter` для свойства `importer` создается только тогда, когда впервые к нему обращаются, например когда запрашивается свойство `fileName`:

```
print(manager.importer.fileName)
//экземпляр DataImporter для свойства importer
только что был создан
//выведет "data.txt"
```

Хранимые свойства и переменные экземпляра

Если у вас есть опыт работы с Objective-C, вы можете быть знаете, что существует два способа хранения значений и ссылок как часть экземпляра класса. В дополнение к свойствам вы можете использовать переменные экземпляра как резервное хранение для значений, хранимых в свойствах.

Swift унифицирует этот концепт в объявление простого свойства. Свойства Swift не имеют соответствующей переменной экземпляра, и резервное хранение для свойства не имеет прямого доступа. Такой подход позволяет избежать путаницы о том, как был осуществлен доступ к значению в различных контекстах и упрощает объявление свойства в одно окончательное выражение. Вся информация о свойстве, включая его имя, тип и характеристики управлением памятью, объявляется в одном месте, как часть определения типа.

Вычисляемые свойства

В дополнение к хранимым свойствам, классам, структурам и перечислениям можно добавить *вычисляемые свойства*, которые фактически не хранят значения. Вместо этого они предоставляют геттер и опциональный сеттер для получения и установки других свойств косвенно.

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x +
(size.width / 2)
            let centerY = origin.y +
(size.height / 2)
            return Point(x: centerX, y:
centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x -
(size.width / 2)
            origin.y = newCenter.y -
(size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y:
0.0),
```

```

        size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at
      (\(square.origin.x), \(square.origin.y))")
// выводит "square.origin is now at (10.0,
10.0)

```

Этот пример определяет три структуры для работы с геометрическими фигурами:

- `Point` инкапсулирует координаты (`x`, `y`).
- `Size` инкапсулирует `width`, `height`.
- `Rect` определяет прямоугольник по начальной точке и размеру.

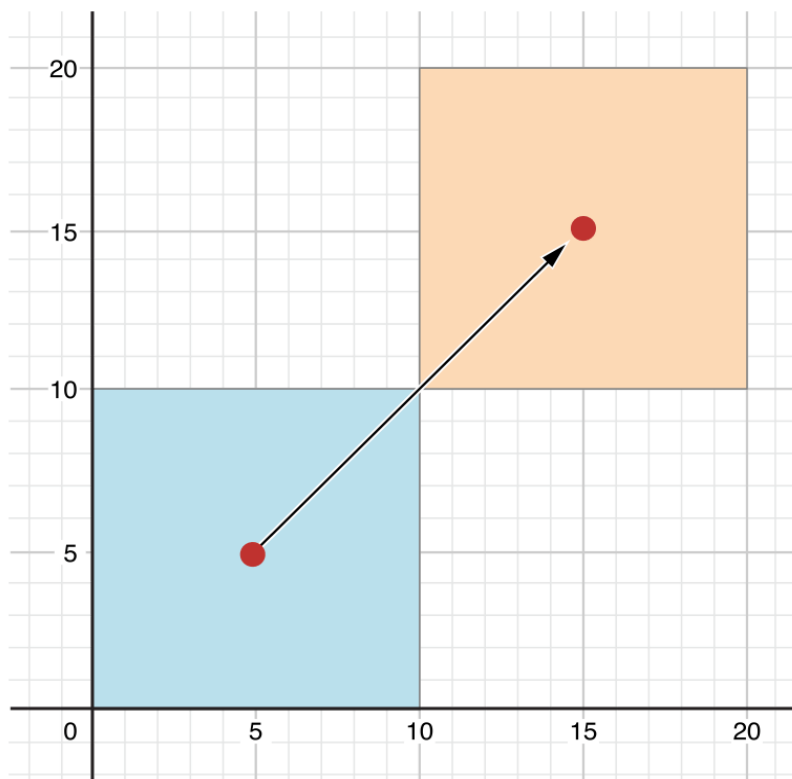
Структура `Rect` так же обеспечивает нас вычисляемым свойством `center`. Текущий центр `Rect` может быть определен из `origin` и `size`, так что вам не нужно хранить точку центра как явное значение `Point`. Вместо этого `Rect` объявляет пользовательский геттер и сеттер для вычисляемой переменной называемой `center`, для того чтобы была возможность работать с `center` прямоугольника, как если бы она была обычным хранимым свойством.

Предшествующий пример создает новую переменную `square` типа `Rect`. Переменная `square` инициализирована с начальной точкой `(0, 0)`, с высотой и шириной равными `10`. Этот квадрат представлен на диаграмме голубым цветом.

Свойство `center` переменной `square` доступно тогда, когда мы используем точечный синтаксис (`square.center`), что вызывает геттер для `center`, для получения текущего значения свойства. Вместо того, чтобы возвращать

существующее значение, геттер считывает текущее значение и возвращает новую `Point`, показывающую центр квадрата. Как вы можете видеть выше, геттер корректно возвращает точку центра как $(5, 5)$.

Потом свойство `center` устанавливает значение $(15, 15)$, что передвигает квадрат вверх и вправо, на новую позицию, как показано на диаграмме оранжевым квадратом. Установка свойства `center` вызывает сеттер для `center`, что обновляет значения `x`, `y` свойства `origin` и двигает квадрат на новую позицию.



Сокращенный вариант объявления сеттера

Если сеттер высчитываемого свойства не определяет имени для нового значения, то используется имя по умолчанию `newValue`. Альтернативный вариант структуры `Rect`, обладающей преимуществом сокращенного синтаксиса:

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x +
(size.width / 2)
            let centerY = origin.y +
(size.height / 2)
            return Point(x: centerX, y:
centerY)
        }
        set {
            origin.x = newValue.x -
(size.width / 2)
            origin.y = newValue.y -
(size.height / 2)
        }
    }
}
```

Вычисляемые свойства только для чтения

Вычисляемое свойство имеющее геттер, но не имеющее сеттера известно так же как вычисляемое свойство *только для чтения*. Такое вычисляемое свойство только для чтения возвращает значение и может быть доступно через точечный синтаксис, но не может изменить свое текущее значение.

Заметка

Вы должны объявлять вычисляемые свойства, включая вычисляемые свойства для чтения, как переменные свойства с ключевым словом `var`, потому что их значение не фиксировано. Ключевое слово `let` используется только для константных свойств, значение которых не может меняться, после того как было установлено как часть инициализации экземпляра.

Вы можете упростить объявление вычисляемых свойств только для чтения, удаляя ключевое слово `get` и его скобки:

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0,
height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is
\(fourByFiveByTwo.volume)")
// выводит "the volume of fourByFiveByTwo is
40.0"
```

Этот пример объявляет новую структуру `Cuboid`, которая представляет 3D прямоугольную коробку

`cwidth`, `height`, `depth` свойствами. Так же эта структура имеет свойство доступное только для чтения `volume`, которое считает и возвращает текущий объем кубойда. Никакого смысла делать `volume` значением установленным, так как будет не понятно какие значения `width`, `height` и `depth` должны быть использованы для конкретного значения объема. Тем не менее для кубойда полезно иметь вычисляемое свойство только для чтения, чтобы пользователи могли узнать текущий посчитанный объем.

Наблюдатели свойства

Наблюдатели свойства наблюдают и реагируют на изменения значения свойства. Наблюдатели свойства вызываются каждый раз, как присваивается новое значение свойству, даже если новое значение такое же как и старое.

Вы можете добавить наблюдатели свойства для любого хранимого свойства, которое вы объявили, кроме свойств ленивого хранения. Вы так же можете добавить наблюдателя для наследованного свойства (не важно вычисляемое оно или просто хранимое) путем переопределения свойства внутри подкласса.

Заметка

Вам не обязательно определять наблюдателей свойства для непереопределяемых вычисляемых свойств, так как вы можете наблюдать и реагировать на их изменения напрямую, через сеттер вычислительного свойства.

У вас есть опция определять один или оба этих наблюдателя свойства:

- `willSet` вызывается прямо перед сохранением значения
- `didSet` вызывается сразу после сохранения значения

Если вы реализуете наблюдатель `willSet`, то он передает новое значение свойства как константный параметр. Вы можете сами определить ему имя внутри реализации `willSet`. Если вы не станете указывать новое имя параметра и скобки внутри реализации, то параметр все равно будет доступен через имя параметра по умолчанию `newValue`.

Аналогично, если вы реализуете наблюдатель `didSet`, то ему будет передан параметр-константа, содержащий старое значение свойства. Вы можете задать имя параметру, но если вы этого не сделаете, то он все равно будет доступен через имя параметра по умолчанию `oldValue`.

Заметка

Наблюдатели `willSet` и `didSet` не будут вызваны, когда свойству будет установлено значение в инициализаторе, еще до того как наблюдатели будут назначены!

Вот пример наблюдатели `willSet` и `didSet` в действии. Пример ниже объявляет новый класс `StepCounter`, который следит за общим числом шагов, которые совершает человек во время прогулки. Этот класс может быть использован с входящими значениями от шагомера или другого счетчика шагов для отслеживания упражнений человека в течении всего рабочего дня.

```

class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to
\ (newTotalSteps) ")
        }
        didSet {
            if totalSteps > oldValue {
                print("Added \ (totalSteps -
oldValue) steps")
            }
        }
    }
}

let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps

```

Класс `StepCounter` объявляет свойство `totalSteps` типа `Int`. Это хранимое свойство с наблюдателями `willSet`, `didSet`.

Как уже говорилось, наблюдатели `willSet` и `didSet` вызываются при любом присваивании значения свойством, даже если это значение совпадает со старым.

В этом примере наблюдатель `willSet` использует пользовательский параметр `newTotalSteps` для

предстоящего нового значения. В этом примере он просто выводит на экран значение, которое будет установлено.

Наблюдатель `didSet` вызывается после того как `totalSteps` обновляется. Он сравнивает новое значение `totalSteps` со старым. Если общее количество шагов увеличилось, то выводится сообщение о том, сколько новых шагов было сделано. Наблюдатель `didSet` не предоставляет имени пользовательского параметра для старого значения, но по умолчанию это имя `oldValue`.

Заметка

Если вы присваиваете значение свойству внутри наблюдатель `didSet`, то новое значение заменит то, которое было только что установлено.

Глобальные и локальные переменные

Возможности, описанные выше, для вычисляемых и наблюдаемых свойств так же доступны в *глобальных переменных* и в *локальных переменных*. Глобальные переменные - переменные которые объявляются снаружи любой функции, метода, замыкания или контекста типа. Локальные переменные - переменные, которые объявляются внутри функции, метода или внутри контекста замыкания.

Глобальные и локальные переменные, с которыми вы столкнулись в предыдущих главах, были хранимые свойства. Хранимые переменные похожи на *хранимые свойства*, которые предоставляют хранилище для значения определенного типа и позволяют тому значению быть установленным и полученным.

Однако вы так же можете объявить *вычисляемые переменные* и объявить обозреватели для хранимых значений и в глобальной, и в локальной области своего действия. Вычисляемые переменные считают значение, вместо того, чтобы его хранить, и записываются они таким же образом как и вычисляемые свойства.

Заметка

Глобальные константы и переменные всегда являются вычисляемыми отложено, аналогично [свойствам ленивого хранения](#). В отличии от свойств ленивого хранения глобальные константы и переменные не нуждаются в маркере lazy.

Локальные константы и переменные никогда не вычисляются отложено.

Свойства типа

Свойства экземпляров - свойства которые принадлежат экземпляру конкретного типа. Каждый раз, когда вы создаете экземпляр этого типа, он имеет свои собственные свойства экземпляра, отдельные от другого экземпляра.

Вы так же можете объявить свойства, которые принадлежат самому типу, а не экземплярам этого типа. Будет всего одна копия этих свойств, и не важно сколько экземпляров вы создадите. Такие свойства называются свойствами типа.

Свойства типа полезны при объявлении значений, которые являются универсальными для всех экземпляров конкретного типа, как например свойство-константа, которое могут использовать все экземпляры (как например статическая константа в C), или свойство-переменная, которое хранит

значение и которое является глобальным для всех экземпляров данного типа (как статическая переменная в C).

Для типов значений (то есть для структур и перечислений), вы можете объявить хранимые и вычисляемые свойства. Для классов вы можете объявить только вычисляемые свойства.

Хранимые свойства типов для типов значений могут быть переменными или константами. Вычисляемые свойства всегда объявляются как переменные свойства, таким же способом, как и вычисляемые свойства экземпляра.

Заметка

В отличие от хранимых свойств экземпляра, вы должны всегда давать хранимым свойствам типов значение по умолчанию. Это потому, что тип сам по себе не имеет инициализатора, который мог бы присвоить значение хранимому свойству типа.

Синтаксис свойства типа

В C и в Objective-C вы объявляете статические константы и переменные связанные с типом как глобальные статические переменные. Однако в Swift, свойства типа записаны как часть определения типа, внутри его фигурных скобок, и каждое свойство ограничено областью типа, который оно поддерживает.

Чтобы объявить свойства типа используйте ключевое слово `static`. Для вычисляемых свойств типа для классов, вы должны использовать ключевое слово `class`, чтобы разрешать подклассам переопределение инструкций суперкласса. Пример ниже показывает синтаксис для хранимых и вычисляемых свойств типа:

```

struct SomeStructure {
    static var storedTypeProperty = "Some
value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some
value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some
value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var
overrideableComputedTypeProperty: Int {
        return 107
    }
}

```

Заметка

Высчитываемые (расчетные) свойства типа, примера выше, являются свойствами только для чтения, но вы можете определить их как редактируемые и читаемые вычисляемые свойства типа с помощью того же синтаксиса для вычисляемых свойств экземпляра.

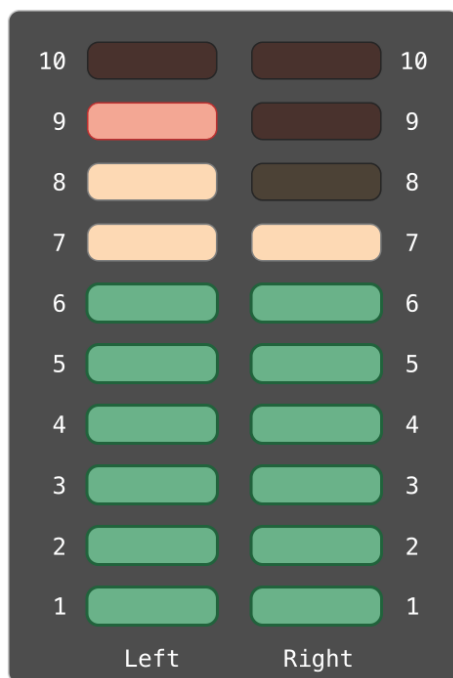
Запросы и установка свойств типа

Обращение к свойству типа и присваивание ему значения происходит с использованием точечного синтаксиса. Однако запрос и присваивание значения происходит в свойстве типа, а не в экземпляре того типа. К примеру:

```
print(SomeStructure.storedTypeProperty)
// напечатает "Some value."
SomeStructure.storedTypeProperty = "Another
value."
print(SomeStructure.storedTypeProperty)
// напечатает "Another value."
print(SomeEnumeration.computedTypeProperty)
// напечатает "6"
print(SomeClass.computedTypeProperty)
// напечатает "27"
```

Примеры, которые будут позже, используют два хранимых свойства типа как часть структуры, которая моделирует индикатор уровня громкости для звуковых каналов. Каждый канал имеет целочисленный уровень звука между 0 и 10 включительно.

На рисунке ниже показано, как два из этих звуковых канала могут быть объединены для моделирования стерео индикатора уровня звука. Когда уровень звука канала 0, ни один из огней для этого канала не горят, но когда уровень звука 10, все огни на этом канале горят. На этом рисунке, левый канал имеет текущий уровень 9, а правый канал в настоящее время имеет уровень 7:



Аудиоканалы, описанные выше, представлены экземплярами структуры `AudioChannel`:

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel >
AudioChannel.thresholdLevel {
                // cap the new audio level to
the threshold level
                currentLevel =
AudioChannel.thresholdLevel
            }
            if currentLevel >
AudioChannel.maxInputLevelForAllChannels {
                // store this as the new
overall maximum input level

AudioChannel.maxInputLevelForAllChannels =
currentLevel
            }
        }
    }
}
```

Структура `AudioChannel` объявляет два хранящихся свойства для поддержания функциональности. Первое - `thresholdLevel` определяет максимальное значение порога, которое звуковой уровень может воспроизвести. Это константное значение равно 10 для всех экземпляров `AudioChannel`. Если звуковой сигнал идет со значением выше чем 10, то он будет ограничен до порогового значения (как описано ниже).

Второе свойство типа - хранящееся свойство-переменная названная `maxInputLevelForAllChannels`. Оно отслеживает максимальное входное значение, которое было получено любым экземпляром `AudioChannel`. Исходное значение равно 0.

Структура `AudioChannel` так же определяет хранимое свойство `currentLevel` экземпляра, которое определяет текущий уровень аудиоканала в диапазоне от 0 до 10.

У свойства `currentLevel` есть обозреватель `didSet` для проверки значения `currentLevel`, всякий раз как оно присваивается. Этот наблюдатель выполняет две задачи:

- Если новое значение `currentLevel` больше, чем допущено `thresholdLevel`, наблюдатель свойства ограничивает `currentLevel` до уровня `thresholdLevel`.
- Если новое значение `currentLevel` (после возможных ограничений) больше, чем какое-либо другое, полученное экземпляром `AudioChannel` ранее, то наблюдатель свойства сохраняет значение нового `currentLevel` в статическое свойство `maxInputLevelForAllChannels`.

Заметка

В первой из этих двух проверок, наблюдатель `didSet` устанавливает `currentLevel` другое значение. Однако смена значения не заставляет вызывать наблюдателя еще раз!

Вы можете использовать структуру `AudioChannel` для создания двух новых аудиоканалов, названных `leftChannel`, `rightChannel`, для отображения уровней звука стерео системы:

```
var leftChannel = AudioChannel()  
var rightChannel = AudioChannel()
```

Если вы установите значение `leftChannel` на 7, то вы увидите, что значение свойства `maxInputLevelForAllChannels` обновится и станет равным 7:

```
leftChannel.currentLevel = 7  
print(leftChannel.currentLevel)  
// ВЫВОДИТ "7"  
print(AudioChannel.maxInputLevelForAllChannels  
)  
// ВЫВОДИТ "7"
```

Если вы попытаете установить значение правого канала `currentLevel` равным 11, то вы можете увидеть, что значение свойства правого канала `currentLevel` будет ограничено максимальным значением 10, и значение свойства типа `maxInputLevelForAllChannels` обновится и станет равным 10:

```
rightChannel.currentLevel = 11  
print(rightChannel.currentLevel)  
// ВЫВОДИТ "10"  
print(AudioChannel.maxInputLevelForAllChannels  
)  
// ВЫВОДИТ "10"
```

Методы

Методы - это функции, которые связаны с определенным типом. Классы, структуры и перечисления - все они могут определять методы экземпляра, которые включают в себя определенные задачи и функциональность для работы с экземпляром данного типа. Классы структуры и перечисления так же могут определить методы типа, которые связаны с самим типом. Методы типа работают аналогично методам класса в Objective-C.

Дело в том, что структуры и перечисления могут определить методы в Swift, что является главным отличием от C или Objective-C. В Objective-C классы единственный тип, который может определять методы. В Swift вы можете выбирать, стоит ли вам определять класс, структуру или перечисление, и вы все равно, при любом раскладе, получаете возможность определения методов типа, который вы создадите.

Методы экземпляра

Методы экземпляра являются функциями, которые принадлежат экземпляром конкретного класса, структуры или перечисления. Они обеспечивают функциональность этих экземпляров, либо давая возможность доступа и изменения свойств экземпляра, либо обеспечивая функциональность экземпляра в соответствии с его целью. Методы экземпляра имеют абсолютно одинаковый синтаксис как и функции, что описаны в “[Функции](#)”.

Вы пишете метод экземпляра внутри фигурных скобок типа, которому он принадлежит. Метод экземпляра имеет неявный доступ ко всем остальным методам экземпляра и свойствам этого типа. Метод экземпляра может быть вызван только для конкретного экземпляра типа, которому он принадлежит. Его нельзя вызвать в изоляции, без существующего экземпляра.

Ниже пример, который определяет простой класс `Counter`, который может быть использован для счета количества повторений действия:

```
class Counter {
    var count = 0
    func increment() {
        ++count
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

Класс `Counter` определяет три метода экземпляра:

- `increment` увеличивает значение счетчика на 1
- `incrementBy(amount: Int)` увеличивает значение счетчика на определенное значение `amount`.
- `reset` сбрасывает значение счетчика на 0.

Класс `Counter` так же определяет свойство-переменную `count`, для отслеживания значения счетчика.

Вы можете вызвать методы экземпляра с тем же точечным синтаксисом:

```
let counter = Counter()  
//начальное значение counter равно 0  
counter.increment()  
//теперь значение counter равно 1  
counter.incrementBy(5)  
//теперь значение counter равно 6  
counter.reset()  
//теперь значение counter равно 0
```

Локальные и внешние имена методов

Параметры функций имеют и локальные (используемые внутри функции) и внешние имена (для использования во время вызова функции), как подробно описано в главе [“Внешние имена параметров”](#). То же самое верно и для параметров метода, потому что методы - связанные функции с определенным типом. Однако стандартное поведение у локальных и внешних имен методов и функций различается.

Методы в Swift очень похожи на своих коллег в Objective-C. Так же как и в Objective-C имя метода в Swift относится к первому параметру метода с использованием предлогов `with`,

`for`, `by`, как вы видели в случае с методом `incrementBy(_:)` из примера класса `Counter`. Использование предлога в имени позволяет читать метод во время его вызова как предложение. Swift упрощает написание этого укоренившегося способа называния методов различными уловками по умолчанию для параметров метода, но не использует в параметрах функции.

К примеру, Swift дает имя первому параметру метода, и оно по умолчанию локальное, а остальным параметрам дает и *локальное*, и *внешнее* имя по умолчанию. Это соглашение соответствует обычному соглашению о именовании и вызове, с которыми вы познакомитесь при написании методов Objective-C, что делает вызов метода развернутым, без необходимости разъяснения значения ваших параметров.

Рассмотрим альтернативную версию класса `Counter`, который определяет более сложные формы метода `incrementBy(_:)`:

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int,
numberOfTimes: Int) {
        count += amount * numberOfTimes
    }
}
```

Метод `incrementBy(_:numberOfTimes:)` имеет два параметра: `amount`, `numberOfTimes`. По умолчанию Swift рассматривает имя параметра `amount` как локальное имя, но `numberOfTimes` и как локальное, и как внешнее. Вы можете вызвать метод вот так:


```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// сейчас значение counter равно 15
```

Вам не обязательно определять внешнее имя первого значения параметра, так как его цель понятна из названия имени функции `incrementBy(_ :numberOfTimes:)`. Второй аргумент, однако, определяется внешним именем параметра, которое делает его присутствие в методе понятным.

Такое стандартное поведение фактически относится к методу так, как будто вы написали символ решетки (#) до имени параметра `numberOfTimes`:

```
func incrementBy(amount: Int, #numberOfTimes:
Int) {
    count += amount * numberOfTimes
}
```

Стандартное поведение описанное выше значит то, что определения методов в Swift написаны точно так же как и в Objective-C, и вызываются в обычном, развернутом виде.

Изменение поведения имени внешнего параметр для методов

Иногда бывает полезно, чтобы первый параметр так же имел внешнее имя, даже если это нестандартное поведение. Вы можете либо добавить внешнее имя явно, либо использовать добавление решетки перед локальным именем, чтобы оно также использовалось и как внешнее.

Или же наоборот, если вы не хотите, чтобы ваше следующие параметры имели внешнее имя, то вы можете переписать

стандартное поведение, используя символ подчеркивания () в качестве внешнего имени параметра.

Свойство `self`

Каждый экземпляр типа имеет неявное свойство `self`, которое является абсолютным эквивалентом самому экземпляру. Вы используете свойство `self` для ссылки на текущий экземпляр, внутри методов этого экземпляра.

Метод `increment` может быть вызван так:

```
func increment() {  
    self.count++  
}
```

На практике вам не нужно писать `self` очень часто. Если вы не пишете `self`, то Swift полагает, что вы ссылаетесь на свойство или метод текущего экземпляра каждый раз, когда вы используете известное имя свойства или метода внутри метода. Этот вариант показан использованием свойства `count` (а не `self.count`) внутри трех методов `Counter`.

Главное исключение из этого правила получается, когда имя параметра экземпляра совпадает с именем свойства экземпляра. В этой ситуации имя параметра имеет приоритет и появляется необходимость ссылаться на свойство в более подходящей форме. Вы используете свойство `self` для того, чтобы увидеть различие между именем параметра и именем свойства.

Здесь `self` разграничивает параметр метода `x` и свойство экземпляра, которое тоже `x`:

```

struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x
    }
}

let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
    println("Эта точка находится справа от
линии, где x == 1.0")
}
// выводит "Эта точка находится справа от
линии, где x == 1.0"

```

Без префикса `self`, Swift будет думать, что в обоих случаях `x` - параметр метода, который мы вызываем.

Изменение типов значений методами экземпляра

Структуры и перечисления являются *типами значений*. По умолчанию, свойства типов значений не могут быть изменены внутри методов экземпляра.

Однако, если вам нужно изменить свойства вашей структуры или перечисления внутри конкретного метода, то вы можете выбрать поведение как изменяющееся для этого метода. После этого метод может изменить свои свойства внутри метода, и все изменения будут отменены, когда выполнение метода закончится. Метод так же может присвоить совершенно новый экземпляр для свойства `self`, и этот новый экземпляр заменить существующий, после того как выполнение метода закончится.

Вы можете все это осуществить, если поставите ключевое слово `mutating` перед словом `func` для определения метода:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y
deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
println("Сейчас эта точка на \(somePoint.x),
\ (somePoint.y) ")
//выводит "Сейчас эта точка на (3.0, 4.0)"
```

Структура `Point` определяет метод `moveByX`, который передвигает точку типа `Point` на определенное количество значений. Вместо того, чтобы вернуть новую точку, этот метод фактически изменяет координаты точки, которая его вызвала. Ключевое слово `mutating` добавлено к определению метода, для того, чтобы изменить значения свойств.

Обратите внимание, что вы не можете вызвать изменяющийся (`mutating`) метод для константных типов структуры, потому как ее свойства не могут быть изменены, даже если свойства являются переменными, что описано в главе “Хранимые свойства экземпляров структуры”

```
let fixedPoint = Point (x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y:3.0)
// это вызовет сообщение об ошибке
```

Присваивание значения для `self` внутри изменяющегося метода

Изменяющиеся методы могут присваивать полностью новый экземпляр неявному свойству `self`. Пример `Point`, приведенный выше, мог бы быть записан в такой форме:

```
struct Point {  
    var x = 0.0, y = 0.0  
    mutating func moveByX( deltaX: Double, y  
deltaY: Double) {  
        self = Point(x: x + deltaX, y: y +  
deltaY)  
    }  
}
```

Такая версия изменяющегося метода `moveByX` создает абсолютно новую структуру, чьим значениям `x`, `y` присвоены значения конечной точки. Конечный результат вызова этой альтернативной версии метода будет абсолютно таким же как и в ранней версии.

Изменяющиеся методы для перечислений могут установить отдельный член перечисления как неявный параметр `self`:

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
            case Off:
                self = Low
            case Low:
                self = High
            case High:
                self = Off
        }
    }
}
```

В этом примере мы рассматриваем перечисление с тремя положения переключателя. Переключатель проходит последовательно три положения (`Off`, `Low`, `High`), каждый раз меняя положение, как вызывается метод `next`.

Методы типа

Методы экземпляра, которые описаны выше, являются методами, которые вызываются экземпляром конкретного типа. Вы так же можете определить методы, которые вызываются самим типом. Такие методы зовутся *методами типа*. Индикатор такого метода - ключевое слово `class`, которое ставится до ключевого слова метода `func`, в случае классов, ну а в случае структур или перечислений ставится ключевое слово `static`, перед ключевым словом `func`.

Заметка

В Objective-C определить метода типа только для классов. В Swift вы можете создавать методы типа не только для классов, но и для структур и перечислений. Метод каждого типа ограничен самим типом, который он поддерживает.

Такие методы так же используют точечный синтаксис, как и методы экземпляра. Однако эти методы вы вызываете самим типом, а не экземпляром этого типа. Вот как вы можете вызвать метод самим классом `SomeClass`:

```
class SomeClass {  
  
    class func someTypeMethod() {  
        //здесь идет реализация метода  
    }  
}  
SomeClass.someTypeMethod()
```

Внутри тела метода типа неявное свойство `self` ссылается на сам тип, а не на экземпляр этого типа. Для структур и перечислений это значит, что вы можете использовать `self` для того, чтобы устранить

неоднозначность между `static` свойствами и параметрами метода, точно так же как вы делали для свойств экземпляра и параметров метода экземпляра.

Если обобщить, то любое имя метода и свойства, которое вы используете в теле метода типа, будет ссылаться на другие методы и свойства на уровне типа. Метод типа может вызвать другой метод типа с именем третьего метода, без использования какого-либо префикса имени типа. Аналогично методам типа, структуры и перечисления могут получить доступ к `static` свойствам, если используют имя этого `static` свойства, без написания префикса.

Пример ниже определяет структура с именем `LevelTracker`, которая отслеживает прогресс игрока на разных уровнях игры. Это одиночная игра, но может хранить информацию для нескольких игроков на одном устройстве.

Все уровни игры (кроме первого уровня) заблокированы, когда играют в первый раз. Каждый раз, заканчивая уровень, этот уровень открывается и у остальных игроков на устройстве. Структура `LevelTracker` использует `static` свойства и методы для отслеживания уровней, которые были разблокированы. Так же она отслеживает текущий уровень каждого игрока.


```

struct LevelTracker {

    static var highestUnlockedLevel = 1
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel {
            highestUnlockedLevel = level
        }
    }
    static func levelIsUnlocked(level: Int) ->
Bool {
        return level <= highestUnlockedLevel
    }
    var currentLevel = 1
    mutating func advanceToLevel(level: Int) -
> Bool {
        if LevelTracker.levelIsUnlocked(level)
    {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}

```

Структура `LevelTracker` следит за самым последним уровнем, который разблокировал игрок. Это значение лежит в свойстве типа (`static`) `highestUnlockedLevel`.

`LevelTracker` также определяет две функции для работы со свойством `highestUnlockedLevel`. Первая функция типа `unlockLevel`, которая обновляет значение `highestUnlockedLevel`, каждый раз когда открывается новый уровень. Вторая функция типа `levelIsUnlocked`, которая возвращает `true`, если конкретный уровень (число уровня) уже разблокирован.

(Обратите внимание, что методы типа могут получить доступ к `highestUnlockedLevel` без написания `LevelTracker.highestUnlockedLevel`.)

В дополнение к его свойствам типа и методам типа, структура `LevelTracker` также отслеживает и текущий прогресс игрока в игре. Она использует свойство экземпляра `currentLevel` для отслеживания уровня, на котором игрок играет.

Для помощи в управлении свойством `currentLevel`, структура `LevelTracker` определяет метод экземпляра `advanceToLevel`. До того как обновить `currentLevel`, этот метод проверяет доступен ли запрашиваемый новый уровень. Метод `advanceToLevel` возвращает логическое значение, указывающее, удалось ли ему поставить `currentLevel`.

Структура `LevelTracker` используется классом `Player`, который описан ниже, для отслеживания и обновления прогресса конкретного игрока:

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func completedLevel(level: Int) {
        LevelTracker.unlockLevel(level + 1)
        tracker.advanceToLevel(level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

Класс `Player` создает новый экземпляр `LevelTracker` для отслеживания прогресса игрока. Так же он определяет и использует метод `completedLevel`, который вызывается каждый раз, как игрок заканчивает уровень. Этот метод открывает следующий уровень для всех игроков. (Логическое значение `advanceToLevel` игнорируется, так как уровень открывается функцией `LevelTracker.unlockLevel` на предыдущей строке.)

Мы можете создать экземпляр класса `Player` для нового игрока и увидеть, что будет, когда игрок закончит первый уровень:

```
var player = Player(name: "Sheldon Kupper")
player.completedLevel(1)
println("Самый последний доступный уровень
сейчас равен
\ (LevelTracker.highestUnlockedLevel) ")
//выводит "Самый последний доступный уровень
сейчас равен 2"
```

Если вы создадите второго игрока, и попытаете им начать прохождение уровня, который не был разблокирован ни одним игроком в игре, то вы увидите, что эта попытка будет неудачной:

```
player = Player(name: "Leonard")

if player.tracker.advanceToLevel(6) {
    println("Игрок на уровне 6")
} else {
    println("Уровень 6 еще не разблокирован")
}
```

Индексы

Классы, структуры и перечисления могут определять индексы, которые являются сокращенным вариантом доступа к члену коллекции, списка или последовательности. Вы можете использовать индекс для получения или установки нового значения элемента без разделения этих двух методов (получения значения и установкой нового). К примеру, вы можете воспользоваться индексом в экземпляре массива для получения значения элемента `someArray[index]` или в экземпляре словаря `someDictionary[key]`.

Вы можете определить несколько индексов для одного типа, при необходимости загружается подходящий индекс, который выбирается в зависимости от типа значения индекса, который вы передаете в индекс. Индексы не ограничены одной размерностью, вы можете определить индексы с множественными вводами параметров для удовлетворения потребностей вашего пользовательского типа.

Индексный синтаксис

Индексы позволяют вам запрашивать экземпляры определенного типа, написав одно или несколько значений в квадратных скобках после имени экземпляра. Синтаксис индекса аналогичный синтаксису методу экземпляра и вычисляемому свойству. Вы пишете определения индекса с помощью ключевого слова `subscript` и указываете один или более входных параметров и возвращаемый тип, точно так же как и в методах экземпляра. В отличие от методов экземпляра, индексы могут быть `read-write` или `read-only`. Такое поведение сообщается геттом и сеттером в точности так же как и в вычисляемых свойствах:

```
subscript(index: Int) -> Int {  
    get {  
        //возвращает надлежащее значение  
скрипта  
    }  
    set(newValue) {  
        //проводит надлежащие установки  
    }  
}
```

Тип `newValue` такой же как и у возвращаемого значения индекса. Что же касается вычисляемых свойств, то вы можете не указывать параметр сеттера (`newValue`). Параметр по умолчанию называется `newValue` и предоставляется, если не было назначено другого.

Как и в случае с нередактируемыми (`read-only`) вычисляемыми свойствами, вы можете опустить слово `get` для нередактируемых (`read-only`) индексов:

```
subscript(index: Int) -> Int {  
    //возвращает надлежащее значение  
скрипта  
}
```

Пример определения нередактируемого индекса, который определяет структуру `TimesTable` показать таблицу умножения на `n3`:

```
struct TimesTable{  
    let multiplier: Int  
    subscript(index: Int) -> Int {  
        return multiplier * index  
    }  
}  
let threeTimesTable = TimesTable(multiplier:  
3)  
  
println("шесть умножить на три будет  
\(threeTimesTable[6])")  
//ВЫВОДИТ "шесть умножить на три будет 18"
```

В этом примере новый экземпляр `TimesTable` создан для отображения таблицы умножения на три. Это определяется переданным ему значением 3 в инициализатор структуры как значение для параметра экземпляра `multiplier`.

Вы можете запросить экземпляр `threeTimesTable` с помощью индекса, как уже сделано в примере выше `threeTimesTable[6]`. Он запрашивает шестую запись в таблице умножения на три, которая возвращает значение 18 или 6 умноженное на 3.

Заметка

Таблица умножения на n основана на фиксированном математическом правиле. Присваивание нового значения `threeTimesTable[someIndex]` не подходит для нашего варианта, значит индекс `TimesTable` определен как неизменяемый индекс.

Использование индекса

Точное значение “индекса” зависит от контекста, в котором он применяется. Обычно индексы используются в качестве сокращенного способа обращения к элементу коллекции, списка или последовательности. Вы свободны применять индексы в необходимой форме для вашего класса или для функциональности структуры.

К примеру, словарь в языке Swift использует индекс для присваивания или получения значения, которое храниться в экземпляре `Dictionary`. Вы можете задать значение в словаре, и используя ключ типа ключа словаря в квадратных скобках, присваивая значение типа словаря через индекс:

```
var numberOfLegs = ["паук": 8, "муравей": 6,
                    "кошка": 4]
numberOfLegs["птичка"] = 2
```

В примере выше мы объявляем переменную `numberOfLegs` и инициализируем ее с помощью литерала словаря, который содержит три пары ключ-значение. Тип словаря `numberOfLegs` выводится как `[String: Int]`. После того как словарь создан, в этом примере используется индексное присваивание для добавления ключа типа `String` “птичка”, значения типа `Int` “2”.

Для более подробного описания работы со словарями обратитесь к разделу “[Словари](#)”.

Заметка

Тип `Dictionary` в Swift осуществляет ключ-значение индексирование, как индекс, который получает опциональное значение. Для словаря `numberOfLegs`, индекс ключ-значение берет и возвращает значение типа `Int?` или “опциональный `Int`”. Тип `Dictionary` использует опциональный тип индекса, чтобы смоделировать факт того, что не каждый ключ может иметь значение, и для того, чтобы была возможность удаления значения для ключа, присваивая ему `nil`.

Опции индекса

Индексы могут принимать любое количество входных параметров, и эти параметры могут быть любого типа. Индексы так же могут возвращать любой тип. Индексы могут использовать параметры-переменные или вариативные параметры, но они не могут иметь сквозных параметров или обеспечивать исходные значения параметрам.

Класс или структура могут обеспечить столько индексных реализаций, сколько нужно, и подходящий индекс, который будет использоваться, будет выведен, основываясь на типе значения или значений, которые содержатся внутри индексных скобок, в том месте, где этот индекс используется. Определение множественных индексов так же известно как индексная перегрузка.

Сейчас в большинстве случаев индекс принимает один единственный параметр, вы так же можете определить индекс с несколькими параметрами, если этот вариант подходит для вашего типа. Следующий пример определяет структуру `Matrix`, которая представляет собой двухмерную

матрицу значений типа `Double`. Индекс структура `Matrix` принимает два целочисленных параметра.

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns,
repeatedValue: 0.0)
    }
    func isValidForRow(row: Int, column:
Int) -> Bool {
        return row >= 0 && row < rows && column >= 0
&& column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(isValidForRow(row, column:
column), "Index is out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(isValidForRow(row, column:
column), "Index is out of range")
            return grid[(row * columns) + column] =
newValue
        }
    }
}
```

`Matrix` предоставляет инициализатор, который принимает два параметра `rows` и `columns`, и создает массив типа `Double`, который имеет размер `rows * columns`. Каждой позиции в матрице дается начальное значение `0.0`. Чтобы этого достичь, размер массива и начальное значение клетки

равное 0.0 передаются в инициализатор массива, который создает и инициализирует новый массив необходимого размера. Это описано подробнее в главе [“Создание и инициализация массива”](#).

Вы можете создать новый экземпляр типа `Matrix`, передав количество рядов и столбцов в его инициализатор:

```
var matrix = Matrix(rows: 2, columns: 2)
```

Этот пример создает новый экземпляр `Matrix`, который имеет всего два ряда и два столбца. Массив `grid` для экземпляра `Matrix` фактически упрощенный вариант версии матрицы, который читается с левой верхней части в правую нижнюю часть:

$$\text{grid} = \begin{bmatrix} 0.0, & 0.0, & 0.0, & 0.0 \end{bmatrix}$$

			column	
			0	1
row	0	$\begin{bmatrix} \begin{bmatrix} 0.0, & 0.0, \\ 0.0, & 0.0 \end{bmatrix} \end{bmatrix}$		
	1			

Значения в матрице могут быть установлены через передачу значений ряда и столбца в индексе, разделенных между собой запятой:

```
matrix[0, 1] = 1.5  
matrix[1, 0] = 3.2
```

Эти два выражения в сеттере индекса устанавливают значения 1.5 для верхней правой позиции (где `row` равен 0, `column` равен 1), и значение 3.2 для нижней левой позиции (где `row` равен 1, а `column` равен 0):

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

Геттер и сеттер индекса `Matrix` оба содержат утверждения для проверки валидности значений `row` и `column`. Для помощи утверждениям, `Matrix` имеет удобный метод под названием `indexIsValidForRow(_ : column:)`, который проверяет наличие запрашиваемых `row`, `column` в существующей матрице:

```
func indexIsValidForRow(row: Int, column: Int)  
-> Bool {  
    return row >= 0 && row < rows && column >= 0  
    && column < columns  
}
```

Утверждения срабатывают, если вы пытаетесь получить доступ к индексу, который находится за пределами матрицы:

```
let someValue = matrix[2, 2]  
//это вызывает утверждение, потому что [2, 2]  
находится за пределами матрицы
```

Наследование

Класс может *наследовать* методы, свойства и другие характеристики другого класса. Когда один класс наследует у другого класса, то наследующий класс называется *подклассом*, класс у которого наследуют - *суперклассом*. Наследование - фундаментальное поведение, которое отделяет классы от других типов Swift.

Классы в Swift могут вызывать или получать доступ к методам, свойствам, индексам, принадлежащим их суперклассам и могут предоставлять свои собственные переписанные версии этих методов, свойств, индексов для усовершенствования или изменения их поведения.

Классы так же могут добавлять наблюдателей свойств к наследованным свойствам для того, чтобы быть в курсе, когда происходит смена значения свойства. Наблюдатели свойств могут быть добавлены для любого свойства, несмотря на то были ли они изначально определены как хранимые свойства или вычисляемые.

Определение базового класса

Любой класс, который ничего не наследует из другого класса, называется *базовым классом*.

Заметка

Классы в Swift ничего не наследуют от универсального базового класса. Классы, у которых не указан супер класс (родительский класс), называются базовыми, которые вы можете использовать для строительства других классов.

Пример ниже определяет класс `Vehicle`. Этот базовый класс определяет хранимое свойство `currentSpeed`, с начальным значением `0.0` (выведенный тип `Double`). Значение свойства `currentSpeed` используется вычисляемым нередактируемым свойством `description` типа `String`, для создания описания транспортного средства (экземпляра `Vehicle`). Так же класс `Vehicle` определяет метод `makeNoise`. Этот метод фактически ничего не делает для базового экземпляра класса `Vehicle`, но будет настраиваться суперклассом класса `Vehicle` чуть позже:

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "движется на скорости
\ (currentSpeed) миль в час"
    }
    func makeNoise() {
        //ничего не делаем, так как не каждый
        транспорт шумит
    }
}
```

Вы создаете новый экземпляр класса `Vehicle` при помощи синтаксиса инициализатора, который написан как `TypeName`, за которым идут пустые круглые скобки:

```
let someVehicle = Vehicle()
```

Создав новый экземпляр класса `Vehicle`, вы можете получить доступ к его свойству `description`, для вывода на экран описания текущей скорости транспорта:

```
println("Транспорт: \(someVehicle.description)")
```

```
//Транспорт: движется на скорости 0.0 миль в час
```

Класс `Vehicle` определяет обычные характеристики для обычного транспортного средства, но особо мы их использовать не можем. Чтобы сделать класс более полезными, вам нужно усовершенствовать его для описания более специфичных видов транспорта.

Наследование подклассом

Наследование является актом создания нового класса на базе существующего класса (базового класса). Подкласс наследует характеристики от существующего класса, который затем может быть усовершенствован. Вы также можете добавить новые характеристики подклассу.

Для индикации того, что подкласс имеет суперкласс, просто напишите имя подкласса, затем имя суперкласса и разделите их двоеточием:

```
class SomeSubclass: SomeSuperclass {  
    // определение подкласса проводится тут  
}
```

Приведенный пример определяет подкласс `Bicycle` с суперклассом `Vehicle`:

```
class Bicycle: Vehicle {  
    var hasBasket = false  
}
```

Новый класс `Bicycle` автоматически собирает все характеристики `Vehicle`, например, такие свойства как `currentSpeed` и `description` и метод `makeNoise`.

В дополнение к характеристикам, которые он наследует, класс `Bicycle` определяет свое новое хранимое свойство `hasBasket`, со значением по умолчанию `false` (тип свойства выведен как `Bool`).

По умолчанию, любой новый экземпляр `Bicycle`, который вы создадите не будет иметь корзину (`hasBasket = false`). Вы можете установить `hasBasket` на значение `true` для конкретного экземпляра `Bicycle`, после того как он создан:

```
let bicycle = Bicycle()  
bicycle.hasBasket = true
```

Вы так же можете изменить унаследованное свойство `currentSpeed` экземпляра `Bicycle` и запросить его свойство `description`:

```
bicycle.currentSpeed = 15.0  
println("Велосипед: \"(bicycle.description)\"")  
//Велосипед: движется на скорости 15.0 миль в час
```

Подклассы сами могут создавать подклассы. В следующем примере класс `Bicycle` создает подкласс для двухместного велосипеда известного как “тандем”:

```
class Tandem: Bicycle {  
    var currentNumberOfPassengers = 0  
}
```

Класс `Tandem` наследует все свойства и метода `Bicycle`, который в свою очередь наследует все свойства и методы от `Vehicle`. Подкласс `Tandem` так же добавляет новые хранимые свойство `currentNumberOfPassengers`, которое по умолчанию равно 0.

Если вы создадите экземпляр `Tandem`, то вы можете работать с любой из его новых и наследованных свойств и запрос свойства только для чтения `description` он наследует от `Vehicle`:

```
let tandem = Tandem()  
tandem.hasBasket = true  
tandem.currentNumberOfPassengers = 2  
tandem.currentSpeed = 22.0  
println("Тандем: \${tandem.description}")  
// Тандем: движется на скорости 22.0 миль в  
час
```


Переопределение

Подклассы могут проводить свои собственные реализации методов экземпляра, методов класса, свойств экземпляра, свойств класса или индекса, который в противном случае будет наследовать от суперкласса. Это известно как *переопределение*.

Для переопределения характеристик, которые все равно будут унаследованы, вы приписываете к переписываемому определению ключевое слово `override`. Делая так, вы показываете свое намерение провести переопределение, и что оно будет сделано не по ошибке. Переписывание по случайности может вызвать непредвиденное поведение, и любое переопределение без ключевого слова `override`, будет считаться ошибкой при компиляции кода.

Ключевое слово `override` так же подсказывает компилятору Swift проверить, что вы переопределяете суперкласс класса (или один из его параметров), который содержит то определение, которое вы хотите переопределить. Эта проверка проверяет, что ваше переопределение определения корректно.

Доступ к методам, свойствам, индексам суперкласса

Когда вы проводите переопределение метода, свойства, индекса для подкласса, иногда бывает полезно использовать существующую реализацию суперкласса как часть вашего переопределения. Для примера, вы можете усовершенствовать поведение существующей реализации или сохранить измененное значение в существующей унаследованной переменной.

Там, где это уместно, вы можете получить доступ к методу, свойству, индексу версии суперкласса, если будете использовать префикс `super`:

- Переопределенный метод `someMethod` может вызвать версию суперкласса метода `someMethod`, написав `super.someMethod()` внутри переопределения реализации метода.
- Переопределённое свойство `someProperty` может получить доступ к свойству версии суперкласса `someProperty` как `super.someProperty` внутри переопределения реализации геттера или сеттера.
- Переопределенный индекс для `someIndex` может получить доступ к версии суперкласса того же индекса как `super[someIndex]` изнутри переопределения реализации индекса.

Переопределение методов

Вы можете переопределить унаследованный метод экземпляра или класса, чтобы обеспечить индивидуальную или альтернативную версию реализации метода в подклассе.

Следующий пример определяет новый подкласс `Train` класса `Vehicle`, который переопределяет метод `makeNoise`, который `Train` наследует от `Vehicle`:

```
class Train: Vehicle {
```

```
    override func makeNoise() {  
        println("Чу-чу")  
    }  
}
```

Если вы создаете новый экземпляр класса `Train` и вызовете его метод `makeNoise`, вы увидите, что версия метода подкласса `Train` вызывается вот так:

```
let train = Train()  
train.makeNoise()  
//ВЫВОДИТ "Чу-чу"
```

Переопределение свойств

Вы можете переопределить унаследованные свойства класса или экземпляра для установки вашего собственного геттера и сеттера для этого свойства, или добавить наблюдателя свойства для наблюдения за переопределяемым свойством, когда меняется лежащее в основе значение свойства.

Переопределения геттеров и сеттеров свойства

Вы можете предусмотреть пользовательский геттер (и сеттер, если есть в этом необходимость) для переопределения любого унаследованного свойства, не смотря на то как свойство было определено в самом источнике, как хранимое свойство или как вычисляемое. Подкласс не значит хранимое изначально унаследованное свойство или вычисляемое, все что он знает, так это имя свойства и его тип. Вы всегда должны констатировать и имя и тип свойства, которое вы переопределяете, для того чтобы компилятор мог проверить соответствие и наличие переопределяемого свойства у суперкласса.

Вы можете представить унаследованное свойство только для чтения, как свойство, которое можно читать и редактировать, прописывая и геттер и сеттер в вашем переопределяемом свойстве подкласса. Однако вы не можете сделать наоборот,

то есть редактируемо-читаемое свойство сделать свойством только для чтения.

Заметка

Если вы предоставляете сеттер как часть переопределения свойства, то вы должны предоставить и геттер для этого переопределения. Если вы не хотите изменять значение наследуемого свойства внутри переопределяемого геттера, то вы можете просто передать через наследуемое значение, возвращая `super.someProperty` от геттера, где `someProperty` - имя параметра, который вы переопределяете.

Следующий пример определяет класс `Car`, который является подклассом `Vehicle`. Класс `Car` предоставляет новое хранимое свойство `gear`, имеющее значение по умолчанию равное 1. Класс `Car` также переопределяет свойство `description`, которое он унаследовал от `Vehicle`, для **предоставления** собственного описания, которое включает в себя текущую передачу:

```
class Car: Vehicle {  
  
    var gear = 1  
    override var description: String {  
        return super.description + "на передаче  
    \ (gear) "  
    }  
}
```

Переопределение свойства `description` начинается с `super.description`, который возвращает свойство `description` класса `Vehicle`. Версия класса `Car` свойства `description` добавляет дополнительный текст в конец описания текущего свойства `description`.

Если вы создадите экземпляр класса `Car` и зададите свойства `gear`, `currentSpeed`, то вы увидите что его

свойство `description` возвращает новое описание класса `Car`:

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
println("Машина: \(car.description)")
// Машина: движется на скорости 25.0 миль в
час на 3 передаче
```

Переопределение наблюдателей свойства

Вы можете использовать переопределение свойства для добавления наблюдателей к унаследованному свойству. Это позволяет вам получать уведомления об изменении значения унаследованного свойства, не смотря на то как изначально это свойство было реализовано. Для большей информации о наблюдателях свойств читайте в главе “Наблюдатели свойств”.

Заметка

Вы не можете добавить наблюдателей свойства на унаследованное константное свойство или на унаследованные вычисляемые свойства только для чтения. Значение этих свойств не может меняться, так что нет никакого смысла вписывать `willSet`, `didSet` как часть их реализации.

Так же обратите внимание, что вы не можете обеспечить одно и то же свойство и переопределяемыми наблюдателями свойства, переопределяемым и сеттером свойства. Если вы хотите наблюдать за изменениями значения свойства, и вы готовы предоставить пользовательский сеттер для этого свойства, то вы можете просто наблюдать за изменением какого-либо значения из сеттера.

В следующем примере определим новый класс `AutomaticCar`, который является подклассом `Car`. Класс `AutomaticCar` представляет машину с автоматической коробкой передач, которая автоматически переключает передачи в зависимости от текущей скорости:

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Когда бы вы не поставили свойство `currentSpeed` экземпляра класса `AutomaticCar`, наблюдатель `didSet` свойства устанавливает свойство экземпляра `gear` в подходящее значение передачи в зависимости от скорости. Если быть точным, то наблюдатель свойства выбирает передачу как значение равное `currentSpeed` поделенная на 10 и округленная вниз и выбираем ближайшее целое число + 1. Если скорость равно 10.0, то передача равна 1, если скорость 35.0, то передача 4:

```
let automatic = AutomaticCar ()
automatic.currentSpeed = 35.0
println("Машина с автоматом: "
    \ (automatic.description) ")
//выводит: Машина с автоматом движется на
скорости 35.0 миль в час на 4 передаче
```

Предотвращение переопределений

Вы можете предотвратить переопределение метода, свойства или индекса, обозначит его как *конечный*. Сделать это можно написав ключевое слово `final` перед ключевым словом метода, свойства или индекса (`final var`, `final func`, `final class func`, и `final subscript`).

Любая попытка переписать конечный метод, свойство или индекс в подклассе приведет к ошибке компиляции. Методы, свойства и индексы, которые вы добавляете в класс в расширении так же могут быть отмечены как конечные внутри определения расширения.

Вы можете отметить целый класс как конечный или финальный, написав слово `final` перед ключевым словом `class` (`final class`). Любая попытка изменить класс так же приведет к ошибке компиляции.

Инициализация

Инициализация - подготовительный процесс экземпляра класса, структуры или перечисления для дальнейшего использования. Этот процесс включает в себя установку начальных значений для каждого хранимого свойства этого экземпляра и проведение любых настроек или инициализации, которые нужны до того, как экземпляр будет использоваться.

Вы реализуете эту инициализацию, определяя *инициализаторы*, которые схожи со специальными методами, которые вызываются для создания экземпляра определенного типа. В отличие от инициализаторов в Objective-C, инициализаторы в Swift не возвращают значения. Основная роль инициализаторов - убедиться в том, что новый экземпляр типа правильно инициализирован до того, как будет использован в первый раз.

Экземпляры классовых типов так же могут реализовывать *деинициализаторы*, которые проводят любую чистку прямо перед тем, как экземпляр класса будет освобожден. Для более подробной информации читайте “Деинициализация”.

Установка начальных значений для хранимых свойств

Классы и структуры должны устанавливать начальные значения у всех хранимых свойств во время создания класса или структуры. Хранимые свойства не могут быть оставлены в неопределённом состоянии.

Вы можете установить начальное значение свойства внутри инициализатора или присвоить ему значение по умолчанию, как часть определения свойства. Эти моменты будут описаны подробнее в следующих секциях.

Заметка

Когда вы присваиваете значение по умолчанию хранимому свойству или устанавливаете исходное значение в инициализаторе, то значение устанавливается напрямую, без вызова наблюдателей.

Инициализаторы

Инициализаторы вызываются для создания нового экземпляра конкретного типа. В самой простой своей форме инициализатор работает как метод экземпляра без параметров, написанный с помощью ключевого слова `init`:

```
init() {  
    // инициализация проводится тут  
}
```

Пример ниже определяет новую структуру `Fahrenheit` для хранения температур, представленных в Фаренгейтах. Структура `Fahrenheit` имеет одно свойство, `temperature` типа `Double`:

```
struct Fahrenheit {  
    var temperature: Double  
    init() {  
        temperature = 32.0  
    }  
}  
var f = Fahrenheit()  
println("Значение температуры по умолчанию  
\ (f.temperature) по Фаренгейту")  
// Выведет "Значение температуры по умолчанию  
32.0 по Фаренгейту"
```

Структура определяет один инициализатор, `init`, без параметров, который инициализирует хранимую температуру равную 32.0 (температура замерзания воды по Фаренгейту).

Значения свойств по умолчанию

Вы можете установить исходное значение свойства в инициализаторе, как показано выше. Альтернативно вы можете указать значение свойства по умолчанию, как часть определения свойства. Вы указываете значение свойства по умолчанию, написав исходное значение свойства, когда оно определено.

Заметка

Если свойство каждый раз берет одно и то же исходное значение, то лучше указать это значение, в качестве значения по умолчанию, чем каждый раз устанавливать его в инициализаторе. Конечный результат такой же, но значение

по умолчанию связывает инициализацию свойства ближе к своему объявлению. Так делают, чтобы оставить инициализаторы в более чистой и краткой форме, и это позволяет вам вывести тип свойства из его значения по умолчанию. Значения по умолчанию так же дают вам больше преимуществ для использования инициализаторов по умолчанию или наследования инициализатора, что описано подробнее далее в этой главе.

Вы можете написать структуру `Fahrenheit` в более простой форме, указав значение по умолчанию для свойства `temperature`, в месте его объявления:

```
struct Fahrenheit {  
    var temperature = 32.0  
}
```

Настройка инициализации

Вы можете настроить процесс инициализации входными параметрами и опциональным типом свойства или изменением константных свойств во время инициализации, что будет описано далее.

Параметры инициализации

Вы можете показать параметры инициализации как часть определения инициализатора, для определения типов и имен значений, которые настраивают процесс инициализации. Параметры инициализации имеют те же возможности и синтаксис как и параметры функции или метода.

Следующий пример определяет структуру `Celsius`, которая хранит температуру в Цельсиях. Структура `Celsius` реализует два пользовательских

инициализатора `init(fromFahrenheit:)` и `init(fromKelvin:)`, которые инициализируют новый экземпляр структуры со значением другой температурной шкалы:

```
struct Celsius {  
    var temperatureInCelsius: Double  
    init(fromFahrenheit fahrenheit: Double) {  
        temperatureInCelsius = (fahrenheit - 32.0)  
/ 1.8  
    }  
    init(fromKelvin kelvin: Double) {  
        temperatureInCelsius = kelvin - 273.15  
    }  
}  
let boilingPointOfWater =  
Celsius(fromFahrenheit: 212.0)  
//boilingPointOfWater.temperatureInCelsius  
равно 100.0  
let freezingPointOfWater = Celsius(fromKelvin:  
273.15)  
//freezingPointOfWater.temperatureInCelsius  
равно 0.0
```

Первый инициализатор имеет один параметр с внешним именем `fromFahrenheit` и с локальным именем `fahrenheit`. Второй инициализатор имеет один параметр с внешним именем `fromKelvin` и локальным именем `kelvin`. Оба инициализатора конвертируют их единственный аргумент в значение по Цельсию и сохраняют это значение в свойство `temperatureInCelsius`.

Локальные и внешние имена параметров

Как и в случае с параметрами функций или методов, параметры инициализации могут иметь локальные имена для использования внутри тела инициализатора и внешние для использования при вызове инициализатора.

Однако инициализаторы не имеют своего имени до круглых скобок, как это имеют методы или функции. Поэтому имена и типы параметров инициализатора играют важную роль в определении того, какой инициализатор и где может быть использован. Из-за этого Swift предоставляет автоматические внешние имена для каждого параметра, если вы, конечно, не укажете свое внешнее имя. Внешнее имя, которое дается по умолчанию, такое же, что и локальное, как если бы вы написали решеточку перед инициализацией параметра.

Следующий пример определяет структуру `Color` с тремя постоянными свойствами `red`, `green`, `blue`. Эти свойства имеют значения от 0.0 до 1.0, для индикации количества соответствующего цвета.

`Color` имеет инициализатор с тремя параметрами `red`, `green`, `blue` типа `Double` (компоненты цвета красного, зеленого, синего). Так же `Color` имеет второй инициализатор с одним параметром `white`, который нужен для предоставления значения для всех трех компонентов цвета.

```

struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue:
Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
    init(white: Double) {
        red = white
        green = white
        blue = white
    }
}

```

Оба инициализатора могут быть использованы для создания нового экземпляра `Color`, передав значения в каждый параметр инициализатора:

```

let magenta = Color(red: 1.0, green: 0.0,
blue: 1.0)
let halfGray = Color(white: 0.5)

```

Обратите внимание, что невозможно вызвать инициализатор без использования внешних имен. Внешние имена обязательно должны быть использованы в инициализаторе, если они определены, если пропустить их, то выскочит ошибка компиляции:

```

let veryGreen = Color(0.0, 1.0, 0.0)
//Этот код вызовет ошибку компиляции, так как
здесь нет внешних имен

```

Параметры инициализатора без внешних имен

Если вы не хотите использовать внешние имена для параметров инициализации, напишите подчеркивание () вместо явного указания внешнего имени для этого параметра, чтобы переопределить поведение по умолчанию.

Вот расширенный вариант для `Celsius`, который мы рассматривали ранее, с дополнительным инициализатором, для создания нового экземпляра `Celsius` с типом значения температуры `Double` только уже по шкале Цельсия:

```
struct Celsius {  
    var temperatureInCelsius: Double  
    init(fromFahrenheit fahrenheit: Double) {  
        temperatureInCelsius = (fahrenheit - 32.0)  
    / 1.8  
    }  
    init(fromKelvin kelvin: Double) {  
        temperatureInCelsius = kelvin - 273.15  
    }  
    init(_ celsius: Double) {  
        temperatureInCelsius = celsius  
    }  
}  
  
let bodyTemperature = Celsius(37.0)  
//bodyTemperature.temperatureInCelsius равна  
37.0
```

Инициализатор вызывает `Celsius(37.0)`, что понятно и без внешнего имени параметра. Поэтому целесообразно написать `init(_ celsius: Double)`, для того, чтобы предоставить безымянное значение типа `Double`.

Опциональные типы свойств

Если ваш пользовательский тип имеет свойство, которое логически имеет “отсутствие значения”, возможно потому, что его значение не может быть установлено во время инициализации или потому, что ему разрешается иметь “отсутствие значения” в какой-либо точке кода, то такое свойство нужно объявить с *опциональным* типом. Свойства опционального типа автоматически инициализируются со значением `nil`, указывая на то, что значение стремится иметь значение “пока что отсутствие значения” на этапе инициализации.

Следующий код определяет класс `SurveyQuestion` с опциональным типом `String` свойства `response`:

```
class SurveyQuestion {
  var text: String
  var response: String?
  init(text: String) {
    self.text = text
  }
  func ask() {
    println(text)
  }
}
let cheeseQuestion = SurveyQuestion(text:
"Нравится ли вам сыр?")
cheeseQuestion.ask()
//выводит "Нравится ли вам сыр?"
cheeseQuestion.response = "Да, я люблю сыр"
```


Ответ на вопрос не может быть известен до того, как мы получили на него ответ, так что свойство `response` должно быть типа `String?` (опциональный `String`). Ему автоматически присваивается значение `nil` при инициализации `SurveyQuestion`, значащее, что “значения пока нет”.

Изменение константных свойств во время инициализации

Вы можете изменять значения постоянных(когда свойство константа) свойств в любой точке вашего процесса инициализации.

Заметка

В экземплярах класса постоянное свойство может быть изменено только во время инициализации класса, в котором оно представлено. Оно не может быть изменено подклассом.

Вы можете пересмотреть пример `SurveyQuestion` и использовать вместо переменного свойства `text` постоянное свойство `text`, для индикации того, что это свойство не меняется после создания `SurveyQuestion`. Даже если свойство является постоянным, оно все еще может быть установлено в инициализаторе класса:

```
class SurveyQuestion {  
  
    let text: String  
    var response: String?  
    init(text: String) {  
        self.text = text  
    }  
    func ask() {  
        println(text)  
    }  
}  
  
let beetsQuestion = SurveyQuestion(text: "Что  
насчет свеклы?")  
beetsQuestion.ask()  
//выводит "Что насчет свеклы?"  
beetsQuestion.response = "Я люблю свеклу, но  
не в сыром виде!"
```

Дефолтные

инициализаторы

Swift предоставляет *дефолтный инициализатор* для любой структуры или базового класса, который имеет значение по умолчанию для всех его свойств и не имеет ни одного инициализатора. Дефолтный инициализатор просто создает новый экземпляр со всеми его свойствами с уже присвоенными значениями по умолчанию.

Этот пример определяет класс `ShoppingListItem`, который включает в себя имя, количество и состояние сделки на предмет из листа покупок:

```
class ShoppingListItem {  
    var name: String?  
    var quantity = 1  
    var purchased = false  
}  
var item = ShoppingListItem()
```

Так как все свойства класса `ShoppingListItem` имеют значения по умолчанию и так как этот класс не имеет суперкласса, то `ShoppingListItem` автоматически получает реализацию дефолтного инициализатора, которая создает новый экземпляр со всеми свойствами с уже присвоенными значениями по умолчанию. (Свойство `name` - свойство опционального типа `String`, значит значение по умолчанию равно `nil`). В примере выше используется дефолтный инициализатор для класса `ShoppingListItem` для создания нового экземпляра. Синтаксис дефолтного инициализатора в нашем случае выглядит как `ShoppingListItem()`, что присваивается переменной `item`.

Почленные инициализаторы структурных типов

Структурные типы автоматически получают *почленный инициализатор*, если они не определяют своего пользовательского инициализатора. Это верно даже при условии, что хранимые свойства не имеют значений по умолчанию.

Почленный инициализатор - сокращенный способ инициализировать свойства члена нового экземпляра структуры. Начальные значения для свойств нового экземпляра могут быть переданы в почленный инициализатор по имени.

Пример ниже определяет структуру `Size` с двумя свойствами `width`, `height`. Оба свойства выведены как `Double` из-за начального значения равного `0.0`.

Структура `Size` автоматически получает `init(width:height:)` почленный инициализатор, который вы можете использовать для инициализации `Size` экземпляра:

```
struct Size {  
  
    var width = 0.0, height = 0.0  
}  
let twoByTwo = Size(width: 2.0, height: 2.0)
```

Делегирование инициализатора для типов значений

Инициализаторы могут вызывать другие инициализаторы для инициализации части экземпляра. Этот процесс называется как *делегирование инициализатора*. Он позволяет избегать дублирования кода в разных инициализаторах.

Правила того, как работает это делегирование инициализатора и для каких форм делегирования это возможно для типов значений и ссылочных типов разные. Типы значений (структуры и перечисления) не поддерживают наследование, так что их процесс делегирования инициализатора сравнительно прост, потому что они только могут делегировать другому инициализатору то, что предоставляют сами. Классы, однако, могут наследовать от других классов, как это описано в [“Наследование”](#). Это значит, что у классов есть дополнительная ответственность за проверку наличия корректных значений у каждого хранимого унаследованного свойства класса. Обязанности описаны в главе [“Наследование и инициализация класса”](#).

Для типов значений вы используете `self.init` для ссылки на остальные инициализаторы одного и того же типа значения, когда вы пишете свои инициализаторы. Вы можете вызывать `self.init` из инициализатора.

Обратите внимание, что если вы определите пользовательский инициализатор для типов значений, то вы больше не будете иметь доступа к дефолтному инициализатору (или почленному инициализатору, если это структура) для этого типа. Такое ограничение предотвращает

ситуацию, в которой настройка важного дополнения в более сложном инициализаторе может быть пропущена при случайном использовании автоматического инициализатора.

Заметка

Если вы хотите, чтобы ваш пользовательский тип значения имел возможность быть инициализированным или дефолтным инициализатором, или почленным инициализатором, или вашим пользовательским инициализатором, то вам нужно написать свой пользовательский инициализатор как расширение, чем как часть реализации типа значения.

Следующий пример определяет пользовательскую структуру `Rect` для отображения геометрического прямоугольника. Примеру нужно добавить две вспомогательные структуры `Size`, `Point`, каждая из которых предоставляет значения по умолчанию `0.0` для своих свойств:

```
struct Size {  
  
    var width = 0.0, height = 0.0  
}  
struct Point {  
    var x = 0.0, y = 0.0;  
}
```

Вы можете инициализировать структуру `Rect` тремя способами: используя свою нулевую инициализацию значений свойств `origin` и `size`, предоставляя определенную точку и размер, или предоставляя точку центра и размер. Эти опции инициализации представлены тремя пользовательскими инициализаторами, которые являются частью определения структуры `Rect`:

```

struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y:
originY), size: size)
    }
}

```

Первый инициализатор `Rect` – `init()` функционально то же самое, что и дефолтный инициализатор, который бы получила структура, если бы не имела пользовательских инициализаторов. Инициализатор имеет пустое тело, отображенное парой пустых фигурных скобок {}, и не проводит никакой инициализации. Вызывая такой инициализатор, мы возвращаем экземпляр `Rect`, который имеет инициализированные свойства `origin`, `size` значениями `Point` (`x: 0.0`, `y: 0.0`) и `Size` (`width: 0.0`, `height: 0.0`), которые известны из определения свойств:

```

let basicRect = Rect()
//исходная точка Rect (0.0, 0.0) и его размер
(0.0, 0.0)

```

Второй инициализатор `Rect` – `init(origin:size:)` функционально то же самое что и почтенный инициализатор, который могла бы иметь структура, если бы не имела пользовательских инициализаторов. Этот инициализатор просто присваивает

значения аргументов `origin`, `size` соответствующим свойствам:

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0), size: Size(width: 5.0, height: 5.0))  
//исходная точка Rect (2.0, 2.0) и его размер (5.0, 5.0)
```

Третий инициализатор `Rect` - `init(center:size:)` немного более сложный. Он начинается с вычисления соответствующей исходной точки, основываясь на точке `center` и значении `size`. Только потом он вызывает (или делегирует) `init(origin:size:)` инициализатор, который хранит новую исходную точку и значения размеров соответствующих свойств:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0))  
//исходная точка centerRect'a равна (2.5, 2.5) и его размер (3.0, 3.0)
```

Инициализатор `init(center:size:)` мог бы присвоить новые значения `origin`, `size` соответствующим свойствам самостоятельно. Однако более удобно (т.к. более понятный из-за краткости) для инициализатора `init(center:size:)` воспользоваться преимуществом того, что существует другой инициализатор с абсолютно такой же функциональностью.

Заметка

Для альтернативного способа записи этого примера без инициализаторов `init()`, `init(origin:size:)` смотрите главу [“Расширения”](#).

Наследование и инициализация классов

Всем свойствам класса, включая и те, что унаследованы у суперкласса должны быть присвоены начальные значения, во время их инициализации.

Swift определяет два вида инициализаторов классовых типов для проверки того, что все свойства получили какие-либо значения. Они известны как назначенные инициализаторы (конструкторы) или вспомогательные инициализаторы.

Назначенный и вспомогательный инициализатор

Назначенные инициализаторы в основном инициализаторы класса. Они предназначены для того, чтобы полностью инициализировать все свойства представленные классом и чтобы вызвать соответствующий инициализатор суперкласса для продолжения процесса инициализации цепочки наследований суперклассов.

Так сложилось, что классы чаще всего имеют очень мало назначенных инициализаторов, чаще всего бывает, что класс имеет всего один инициализатор. Назначенные инициализаторы объединяют в себе все точки, через которые проходит процесс инициализации и через которые процесс инициализации идет по цепочке в суперкласс.

Каждый класс должен иметь хотя бы один назначенный инициализатор. В некоторых случаях, это требование удовлетворяется наследованием одного или более назначенных инициализаторов от суперкласса.

Вспомогательные инициализаторы являются вторичными, поддерживающими инициализаторами для класса. Вы можете определить вспомогательный инициализатор для вызова назначенного инициализатора из того же класса, что и вспомогательный инициализатор с некоторыми параметрами назначенного инициализатора с установленными начальными значениями.

Вы не обязаны обеспечивать вспомогательные инициализаторы, если ваш класс не нуждается в них. Создавайте вспомогательный инициализатор всякий раз, когда это является наиболее рациональным путем образца общей инициализации и может сэкономить время и сделать саму инициализацию класса более чистой и краткой.

Синтаксис назначенных и вспомогательных инициализаторов

Назначенные инициализаторы для классов записываются точно так же как и простые инициализаторы для типов значений:

```
init (параметры) {  
    выражения  
}
```

Вспомогательные инициализаторы пишутся точно так же, но только дополнительно используется вспомогательное слово `convenience`, которое располагается до слова `init` и разделяется пробелом:

```
convenience init (параметры) {  
    выражения  
}
```

Делегирование инициализатора для классовых типов

Для простоты отношений между назначенными и вспомогательными инициализаторами, Swift использует следующие три правила для делегирования вызовов между инициализаторами:

Правило 1

Назначенный инициализатор должен вызывать назначенный инициализатор из суперкласса.

Правило 2

Вспомогательный инициализатор должен вызывать другой инициализатор из того же класса.

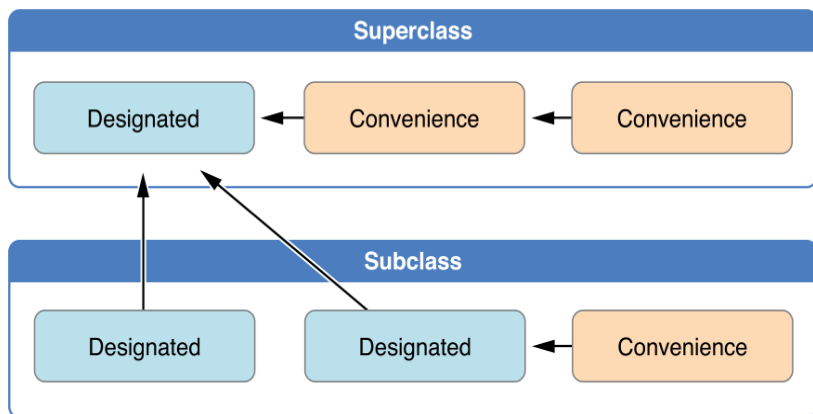
Правило 3

Вспомогательный инициализатор в конечном счете должен вызывать назначенный инициализатор.

Вот как можно просто это запомнить:

- Назначенные инициализаторы должны делегировать наверх
- Вспомогательные инициализаторы должны делегировать по своему уровню (классу).

Вот как это правило выглядит в иллюстрированной форме:



Здесь, в суперклассе есть один назначенный инициализатор и два вспомогательных инициализатора. Один вспомогательный инициализатор вызывает другой вспомогательный инициализатор, который в свою очередь вызывает единственный назначенный инициализатор. Этот рисунок удовлетворяет правилам 2 и 3. Этот суперкласс сам по себе уже дальше не имеет суперкласса, так что первое правило здесь не применимо.

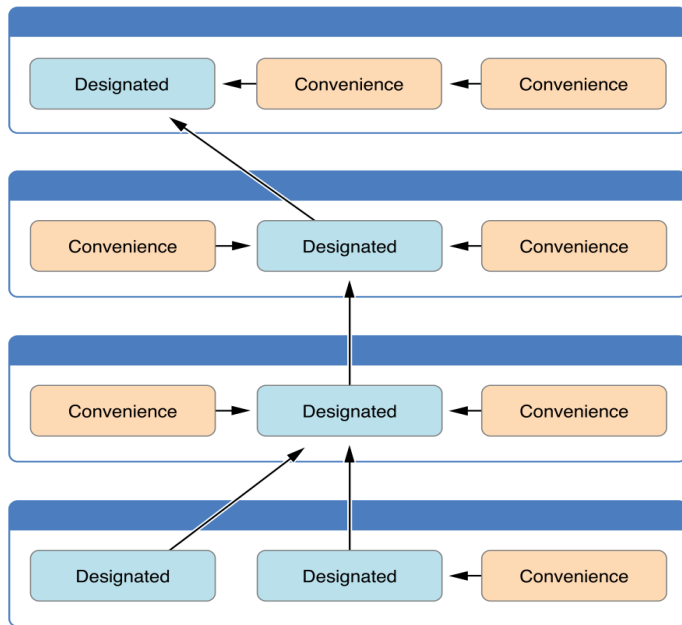
Подкласс на этом рисунке содержит два назначенных инициализатора и один вспомогательный инициализатор. Вспомогательный инициализатор должен вызвать одного из двух назначенных инициализатора, потому что он может вызвать другой инициализатор того же класса. Это правило так же соответствует правилам 2 и 3. Оба назначенных инициализатора должны вызвать один назначенный инициализатор из суперкласса, для того чтобы соответствовать правилу 1.

Заметка

Эти правила никак не относятся к тому, как пользователи ваших классов создают экземпляры каждого класса. Любой инициализатор из схемы выше может быть использован для

создания полностью инициализированного экземпляра класса, которому он принадлежит. Правила влияют лишь на то, как вы будете писать реализацию класса.

Схема ниже показывает более сложную иерархию из четырех классов. Она показывает как назначенные инициализаторы работают в качестве точек прохождения инициализации класса, упрощая внутренние взаимоотношения среди цепочки классов:



Двухфазная инициализация

Инициализация класса в Swift является двухфазным процессом. На первой фазе каждое свойство получает начальное значение от класса, в котором оно представлено. Как только первоначальные значения для хранимых свойств

были определены, начинается вторая фаза, и каждому классу предоставляется возможность изменить свои свойства еще до того как будет считаться, что созданный экземпляр можно использовать.

Использование двухфазного процесса инициализации делает инициализацию безопасной, в то же время обеспечивая полную гибкость классов в классовой иерархии. Двухфазная инициализация предотвращает значения свойств от доступа к ним еще до того, как они будут инициализированы и предотвращает их от случайного значения выставленного из другого инициализатора.

Заметка

Двухфазный процесс инициализации в Swift аналогичен инициализации в Objective-C. Основное отличие между ними проходит на первой фазе в том, что в Objective-C свойства получают значения 0 или `nil`. В Swift же этот процесс более гибкий и позволяет устанавливать пользовательские начальные значения и может обработать типы, для которых значения 0 или `nil`, являются некорректными.

Компилятор Swift проводит четыре полезные проверки безопасности для подтверждения того, что ваша двухфазная инициализация прошла без ошибок:

Проверка 1. Назначенный инициализатор должен убедиться в том, что все свойства представленные его классом инициализированы до того, как он делегирует наверх, в инициализатор суперкласса.

Как было сказано выше, память для объекта считается полностью инициализированной только для полностью инициализированного объекта, где все значения хранимых свойств известны. Для того чтобы удовлетворить этому правилу, назначенный инициализатор должен убедиться, что

всего его собственные свойства инициализированы до того, как будут переданы вверх по цепочке.

Проверка 2. Назначенный инициализатор должен делегировать суперклассу инициализатор до присваивания значений унаследованным свойствам. Если этого сделано не будет, то новое значение, которое присвоит назначенный инициализатор будет переписано суперклассом, как часть инициализации суперкласса.

Проверка 3. Вспомогательный инициализатор должен делегировать другому инициализатору до того, как будут присвоены значения любым свойствам (включая свойства определенные тем же классом). Если этого сделано не будет, то новое значение, которое присваивает вспомогательный инициализатор, будет перезаписано его собственным назначенным инициализатором класса.

Проверка 4. Инициализатор не может вызывать методы экземпляра, читать значения любого свойства экземпляра или ссылаться на `self` как на значение до тех пор, пока не будет закончена первая фаза инициализации.

Экземпляр класса является не совсем корректным до тех пор, пока не закончится первая фаза. К свойствам можно получить доступ и можно вызывать методы только тогда, как стало известно, что экземпляр валиден (корректен) к концу первой фазы.

Вот как проходит двухфазная инициализация, основанная на четырех проверках(описанные выше):

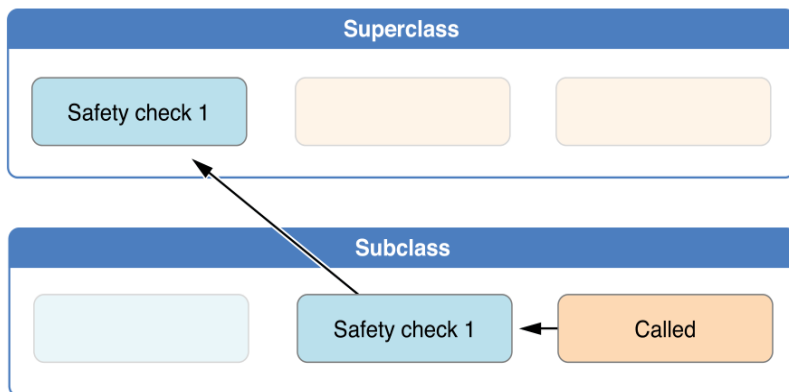
Фаза первая

- Назначенный или вспомогательный инициализатор вызывается в классе.
- Память под новый экземпляр этого класса выделяется. Но она еще не инициализирована.
- Назначенный инициализатор для этого класса подтверждает, что все свойства, представленные этим классом, имеют значения. Память под эти свойства теперь инициализирована.
- Назначенный инициализатор передает инициализатору суперкласса, что пора проводить те же действия, только для его собственных свойств.
- Так продолжается по цепочке до самого верхнего суперкласса.
- После того как верхушка этой цепочки достигнута и последний класс в цепочке убедился в том, что все его свойства имеют значение, только тогда считается, что память для этого экземпляра полностью инициализирована. На этом первая фаза кончается.

Фаза вторая

- Двигаясь вниз по цепочке, каждый назначенный инициализатор в этой цепочке имеет такую возможность, как настраивать экземпляр. Теперь инициализаторы получают доступ к **self** и могут изменять свои свойства, создавать экземпляры и вызывать методы и т.д.
- И наконец, каждый вспомогательный инициализатор в цепочки имеет возможность настраивать экземпляр и работать с **self**.

Вот как выглядит первая фаза для гипотетического подкласса и суперкласса:



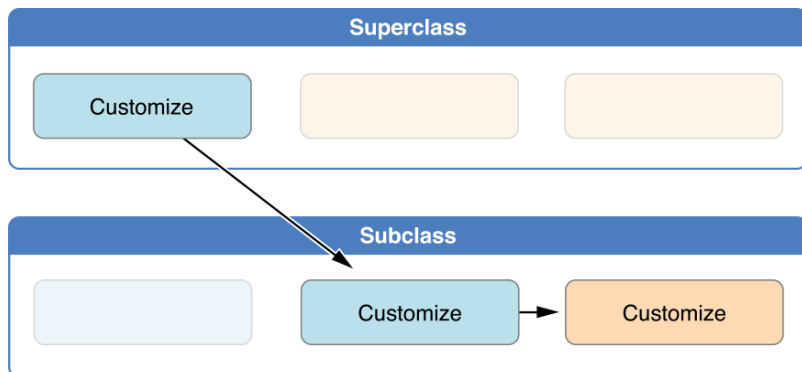
В этом примере инициализация начинается с вызова вспомогательного инициализатора подкласса. Вспомогательный инициализатор пока не может изменять каких-либо свойств. Он делегирует назначенному инициализатору по тому же классу, где и он сам.

Назначенный инициализатор убеждается, что все свойства подкласса имеют значения, как указано в проверке 1. После этого он вызывает назначенный инициализатор своего суперкласса для продолжения инициализации.

Назначенный инициализатор суперкласса проверяет наличие значений у всех свойств суперкласса. Далее нет продолжения цепочки наверх, к другому суперклассу, так что нет дальнейшей нужды в делегировании.

Сразу после того как все свойства суперкласса получают начальные значения, память считается полностью инициализированной, Фаза 1 завершается.

Вот как выглядит Фаза 2:



Назначенный инициализатор суперкласса только теперь получает возможность менять дальнейшие экземпляры(хотя и не обязан).

Как только назначенный инициализатор суперкласса заканчивает работу, получает возможность вносить изменения назначенный инициализатор подкласса(хотя он так же и не обязан это делать).

Наконец, как только заканчивает работу назначенный инициализатор подкласса, то возможность вносить изменения получает вспомогательный инициализатор, который изначально был вызван.

То есть если рассмотреть весь процесс отдаленно, то он получается, как бы, идет вверх, а потом спускается вниз.

Наследование и переопределение инициализатора

В отличие от подклассов в Objective-C, подклассы в Swift не наследуют инициализаторов их суперклассов по умолчанию.

Такой подход в Swift предотвращает ситуации, когда простой инициализатор суперкласса наследуется более специфичным подклассом, а потом используется для создания экземпляра подкласса, который не полностью или не правильно инициализирован.

Заметка

Инициализаторы суперкласса наследуются в определенных обстоятельствах, но только когда это безопасно и когда это имеет смысл делать. Далее мы это разберем.

Если вы хотите, чтобы у вашего подкласса были один или более инициализаторов его суперклассов, вы можете сделать свою реализацию этих инициализаторов внутри подкласса.

Когда вы пишете инициализатор подкласса, который совпадает с назначенным инициализатором суперкласса, вы фактически переопределяете назначенный инициализатор. Таким образом вы должны писать модификатор `override` перед определением инициализатора подкласса. Это верно даже если вы переопределяете автоматически предоставляемый инициализатор, как описано в [“Дефолтные инициализаторы”](#).

Так же как и переопределенные свойства, методы или индексы, присутствие модификатора `override` подсказывает Swift проверить то, что суперкласс имеет совпадающий назначенный инициализатор, который должен быть переписан, и проверить параметры вашего переопределяющего инициализатора, чтобы они были определены так как и предполагалось.

Заметка

Вы всегда можете написать модификатор `override`, когда переписываете назначенный инициализатор суперкласса, даже если ваша реализация вспомогательного

инициализатора подкласса и есть вспомогательный инициализатор.

И наоборот, если вы пишете инициализатор подкласса, который совпадает с вспомогательным инициализатором суперкласса, то этот вспомогательный инициализатор суперкласса никогда не сможет быть вызван напрямую вашим подклассом, в соответствии с правилами указанными выше. Таким образом ваш подкласс не проводит переопределение инициализатора суперкласса. И в результате, вы не пишете модификатор `override`, когда проводите совпадающую реализацию вспомогательного инициализатора суперкласса.

Пример ниже определяет базовый класс `Vehicle`. Это базовый класс объявляет свойства `numberOfWheels` со значением `0` типа `Int`. Свойство `numberOfWheels` используется для вычисляемого свойства `description`, для создания описания характеристик транспортного средства типа `String`:

```
class Vehicle {  
    var numberOfWheels = 0  
    var description: String {  
        return "\ (numberOfWheels) колес (a) "  
    }  
}
```

Класс `Vehicle` предоставляет значение по умолчанию для его единственного свойства, и не имеет никаких собственных пользовательских инициализаторов. И в результате он автоматически получает дефолтный инициализатор, как описано в главе “[Дефолтные инициализаторы](#)”. Дефолтный инициализатор (когда доступен) всегда является назначенным инициализатором для класса и может быть использован для создания нового экземпляра класса `Vehicle` с `numberOfWheels` равным `0`:

```
let vehicle = Vehicle()
println("Транспортное средство
\ (vehicle.description) ")
//Транспортное средство 0 колес(o)
```

Следующий пример определяет подкласс **Bicycle** суперкласса **Vehicle**:

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}
```

Подкласс **Bicycle** определяет пользовательский назначенный инициализатор **init()**. Назначенный инициализатор совпадает с назначенным инициализатором из суперкласса **Bicycle** и, таким образом, версия этого инициализатора класса **Bicycle** отмечена модификатором **override**.

Инициализатор **init()** для **Bicycle** начинается с вызова **super.init()**, который в свою очередь вызывает дефолтный инициализатор для суперкласса **Vehicle** класса **Bicycle**. Он проверяет, что унаследованное свойство **numberOfWheels** инициализировано в **Vehicle**, после чего у **Bicycle** появляется возможность его модифицировать. После вызова **super.init()** начальное значение **numberOfWheels** заменяется значением 2.

Если вы создаете экземпляр **Bicycle**, вы можете вызвать его унаследованное вычисляемое свойство **description**, для того, чтобы посмотреть как обновилось свойство **numberOfWheels**:

```
let bicycle = Bicycle()
println("Велосипед: \ (bicycle.description) ")
//Велосипед: 2 колес(a)
```

Заметка

Подклассы могут менять унаследованные переменные свойства в процессе инициализации, но нельзя менять константные унаследованные свойства.

Наследование автоматического инициализатора

Как было сказано ранее, подклассы не наследуют инициализаторы суперкласса по умолчанию. Однако инициализаторы суперкласса автоматически наследуются, если есть для того специальные условия. На практике это значит, что во многих случаях вам не нужно писать переопределения инициализатора, так как он может наследовать инициализаторы суперкласса с минимальными усилиями, но только когда это безопасно.

Допуская, что вы предоставляете значения по умолчанию любому новому свойству, представленному в подклассе, то применяются два правила:

Правило 1. Если ваш подкласс не определяет ни одного назначенного инициализатора, он автоматически наследует все назначенные инициализаторы суперкласса.

Правило 2. Если у вашего класса есть реализация всех назначенных инициализаторов его суперкласса, то ли они были унаследованы как по правилу 1 или же предоставлены как часть пользовательской реализации определения подкласса, но тогда этот подкласс автоматически наследует все вспомогательные инициализаторы суперкласса.

Эти правила применимы даже если ваш подкласс позже добавляет вспомогательные инициализаторы.

Заметка

Подкласс может реализовать назначенный инициализатор

суперкласса как вспомогательный инициализатор подкласса в качестве части удовлетворяющей правилу 2.

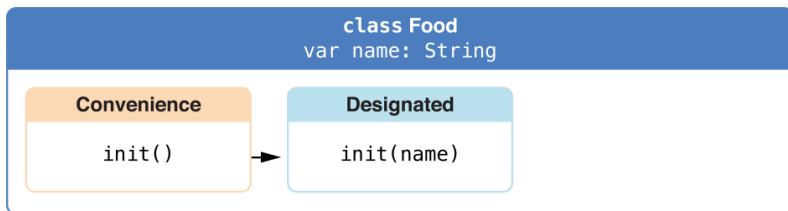
Назначенные и вспомогательные инициализаторы в действии

Следующий пример показывает назначенные и вспомогательные инициализаторы, и автоматическое наследование инициализатора в действии. Этот пример определяет иерархию трех классов `Food`, `RecipeIngredient` и `ShoppingListItem` и демонстрирует как их инициализаторы взаимодействуют.

Основной (базовый) класс называется `Food`, который имеет одно простое свойство типа `String` называемое `name` и обеспечивает два инициализатора для создания экземпляров класса `Food`:

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

Схема ниже показывает цепочку работы инициализаторов в классе `Food`:



Классы по умолчанию не имеют почленного инициализатора, так что класс **Food** предоставляет назначенный инициализатор, который принимает единственный аргумент **name**. Этот инициализатор может быть использован для создания экземпляра **Food** со специфичным именем:

```
let namedMeat = Food(name: "Бекон")
//имя namedMeat является "Бекон"
```

Инициализатор **init(name: String)** из класса **Food**, представлен в виде назначенного инициализатора, потому что он проверяет, что все хранимые свойства нового экземпляра **Food** **полностью** инициализированы. Класс **Food** не имеет суперкласса, так что инициализатор **init(name: String)** не имеет вызова **super.init()** для завершения своей инициализации.

Класс **Food** так же обеспечивает вспомогательный инициализатор **init()** без аргументов.

Инициализатор **init()** предоставляет имя плейсхолдера для новой еды, делегируя к параметру **name** инициализатора **init(name: String)**, давая ему значение **[Unnamed]**:

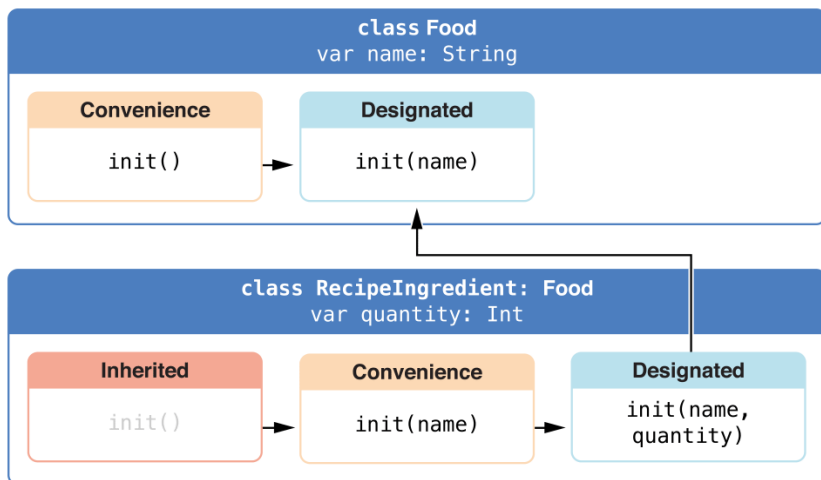
```
let mysteryMeat = Food()
//mysteryMeat называется "[Unnamed]"
```

Второй класс в иерархии - это подкласс **RecipeIngredient** класса **Food**. Класс **RecipeIngredient** создает модель ингредиентов в рецепте. Он

представляет свойство `quantity` типа `Int` (в дополнение к свойству `name`, унаследованное от `Food`) и определяет два инициализатора для создания экземпляров `RecipeIngredient` :

```
class RecipeIngredient: Food {  
    var quantity: Int  
    init(name: String, quantity: Int) {  
        self.quantity = quantity  
        super.init(name: name)  
    }  
    override convenience init(name: String) {  
        self.init(name: name, quantity: 1)  
    }  
}
```

Схема ниже показывает цепочку инициализаторов для класса `RecipeIngredient`:



Класс **RecipeIngredient** имеет один назначенный инициализатор **init(name: String, quantity: Int)**, который может быть использован для распространения всех свойств нового экземпляра **RecipeIngredient**. Этот инициализатор начинается с присваивания переданного аргумента **quantity** свойству **quantity**, которое является единственным новым свойством представленным в **RecipeIngredient**. После того как это сделано, инициализатор делегирует вверх инициализатор **init(name: String)** для класса **Food**. Этот процесс удовлетворяет проверке №1 из раздела “Двухфазная инициализация”, что находится выше на этой же странице.

RecipeIngredient так же определяет вспомогательный инициализатор **init(name: String)**, который используется создания экземпляра **RecipeIngredient** только по имени. Этот вспомогательный инициализатор присваивает значение количеству равное 1 для любого экземпляра, которое создано без явного указания количества. Определение этого вспомогательного инициализатора ускоряет создание экземпляров класса **RecipeIngredient** и позволяет избежать

повторения кода при создании экземпляра, где свойство `quantity` изначально всегда равно 1. Этот вспомогательный инициализатор делегирует по назначенному инициализатору класса, передавая ему `quantity` равное 1.

Вспомогательный инициализатор `init(name: String)` предоставленный `RecipeIngredient` принимает те же параметры, что и назначенный инициализатор `init(name: String)` в `Food`. Из-за того, что вспомогательный инициализатор переопределяет назначенный инициализатор из своего суперкласса, то он должен быть обозначен ключевым словом `override`.

Даже если `RecipeIngredient` представляет инициализатор `init(name: String)` как вспомогательный инициализатор, то `RecipeIngredient` тем не менее проводит реализацию всех назначенных инициализаторов своего суперкласса. Таким образом `RecipeIngredient` автоматически наследует все свойства вспомогательных инициализаторов своего суперкласса тоже.

В этом примере суперкласс для класса `RecipeIngredient` является `Food`, который имеет единственный инициализатор `init()`. Поэтому этот инициализатор наследуется `RecipeIngredient`. Наследованная версия `init()` функционирует абсолютно так же как и версия в `Food`, за исключением того, что она делегирует в `RecipeIngredient` версию `init(name: String)`, а не в версию `Food`.

Все три инициализатора могут быть использованы для создания новых `RecipeIngredient` экземпляров:

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs",
quantity: 6)
```

Третий и последний класс в иерархии - подкласс `ShoppingListItem` класса `RecipeIngredient`. `ShoppingListItem` может создавать рецепты из ингредиентов, как только они появляются в листе покупок.

Каждый элемент в листе покупок (shopping list) начинается с “не куплен” или “`unpurchased`”. Для отображения того факта, что `ShoppingListItem` представляет булево свойство `purchased`, со значением по умолчанию `false`. `ShoppingListItem` так же добавляет высчитываемое свойство `description`, которое предоставляет текстовое описание экземпляра `ShoppingListItem`:

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✓" : " X"
        return output
    }
}
```

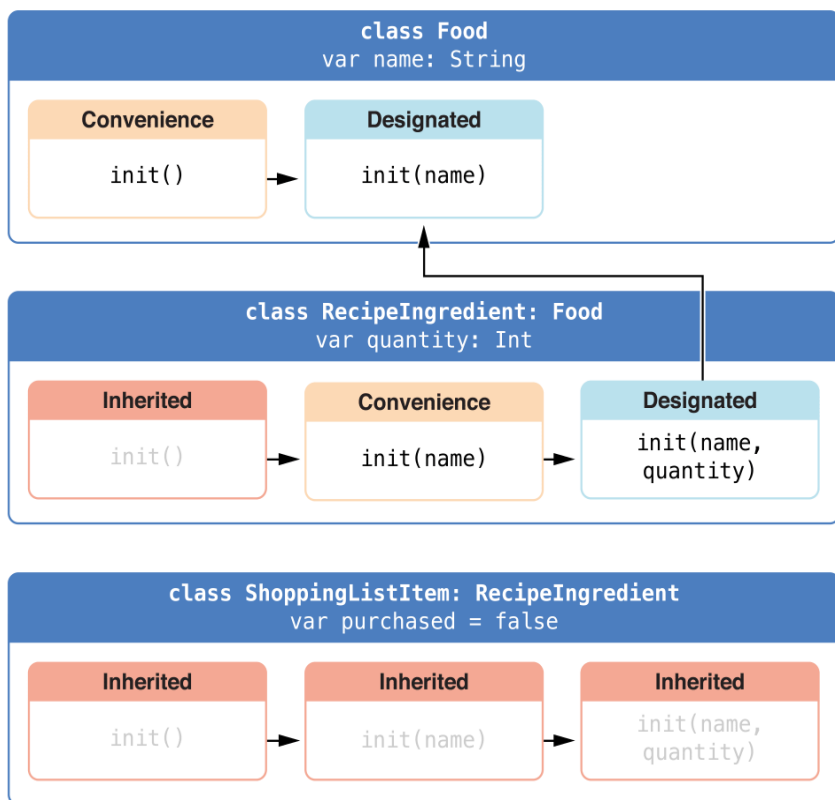
Заметка

`ShoppingListItem` не определяет инициализатор для предоставления исходного значения для `purchased`, потому что элементы в листе покупок, как смоделировано тут, сначала имеют значение `false`, то есть они не куплены.

Так как он предоставляет исходные значения для всех свойств, которые он представляет и не определяет никаких своих инициализаторов, то `ShoppingListItem` автоматически

наследует все назначенные и вспомогательные инициализаторы из своего суперкласса.

Схема ниже отображает общую цепочку инициализаций для всех трех классов:



Вы можете использовать все три унаследованных инициализатора для создания нового экземпляра `ShoppingListItem`:

```

var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Becon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    println(item.description)
}

//1 x Orange juice ✓
//1 x Becon ✗
//6 x Eggs ✗

```

Здесь мы создаем новый массив `breakfastList`, заполняя его тремя экземплярами класса `ShoppingListItem`. Тип массива выводится из `[ShoppingListItem]`. После того как массив создан, мы меняем исходное имя с “[Unamed]” на “Orange juice” и присваиваем свойству `purchased` значение `true`. Потом мы выводим на экран описание каждого элемента массива, где мы можем видеть, что все начальные значения установлены так как мы и ожидали.

Проваливающиеся инициализаторы

Иногда бывает нужно определить класс, структуру или перечисление, инициализация которого может провалиться (претерпеть фиаско!). Такое неисполнение может быть вызвано некорректными значениями параметров или отсутствием требуемого внешнего источника данных или еще какое-нибудь обстоятельство, которое может не позволить завершить инициализацию успешно.

Для того чтобы справиться с условиями инициализации, которые могут провалиться, определите один или несколько проваливающихся инициализатора как часть определения класса, структуры или перечисления. Вы можете написать проваливающийся инициализатор поместив вопросительный после ключевого слова `init` (`init?`).

Заметка

Вы не можете определить проваливающийся инициализатор и обычные инициализаторы с одними и теми же именами и типами параметров.

Проваливающийся инициализатор создает опциональное значение типа, который он инициализирует. Вы пишете `return nil` внутри проваливающегося инициализатора для индикации точки, в которой не срабатывание инициализации может случиться.

Заметка

Строго говоря, инициализаторы не возвращают значений. Их роль заключается в том, что они проверяют, что `self` полностью и корректно инициализирован, до того, как инициализации закончится. Несмотря на то, что вы пишете `return nil` для указания неудачи инициализации, вы не пишете слово `return` в случае, если инициализация прошла успешно.

Пример ниже определяет структуру `Animal`, с константным свойством типа `String` с именем `species`. Структура `Animal` также определяет проваливающийся инициализатор с одним параметром `species`. Этот инициализатор проверяет было ли передано значение из `species` в инициализатор, если оно равно `nil`, то срабатывает проваливающийся инициализатор. В противном случае значение свойства `species` установлено и инициализация проходит успешно.

```
struct Animal {  
  let species: String  
  init?(species: String) {  
    if species.isEmpty { return nil }  
    self.species = species  
  }  
}
```


Вы можете использовать этот проваливающий инициализатор для попытки инициализировать новый экземпляр структуры `Animal` и проверить успешно ли прошла инициализация:

```
let someCreature = Animal(species: "Жираф")
// someCreature имеет тип Animal?, но не
Animal
if let giraffe = someCreature {
    println("Мы инициализировали животное типа
    \(giraffe.species) ")
}
// выведет "Мы инициализировали животное типа
Жираф "
```

Если вы передаете пустую строку в параметр `species` проваливающегося инициализатора, то инициализатор вызывает сбой инициализации:

```
let anonymousCreature = Animal(species: "")
// someCreature имеет тип Animal?, но не
Animal

if anonymousCreature == nil {
    println("Неизвестное животное не может быть
    инициализировано")
}

//выведет "Неизвестное животное не может быть
инициализировано"
```

Заметка

Проверяя значение пустой строки (к примеру "", а не "Жираф") это не тоже самое, что проверять на `nil`, для индикации отсутствия значения опционального `String`. В примере выше, пустая строка ("") корректна и является обычной `String`, а не `String?`. Однако это не допустимо в нашем случае, чтобы животное имело пустое значение, например, свойства `species`. Для того чтобы смоделировать такое ограничение, мы используем проваливающийся инициализатор, который выдает сбой, если находит пустую строку.

Проваливающиеся инициализаторы для перечислений

Вы можете использовать проваливающийся инициализатор для выбора подходящего члена перечисления основываясь на одном или более параметров. Инициализатор может провалиться, если предоставленные параметры не будут соответствовать подходящему члену перечисления.

Пример ниже определяет перечисление `TemperatureUnit` с тремя возможными вариантами (`Kelvin`, `Celsius` и `Fahrenheit`). Проваливающийся инициализатор используется для того, чтобы найти подходящий член перечисления для значения типа `Character`, которое представляет символ температуры:

```
enum TemperatureUnit {  
    case Kelvin, Celsius, Fahrenheit  
    init?(symbol: Character) {  
        switch symbol {  
            case "K":  
                self = .Kelvin  
            case "C":  
                self = .Celsius
```

```

        case "F":
            self = .Fahrenheit
        default:
            return nil
    }
}
}

```

Вы можете использовать этот проваливающийся инициализатор для выбора соответствующего члена из трех возможных состояний и вызвать провал инициализации, если параметр не соответствует этим состояниям:

```

let fahrenheitUnit = TemperatureUnit(symbol:
"F")
if fahrenheitUnit != nil {
    println("Эта единица измерения температура
определена, а значит наша инициализация прошла
успешно!")
}
// выводит "Эта единица измерения температура
определена, а значит наша инициализация прошла
успешно!"
let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    println("Единица измерения температуры не
определена, таким образом мы зафейлили
инициализацию")
}
// выводит "Единица измерения температуры не
определена, таким образом мы зафейлили
инициализацию"

```

Проваливающиеся инициализаторы для перечислений с начальными значениями

Перечисления с начальными значениями по умолчанию получают проваливающийся инициализатор `init?(rawValue:)`, который принимает параметр `rawValue` подходящего типа и выбирает соответствующий член перечисления, если он находит подходящий, или срабатывает сбой инициализации, если существующее значение не находит совпадения среди членов перечисления.

Вы можете переписать пример `TemperatureUnit` из примера выше для использования начальных значений типа `Character` и использовать инициализатор `init?(rawValue:)`:

```
enum TemperatureUnit: Character {
    case Kelvin = "K", Celsius = "C", Fahrenheit
    = "F"
}
let fahrenheitUnit = TemperatureUnit(rawValue:
"F")
if fahrenheitUnit != nil {
    println("Эта единица измерения температура
определена, а значит наша инициализация прошла
успешно!")
}
// выводит "Эта единица измерения температура
определена, а значит наша инициализация прошла
успешно!"
let unknownUnit = TemperatureUnit(rawValue:
"X")
if unknownUnit == nil {
```

```
println("Единица измерения температуры не
определена, таким образом мы зафейлили
инициализацию.")
}
//выводит "Единица измерения температуры не
определена, таким образом мы зафейлили
инициализацию."
```

Проваливающиеся инициализаторы для классов

Проваливающиеся инициализаторы для типов значений (то есть, для структур или перечислений) могут запускать проваливающийся инициализатор в любой точке его процесса инициализации. В структуре `Animal`, которую мы рассматривали ранее, инициализатор запускает провал инициализации в самом начале его реализации, до того как свойство `species` получило значение.

Для классов, однако, проваливающийся инициализатор может вызывать провал инициализации только после того, как все хранимые свойства получат свои исходные значения и произойдет делегация любого инициализатора.

Пример ниже показывает как можно использовать извлеченные неявно опциональные свойства для удовлетворения этому требованию внутри проваливающегося инициализатора класса:

```
class Product {
let name: String!

    init?(name: String) {
        self.name = name
        if name.isEmpty { return nil }
    }
}
```

Класс `Product`, определенный выше, очень похож на структуру `Animal`, которую мы видели ранее. Класс `Product` имеет константное свойство `name`, которое не должно иметь пустого значения типа `String`. Для выполнения этого требования класс `Product` использует проваливающийся инициализатор для проверки того, что значение свойства не пустое, до того, как позволить инициализации успешно завершиться.

Однако `Product` является классом, а не структурой. Это означает, что в отличие от `Animal`, проваливающийся инициализатор для класса `Product` должен предоставлять начальные значения для свойства `name` до того, как произойдет инициализация.

В примере выше свойство `name` класса `Product` определено как неявно извлеченное опциональное `String`, то есть `String!`. Так как она опционального типа, то это значит, что свойство `name` имеет начальное значение `nil`, до того как присваивается другое значение в процессе инициализации. Это значение по умолчанию `nil` в свою очередь значит, что все свойства представленные классом `Product` имеют валидные начальные значения. И в результате проваливающийся инициализатор для `Product` может вызвать провал инициализации в начале инициализатора, если ему передадут пустую строку, до присваивания определенного значения свойству `name` внутри инициализатора.

Из-за того, что свойство `name` является константой, вы можете быть уверены, что оно всегда имеет значение не `nil`, если инициализация прошла успешно. Даже если оно определено при помощи неявно извлеченного опционального типа, вы всегда можете получить доступ к неявно извлеченному значению, без необходимости проверять значение на `nil`:

```
if let bowTie = Product(name: "галстук-бабочка") {  
    // на уже не нужно проверить, что bowTie.name  
    == nil  
    println("Название продукта - \(bowTie.name)")  
}  
//выводит "Название продукта - галстук-бабочка"
```

Распространение проваливающегося инициализатора

Проваливающийся инициализатор класса, структуры, перечисления может быть делегирован к другому проваливающемуся инициализатору из того же класса, структуры, перечисления. Аналогично проваливающийся инициализатор подкласса может быть делегирован вверх в проваливающийся инициализатор суперкласса.

В любом случае, если вы делегируете другому инициализатору, который проваливает инициализацию то и весь процесс инициализации проваливается немедленно за ним, и далее никакой код инициализации уже не исполняется.

Заметка

Проваливающийся инициализатор может также делегировать к непроваливающемуся инициализатору. Используя такой подход, вам следует добавить потенциальное состояние провала в существующий процесс инициализации, который в противном случае не провалиться.

Пример ниже определяет подкласс `CartItem` класса `Product`. `CartItem` создает модель элемент в корзине онлайн заказа. `CartItem` представляет хранимое свойство `quantity` и проверяет, чтобы это свойство всегда имело значение не менее 1:

```
class CartItem: Product {

    let quantity: Int!
    init?(name: String, quantity: Int) {
        super.init(name: name)
        if quantity < 1 { return nil }
        self.quantity = quantity
    }
}
```

Свойство `quantity` имеет неявно извлеченный целочисленный тип `Int!`. Как и в случае со свойством `name` класса `Product`, что означает, что свойство `quantity` имеет значение `nil` до того, как будет присвоено какое-то специфическое значение.

Проваливающийся инициализатор для `CartItem` начинается с делегирования в инициализатор `init(name:)` суперкласса `Product`. Это удовлетворяет требованию, что проваливающийся инициализатор должен всегда проводить делегирование инициализатора до того, как сработает провал инициализации.

Если инициализация суперкласса проваливается из-за того, что значение `name` пустое, то весь процесс инициализации так же проваливается немедленно и никакого кода инициализации больше не выполняется. Если инициализация суперкласса проходит успешно, то инициализатор `CartItem` проверяет, что он получил значение `quantity1` или более.

Если вы создаете экземпляр `CartItem` с `name`, имеющим не пустое значение, и `quantity` равно 1 или более:


```
if let twoSocks = CartItem(name: "носок",
quantity: 2) {
    println("Вещь: \(twoSocks.name), количество:
\(twoSocks.quantity)")
}
//выводит "Вещь: носок, количество: 2"
```

Если вы попытаете создать экземпляр `CartItem` со значением `0`, то инициализатор `CartItem` вызовет провал инициализации:

```
if let zeroShirts = CartItem(name: "футболка",
quantity: 0) {
    println("Вещь: \(zeroShirts.name),
количество: \(zeroShirts.quantity)")
} else {
    println("Невозможно инициализировать ноль
футболок")
}
//выводит "Невозможно инициализировать ноль
футболок"
```

Аналогично, если вы попытаетесь создать экземпляр `CartItem` с пустым значением свойства `name`, инициализатор суперкласса `Product` будет причиной провала инициализации:

```
if let oneUnnamed = CartItem(name: "",
quantity: 1) {
    println("Вещь: \"(oneUnnamed.name)\",
количество: \"(oneUnnamed.quantity)\")
} else {
    println("Невозможно инициализировать один
неименованный продукт")
}
//выводит "Невозможно инициализировать один
неименованный продукт"
```

Переопределение проваливающегося инициализатора

Вы можете переопределить проваливающийся инициализатор суперкласса в подклассе, так же как любой другой инициализатор. Или вы можете переопределить проваливающий инициализатор суперкласса не проваливающимся инициализатором подкласса. Это позволяет вам определить подкласс, для которого инициализация не может провалиться, даже когда инициализация суперкласса позволяет это сделать.

Запомните, что если вы переопределяете проваливающийся инициализатор суперкласса не проваливающимся инициализатором подкласса, инициализатор подкласса не может делегировать вверх, в инициализатор суперкласса. Не проваливающийся инициализатор может никогда не делегировать в проваливающийся инициализатор.

Заметка

Вы можете переопределить проваливающийся инициализатор не проваливающимся инициализатором, но не наоборот.

Пример ниже определяет класс `Document`. Этот класс моделирует документ, который может быть инициализирован свойством `name`, которое имеет может иметь значение отличное от пустого или `nil`, но никак не пустую строку:

```
class Document {
  var name: String?
  //этот инициализатор создает документ со
  //значением nil свойства name
  init() {}
  //этот инициализатор создает документ с не
  //пустым свойством name
  init?(name: String) {
    if name.isEmpty { return nil }
    self.name = name
  }
}
```

Следующий пример определяет подкласс `AutomaticallyNamedDocument` класса `Document`. `AutomaticallyNamedDocument` является подклассом, который переопределяет оба назначенных инициализатора, представленных в `Document`. Это переопределение гарантирует, что экземпляр `AutomaticallyNamedDocument` будет иметь исходное значение "[Untitled]" свойства `name`, если экземпляр создан без имени или если пустая строка передана в инициализатор `init(name:)`:

```

class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}

```

`AutomaticallyNamedDocument` переопределяет проваливающийся инициализатор `init?(name:)` суперкласса не проваливающимся инициализатором `init(name:)`. Из-за того, что `AutomaticallyNamedDocument` справляется с пустой строкой иначе, чем его суперкласс, его инициализатор не обязательно должен провалиться, таким образом он предоставляет не проваливающуюся версию инициализатора вместо проваливающейся.

Проваливающийся инициализатор `init!`

Обычно вы определяете проваливающийся инициализатор, который создает опциональный экземпляр соответствующего типа путем размещения знака вопроса после ключевого слова `init(init?)`. Альтернативно, вы можете определить проваливающийся инициализатор, который создает экземпляр неявно извлекаемого опционала соответствующего типа. Сделать это можно, если вместо вопросительного знака поставить восклицательный знак после ключевого слова `init(init!)`.

Вы можете делегировать от `init?` в `init!` и наоборот, а так же вы можете переопределить `init?` с помощью `init` и наоборот. Вы так же можете делегировать от `init` в `init!`, хотя, делая таким образом, мы заставим сработать утверждение, если `init!` провалит инициализацию.

Требуемые инициализаторы

Напишите `required` перед определением инициализатора класса, если вы хотите, чтобы каждый подкласс этого класса был обязан реализовывать этот инициализатор:

```
class SomeClass {
  required init() {
    //пишем тут реализацию инициализатора
  }
}
```

Вы также должны писать модификатор `required` перед каждой реализацией требуемого инициализатора класса для индикации того, что последующий подкласс так же должен унаследовать этот инициализатор по цепочке. Вы не пишете `override`, когда переопределяете назначенный инициализатор:

```
class SomeSubclass: SomeClass {
  required init() {
    //пишем тут реализацию инициализатора
    подкласса
  }
}
```

Заметка

Вы не должны обеспечивать явную реализацию требуемого инициализатора, если вы можете удовлетворить требование унаследованным инициализатором.

Начальное значение свойства в виде функции и замыкания

Если начальное значение свойства требует какой-то настройки или структуризации, то вы можете использовать замыкание или глобальную функцию, которая будет предоставлять значение для этого свойства. Как только создается новый экземпляр, вызывается функция или замыкание, которая возвращает значение, которое присваивается в качестве начального значения свойства.

Эти виды замыканий или функций обычно создают временное значение того же типа, что и свойство, используя эту величину для отображения желаемого начального состояния, затем возвращают ее в качестве начального значения свойства.

Ниже приведена схема того, как замыкание может предоставлять начальное значение свойству:

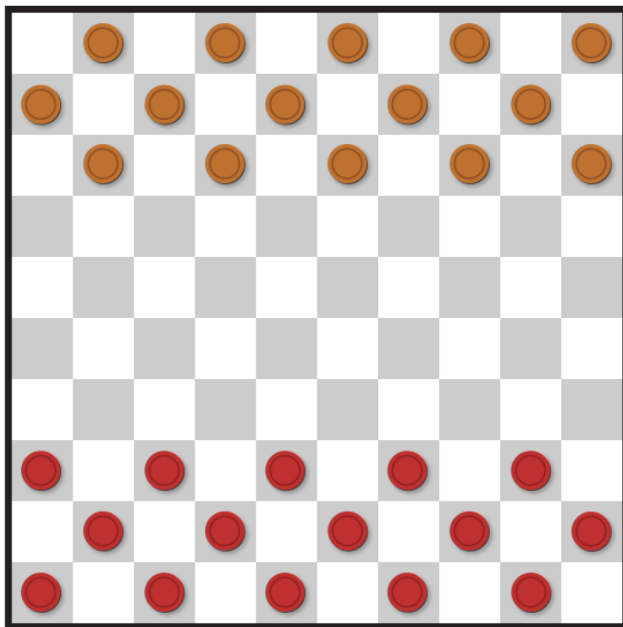
```
class SomeClass {  
    let someProperty: SomeType = {  
        // создаем начальное значения для  
        SomeProperty внутри этого замыкания  
        // someValue должен быть того же типа, что  
        и SomeType  
        return someValue  
    }()  
}
```

Обратите внимание, что после закрывающей фигурной скобки замыкания идут пустая пара круглых скобок. Это говорит Swift, что нужно исполнить это замыкание немедленно. Если вы пропустите эти скобки, то вы присваиваете само значение замыкания свойству, а не возвращаете значения замыкания.

Заметка

Если вы используете замыкание для инициализации свойства, помните, что остальная часть экземпляра еще не инициализирована, на тот момент когда исполняется замыкание. Это значит, что вы не можете получить доступ к значениям других свойств из вашего замыкания, даже если эти свойства имеют начальное значение. Вы так же не можете использовать неявное свойство `self` и не можете вызвать какой-либо метод вашего экземпляра.

Пример ниже определяет структуру `Checkerboard`, которая моделирует доску для игры “Шашки”:



В игру шашки играют на доске 10x10, с чередующимися черными и белыми клетками. Для отображения игровой доски для этой игры, структура `Checkerboard` имеет единственное свойство `boardColors`, которое является массивом из 100 значений типа `Bool`. Значение `true` отображает черные клетки, `false` - белые. Первое значение массива отображает левый верхний угол доски, значение последнего элемента отображает правый нижний угол доски.

Массив `boardColors` инициализирован с помощью замыкания для установки значений цветов:


```

struct Checkerboard {
    let boardColors: [Bool] = {
        var temporaryBoard = [Bool]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    func squareIsBlackAtRow(row: Int, column:
Int) -> Bool {
        return boardColors[(row * 10) + column]
    }
}

```

Когда бы вы не создали новый экземпляр `Checkerboard`, замыкание выполняется, и начальное значение `boardColors` высчитывается и возвращается. Замыкание в примере выше высчитывает временный массив `temporaryBoard` и возвращает его в качестве возвращаемого значения замыканием, как только он весь заполняется. Возвращенный массив сохраняется в `boardColors` и может быть запрошен через метод `squareIsBlackAtRow`:

```

let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column:
1))
// выведет "true"
println(board.squareIsBlackAtRow(9, column:
9))
// выведет "false"

```

Деинициализация

Деинициализатор вызывается сразу после освобождения экземпляра класса. Вы пишете деинициализаторы с ключевого слова `deinit`, аналогично как вы пишете инициализаторы с ключевого слова `init`. Деинициализаторы доступны только для классовых типов.

Как работает деинициализация

Swift автоматически освобождает ваши экземпляры, что освобождает память в свою очередь, когда они больше не нужны. Swift берет на себя управление памятью экземпляров через ARC (automatic reference counting), что будет объяснено чуть позже. Обычно вам не нужно вручную чистить память, когда ваши экземпляры освобождаются. Однако, когда вы работаете с вашими собственными ресурсами, вам, возможно, понадобится проводить дополнительную чистку. К примеру, если вы создаете свой класс для открытия файла и записи в него какой-то информации, а потом его закрываете, то вам понадобится закрыть файл до того как вы освободите экземпляр класса.

Определения класса могут иметь максимум один деинициализатор на один класс. Деинициализатор не принимает ни одного параметра и пишется без круглых скобок:

```
deinit {  
    // проведение деинициализации  
}
```

Деинициализаторы вызываются автоматически прямо перед тем как освобождается экземпляр. У вас нет возможности вызывать деинициализатор самостоятельно.

Деинициализаторы суперкласса наследуются их подклассами, и деинициализаторы суперкласса вызываются автоматически в конце реализации деинициализатора подкласса. Деинициализаторы суперклассов всегда вызываются, даже если подкласс не имеет своего деинициализатора.

Так как экземпляр не освобождается до тех пор пока не будет вызван деинициализатор, то деинициализатор может получить доступ ко всем свойствам экземпляра, который он вызывает, и может изменить свое поведение, основываясь на этих свойствах(например, имя файла, который должен быть закрыт).

Деинициализаторы в действие

Здесь приведен пример деинициализатора в действии. Для создания простой игры в этом примере определяются два новых типа `Bank` и `Player`. Структура `Bank` управляет наличностью, которой не может быть больше 10000 монет в обороте. В игре может быть только один `Bank`, так что `Bank` реализован в качестве структуры со статическими свойствами и методами для хранения и управления текущим состоянием:

```
struct Bank {
    static var coinsInBank = 10000
    static func vendCoins(var
numberOfCoinsToVend: Int) -> Int {
    numberOfCoinsToVend =
min(numberOfCoinsToVend, coinsInBank)
    coinsInBank -= numberOfCoinsToVend
    return numberOfCoinsToVend
}
    static func recieveCoins(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` следит за текущим количеством монет, которые у него есть в свойстве `coinsInBank`. Так же он предоставляет два метода `vendCoins`, `recieveCoins` для обработки событий коллекционирования монет и их распределения.

Метод `vendCoins` проверяет достаточность количества монет в банке перед их дистрибуцией. Если в банке недостаточно монет, то `Bank` возвращает меньшее число, чем было запрошено (и возвращает ноль, если монет в банке не

осталось). Метод `vendCoins` объявляет `numberOfCoinsToVend` как переменный параметр, так что число может быть модифицировано внутри тела метода, без какой-либо необходимости объявлять новую переменную. Он возвращает целое значение для отображения актуального количества монет, которые были предоставлены.

Метод `recieveCoins` просто добавляет полученное число монет обратно в банк.

Класс `Player` описывает игрока в игре. Каждый игрок имеет определенное количество монет, хранящихся для любых целей, которые могут быть использованы в любое время. Это отображается свойством `coinsInPurse`:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
        Bank.recieveCoins(coinsInPurse)
    }
}
```

Каждый экземпляр `Player` инициализирован с начальным допущением определенного количества монет из банка, хотя `Player` может получить и меньшее количество монет, если недостаточно монет в банке.

Класс `Player` определяет метод `winCoins`, который получает определенное число монет от банка и добавляет их в кошелек игрока. Класс `Player` так же реализует деинициализатор, который вызывается сразу после того, как

экземпляр освобождается. В примере ниже деинициализатор просто возвращает все монеты игрока в банк:

```
var playerOne: Player? = Player(coins: 100)
println("Новый игрок присоединился к игре со
\ (playerOne!.coinsInPurse) монетами")
// выводит "Новый игрок присоединился к игре
со 100 монетами"
println("\ (Bank.coinsInBank) монет осталось в
банке")
// выводит "9900 монет осталось в банке"
Новый экземпляр Player создан со 100 монетами, если такое
количество есть в банке. Экземпляр хранит опциональную
переменную playerOne типа Player. Опциональный тип
используется здесь, поскольку игрок может покинуть игру в
любой момент. Опционал позволяет вам отслеживать
присутствие игрока в игре.
```

Так как playerOne является опционалом, то используем восклицательный знак (!), когда мы хотим получить доступ к его свойству coinsInPurse или когда вызываем метод winCoins:

```
playerOne!.winCoins(2000)
println("PlayerOne выиграл 2000 монет и сейчас
у него \ (playerOne!.coinsInPurse) монет")
// выводит "PlayerOne выиграл 2000 монет и
сейчас у него 2100 монет"
println("В банке осталось \ (Bank.coinsInBank)
монет")
// выводит "В банке осталось 7900 монет"
```

Здесь наш игрок выиграл 2000 монет. Теперь у него в кошельке 2100 монет, соответственно, в банке осталось 7900 монет.

```
playerOne = nil
println("PlayerOne покинул игру")
// выводит "PlayerOne покинул игру"
println("Сейчас в банке \(Bank.coinsInBank)
монет")
// выводит "Сейчас в банке 10000 монет"
// этот пример будет корректно работать только
в проекте, а не в песочнице
```

Игрок покинул игру. Это осуществимо, если мы присвоим опционалу `playerOne` значение `nil`, что значит, что игрока больше нет. На этот момент мы больше не имеем доступа к свойствам или методам переменной `playerOne`, то есть у нас сломана ссылка на экземпляр класса `Player`, так что экземпляр освобождается и освобождается память. Прямо перед тем как это случится, автоматически вызывается инициализатор, который возвращает монеты в банк.

Автоматический подсчет ссылок (ARC)

Swift использует *automatic reference counting* (автоматический подсчет ссылок) для отслеживания и управления памятью вашего приложения. В большинстве случаев это означает, что управление памятью "просто работает" в Swift и вам не нужно думать о самостоятельном управлении памятью. ARC автоматически освобождает память, которая использовалась экземпляром класса, когда эти экземпляры больше нам не нужны.

Однако не в некоторых случаях для управления памятью ARC нужно больше информации об отношениях между некоторыми частями вашего кода. Эта глава опишет эти случаи и покажет как включить ARC, чтобы эта система взяла на себя весь контроль памятью вашего приложения.

Заметка

ARC применима только для экземпляров класса. Структуры и перечисления являются типами значений, а не ссылочными типами, и они не передают свои значения по ссылке.

Работа ARC

Каждый раз, когда вы создаете экземпляр класса, ARC выделяет кусок памяти для хранения информации этого экземпляра. Этот кусок памяти содержит информацию о типе экземпляра, о его значении и любых хранимых свойствах связанных с ним.

Дополнительно, когда экземпляр больше не нужен, ARC освобождает память, использованную под этот экземпляр, и направляет эту память туда, где она нужна. Это своего рода гарантия того, что ненужные экземпляры не будут занимать память.

Однако, если ARC освободит память используемого экземпляра, то доступ к свойствам или методам этого экземпляра будет невозможен. Если вы попытаетесь получить доступ к этому экземпляру, то ваше приложение скорее всего выдаст ошибку и будет остановлено.

Для того, чтобы нужный экземпляр не пропал, ARC ведет учет количества свойств, констант, переменных, которые ссылаются на каждый экземпляр класса. ARC не освободит экземпляр, если есть хотя бы одна активная ссылка.

Для того чтобы это было возможно, каждый раз как вы присваиваете экземпляр свойству, константе или переменной создается *strong reference* (сильная связь) с этим экземпляром. Такая связь называется “сильной”, так как она крепко держится за этот экземпляр и не позволяет ему освободиться до тех пор, пока остаются сильные связи.

ARC в действии

Приведем пример того, как работает ARC. Наш пример начнем с класса `Person`, который определяет константное свойство `name`:

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        println("\ (name) инициализируется")
    }
    deinit {
        println("\ (name) освобождается")
    }
}
```

Класс `Person` имеет инициализатор, который устанавливает name свойство экземпляра и выводит сообщение для отображения того, что идет инициализация. Так же класс `Person` имеет деинициализатор, который выводит сообщение, когда экземпляр класса освобождается.

Следующий кусок кода определяет три переменные класса `Person?`, который используется для установки нескольких ссылок к новому экземпляру `Person` в следующих кусках кода. Так как эти переменные опционального типа `Person?`, а не `Person`, они автоматически инициализируются со значением `nil`, и не имеют никаких ссылок на экземпляр `Person`.

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

Теперь вы можете создать экземпляр класса `Person` и присвоить его одной из этих трех переменных:

```
reference1 = Person(name: "John Appleseed")  
// выводит "John Appleseed инициализируется"
```

Обратите внимание, что сообщение "`John Appleseed инициализируется`" выводится во время того, как вы вызываете инициализатор класса `Person`. Это подтверждает тот факт, что происходила инициализация.

Так как новый экземпляр класса `Person` был присвоен переменной `reference1`, значит теперь существует сильная ссылка между `reference1` и новым экземпляром класса `Person`. Теперь у этого экземпляра есть как минимум одна сильная ссылка, значит ARC держит под `Person` память и не освобождает ее.

Если вы присвоите другим переменным тот же экземпляр `Person`, то добавится две сильные ссылки к этому экземпляру:

```
reference2 = reference1  
reference3 = reference1
```

Теперь экземпляр класса `Person` имеет три сильные ссылки.

Если вы сломаете две из этих трех ссылок (включая и первоначальную ссылку), присвоим `nil` двум переменным, то останется одна сильная ссылка, и экземпляр `Person` не будет освобожден:

```
reference1 = nil  
reference2 = nil
```

ARC не освободит экземпляр класса `Person` до тех пор, пока остается последняя сильная ссылка, уничтожив которую мы укажем на то, что наш экземпляр больше не используется:

```
reference3 = nil
// выводит "John Appleseed освобождается"
```

Циклы сильных ссылок между экземплярами классов

В примерах ранее ARC было в состоянии отслеживать количество ссылок к новому экземпляру `Person`, который вы создали, и освободить его, если этот экземпляр уже более не нужен.

Однако возможно написать код, в котором экземпляр класса *никогда* не будет иметь нулевое число сильных ссылок. Это может случиться, если экземпляры классов имеют сильные связи друг с другом, что не позволяет им освободиться. Это известно как *цикл сильных ссылок*.

Вы сами решаете, когда сделать вместо сильной (`strong`) ссылки слабую (`weak`) или бесхозную (`unowned`). Однако перед тем как узнать, в каких случаях разрешить сильный ссылочный цикл, давайте узнаем что вызывает его.

Ниже приведен пример того, как сильный ссылочный цикл может быть создан по ошибке. В этом примере мы определяем два класса `Person` и `Apartment`, которые создают модель блока квартир с их жителями:

```
class Person {  
  
    let name: String  
    init(name: String) {  
        self.name = name  
    }  
    var apartment: Apartment?  
    deinit {  
        println("\ (name) освобождается")  
    }  
}  
class Apartment {  
    let number: Int  
    init(number: Int) {  
        self.number = number  
    }  
    var tenant: Person?  
    deinit {  
        println("Апартаменты под номером \ (number)  
освобождаются")  
    }  
}
```

Каждый экземпляр `Person` имеет свойство `name` типа `String` и опциональное свойство `apartment`, которое изначально `nil`. Свойство `apartment` опционально, так как наша персона не обязательно всегда должна иметь апартаменты.

Аналогично, что каждый экземпляр `Apartment` имеет свойство `number` типа `Int` и опциональное свойство `tenant`, которое изначально `nil`. Свойство `tenant` опциональное, потому как не всегда в апартаментах кто-то живет.

Оба этих класса определяют деинициализатор, который отображает факт того, что экземпляр освобождился. Это позволяет вам видеть освободились ли экземпляры этих классов как вы ожидали или нет.

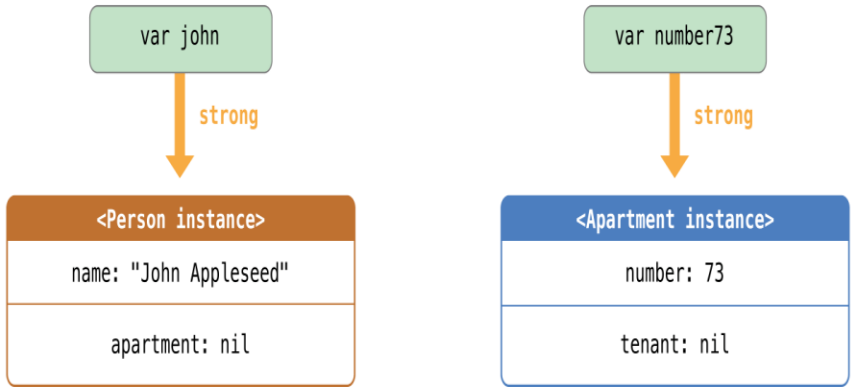
Следующий фрагмент кода определяет две опциональные переменные с именами `john` и `number73`, которые будут назначены определенным экземплярам классов `Apartment` и `Person`. Оба значения переменных равны `nil`, в силу того, что они опциональны:

```
var john: Person?
var number73: Apartment?
```

Теперь вы можете создать свои экземпляры `Person` и `Apartment` и присвоить их этим переменным `john`, `number73`:

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

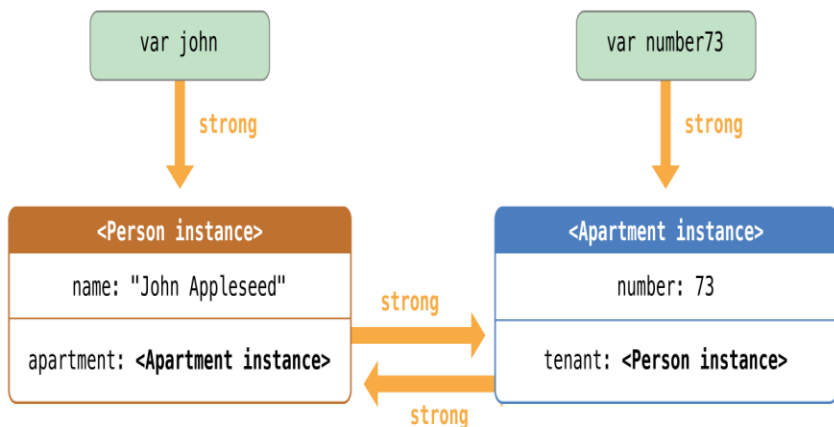
Вот как выглядят сильные связи после того создания и присваивания этих двух экземпляров. Переменная `john` имеет сильную связь с экземпляром класса `Person`, переменная `number73` имеет сильную связь с экземпляром `Apartment`:



Теперь вы можете соединить эти два экземпляра вместе, так что житель будет иметь апартаменты, а апартаменты будут иметь своих жителей. Обратите внимание, что восклицательный знак (!) используется для развертывания и допуска к экземплярам, хранимым в опциональных переменных `john`, `number73`, так что установить значения свойством можно в такой форме:

```
john!.apartment = number73
number73!.tenant = john
```

Вот как выглядят сильные связи после того, как мы соединили экземпляры:



К сожалению, соединяя таким образом, образуется цикл сильных ссылок между экземплярами. Экземпляр `Person` имеет сильную ссылку на экземпляр `Apartment`, экземпляр `Apartment` имеет сильную ссылку на экземпляр `Person`. Таким образом, когда вы разрушаете сильные ссылки, принадлежащие переменным `john` и `number73`, их количество все равно не падает до нуля, и экземпляры не освобождаются:

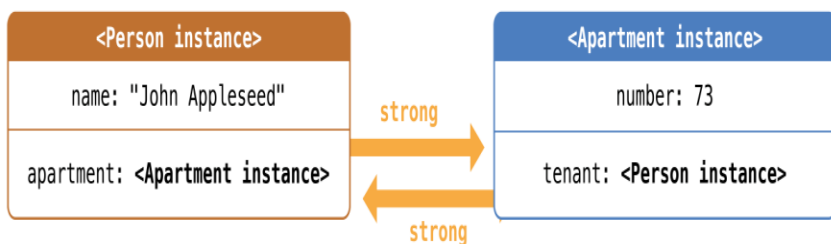
```
john = nil
number73 = nil
```

Обратите внимание, что ни один деинициализатор не был вызван, когда вы присваивали `nil`. Цикл сильных ссылок предотвратил экземпляры `Person` и `Apartment` от освобождения, что вызывает утечку памяти в вашем приложении.

Вот как выглядят сильные ссылки после того, как вы присвоили `nil` переменным, `john`, `number73`:

var john

var number73



Сильные взаимные ссылки остались между экземплярами `Person` и `Apartment` и не могут быть разрушены.

Замена циклов сильных ссылок между экземплярами классов

Swift предлагает два способа переопределить ссылку, чтобы она была не сильной, а слабой или безхозной.

Слабые и безхозные ссылки позволяют одному экземпляру в цикле ссылок ссылаться на другой экземпляр без сильного прикрепления. Экземпляры могут ссылаться друг на друга без создания цикла сильных связей.

Используйте слабые ссылки, когда есть вероятность того, что в какой-то момент ссылка станет `nil`. И наоборот, используйте безхозные ссылки, когда вы точно знаете, что она никогда не станет `nil`, после того как будет инициализирована.

Слабые (weak) ссылки

Слабые ссылки не удерживаются за экземпляр, на который они указывают, так что ARC не берет их во внимание, когда считает ссылки экземпляра. Такой подход позволяет избежать ситуации, когда ссылка становится частью цикла сильных ссылок. Вы указываете слабую ссылку ключевым словом `weak` перед именем объявляемого свойства или переменной.

Используйте слабые связи для избежания ссылочных циклов, когда есть вероятность того, что в какой-то момент времени ссылка будет иметь значение “`nil`”. Если ссылка будет всегда иметь значение, то используйте безхозные ссылки, которые будут описаны далее. В примере с апартаментами выше,

есть некоторая вероятность того, что апартаменты могут иметь “отсутствие” жильцов в какой-то момент времени, так что в нашем случае слабая ссылка подходит для разрыва цикла сильных связей.

Заметка

Слабые (*weak*) ссылки должны объявляться как переменные, что отображает изменчивость их значений с течением времени. Слабые ссылки не могут быть константами.

Из-за того, что слабым ссылкам разрешается иметь *nil*, что означает “отсутствие значения”, то вы должны объявлять каждую слабую ссылку, как ту, которая имеет опциональное значение. Опциональные типы предпочтительный тип для отображения, потому что переменная может не иметь значения в Swift.

Так как слабая ссылка не сильно держится за экземпляр, на который указывает, то можно освободить экземпляр, на который указывает такая ссылка. Таким образом ARC автоматически присваивает слабой ссылке *nil*, когда экземпляр, на который она указывает, освобождается. Вы можете проверить существование значения в слабой ссылке точно так же как и с любыми другими опциональными значениями, и вы никогда не будете иметь ссылку с недопустимым значением, например, указывающую на несуществующий экземпляр.

Пример ниже идентичен тому, что мы разбирали с вами с классами *Person*, *Apartment*, но только теперь в нем есть одно существенное отличие. В этот раз свойство *tenant* экземпляра класса *Apartment* объявлено как слабая ссылка:

```

class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being
deinitialized") }
}
class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    weak var tenant: Person?
    deinit { println("Apartment #\(number) is
being deinitialized") }
}

```

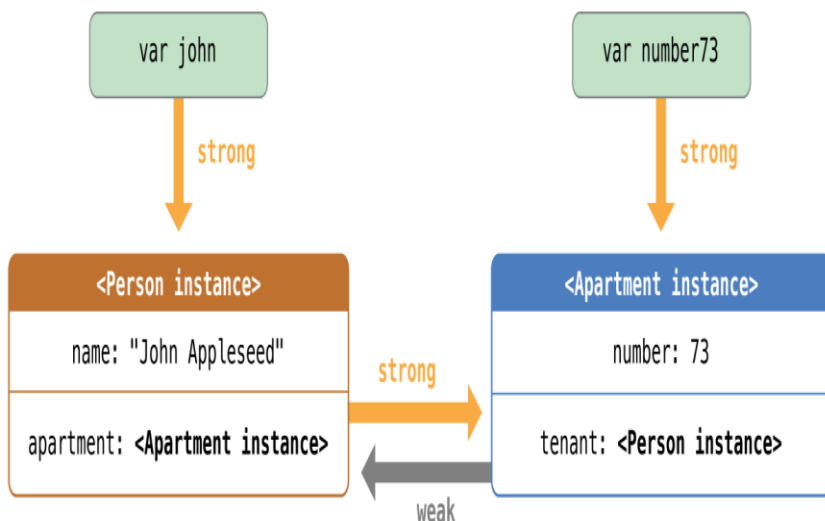
Создадим как и в предыдущем примере сильные ссылки от двух переменных (`john`, `number73`) и связи между двумя экземплярами:

```

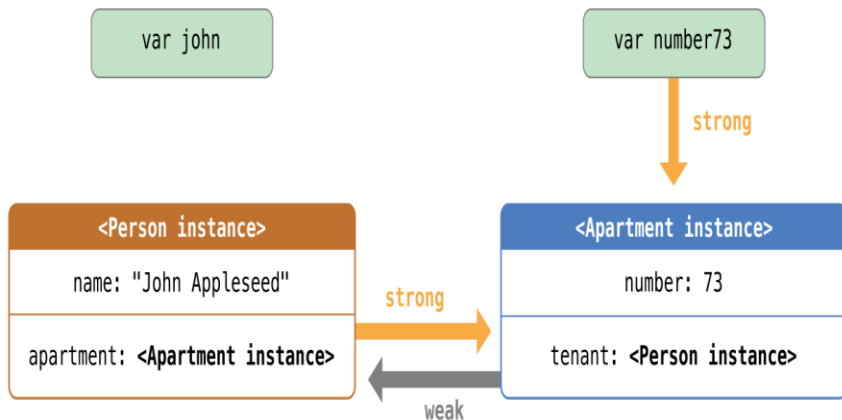
var john: Person?
var number73: Apartment?
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
john!.apartment = number73
number73!.tenant = john

```

Вот как теперь выглядит соединение двух экземпляров между собой:



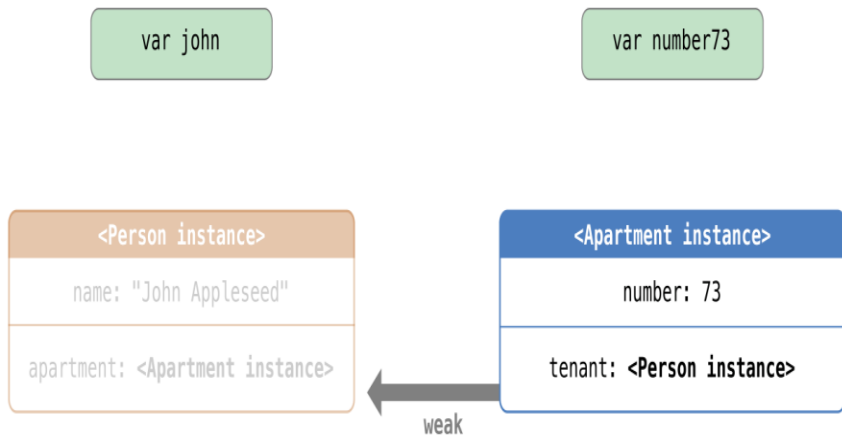
Экземпляр `Person` все еще имеет сильную ссылку на экземпляр `Apartment`, но `Apartment` имеет слабую (`weak`) ссылку на экземпляр `Person`. Это означает, что когда вы разрушаете сильную ссылку, которая содержится в переменной `john`, то больше сильных ссылок, указывающих на экземпляр `Person`, не остается:



А так как больше сильных ссылок на экземпляр `Person` нет, то он освобождается:

```
john = nil
// выводит "John Appleseed освобождается"
```

Остается только одна сильная ссылка на экземпляр `Apartment` из переменной `number73`. Если вы разрушите эту сильную ссылку, то их общее количество станет равным нулю:



А так как больше сильных ссылок нет, то и экземпляр `Apartment` тоже освобождается:

```
number73 = nil
// выводит "Апартаменты под номером 73
освобождаются"
```

Два последних фрагмента кода выше показывают, что деинициализаторы экземпляров `Person`, `Apartment` вывели свое сообщение на экран, после того как переменным `john`, `number73` были присвоены `nil`. Это доказывает, что наш цикл сильных ссылок был разрушен.

Бесхозные ссылки

Как и слабые ссылки, бесхозные ссылки также не имеют сильной связи с экземпляром, на который они указывают. В отличие от слабых ссылок, бесхозные ссылки всегда имеют значение. Из-за этого бесхозные ссылки имеют неопциональный тип. Вы указываете на то, что ссылка

бесхозная ключевым словом `unowned`, поставленным перед объявлением свойства или переменной.

Так как бесхозная ссылка не является опциональной, то вам не нужно и разворачивать ее каждый раз, когда вы собираетесь ее использовать. Вы можете обратиться к бесхозной ссылке напрямую. Однако ARC не может установить значение ссылки на `nil`, когда экземпляр, на который она ссылается, освобожден, так как переменные неопционального типа не могут иметь значения `nil`.

Заметка

Если вы попытаетесь получить доступ к бесхозной ссылке после того, как экземпляр, на который она ссылается освобожден, то выскочит `runtime` ошибка. Используйте бесхозные ссылки только в том случае, если вы абсолютно уверены в том, ссылка всегда будет указывать на экземпляр. Обратите внимание, что Swift гарантирует, что ваше приложение прекратит работу из-за ошибки, если вы попытаетесь обратиться к бесхозной ссылке, после того как экземпляр, на который она указывает, будет освобожден. В такой ситуации поведение будет всегда именно таким. Ваше приложение точно будет всегда выкидывать вам ошибку, хотя вы должны, конечно, предотвратить это.

Следующий пример определяет два класса `Customer` и `CreditCard`, которые обыгрывают ситуацию клиента банка и кредитной карточки для этого клиента. Эти оба класса содержат экземпляры друг друга в качестве свойства. Такое взаимоотношение классов является потенциальной возможностью образования зацикливания сильных ссылок,

Взаимоотношения между `Customer` и `CreditCard` немного отличаются от предыдущего примера `CApartment` и `Person`. В этом случае клиент может иметь или не иметь кредитной

карты, но кредитная карта всегда имеет владельца. Чтобы это отобразить, класс `Customer` имеет опциональное свойство `card`, а `CreditCard` имеет неопциональное свойство `customer`.

Более того, новый экземпляр `CreditCard` может быть только создан путем передачи значения `number` и экземпляра `customer` в инициализатор класса `CreditCard`. Это гарантирует, что экземпляр `CreditCard` всегда будет иметь экземпляр `customer`, который будет связан с ним, когда экземпляр `CreditCard` будет создан.

Так как кредитная карта всегда будет иметь своего хозяина, вы определяете свойство `customer` как бесхозное, для избежания цикла сильных ссылок:

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { println("\(name) освобождается")
}
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { println("Карта #\(number) освобождается") }
}
```

Заметка

Свойство `number` класса `CreditCard` определено как значение типа `UInt64`, а не `Int`, для того, чтобы оно было достаточно большим, чтобы хранить числа с 16 цифрами и на 32, и на 64 разрядных системах.

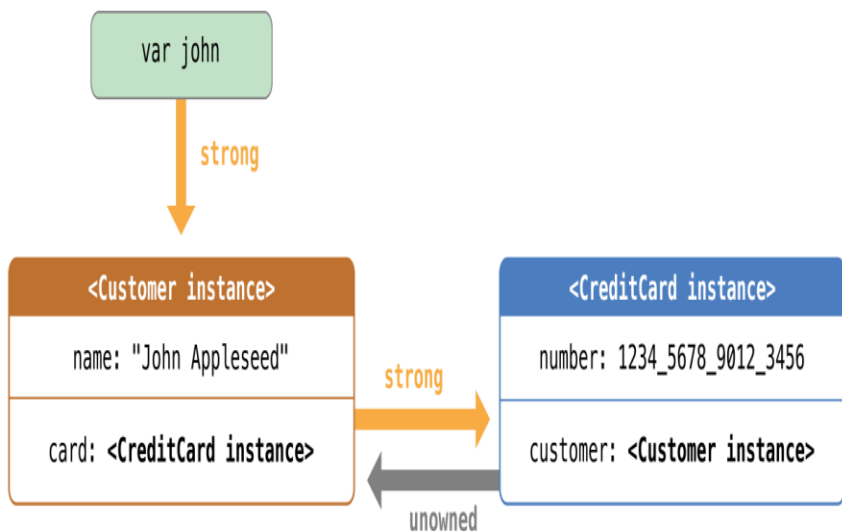
Следующий кусок кода определяет опциональную переменную типа `Customer?` с именем `john`, которая будет использоваться для хранения ссылки на определенного клиента. Эта переменная имеет начальное значение `nil`, в силу того, что это опциональный тип:

```
var john: Customer?
```

Вы можете создать экземпляр `Customer` и использовать его для инициализации и присваивания нового экземпляра `CreditCard`, как свойство клиентской `card`:

```
john = Customer(name: "John Appleseed")  
john!.card = CreditCard(number:  
1234567890123456, customer: john!)
```

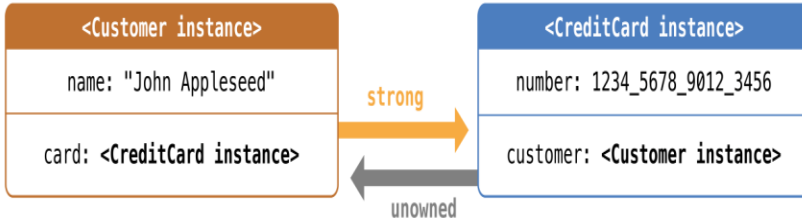
Вот как выглядят ссылки, после того как вы связали эти два экземпляра:



Экземпляр `Customer` имеет сильную ссылку на экземпляр `CreditCard`, а экземпляр `CreditCard` имеет бесхозную ссылку на экземпляр `Customer`.

Из-за того, что ссылка `customer` является бесхозной, то при разрушении сильной ссылки, которая находится в переменной `john`, больше сильных ссылок, указывающих на экземпляр `Customer` не остается:

```
var john
```



Из-за того, что более сильных ссылок, ссылающихся на экземпляр `Customer` нет, то этот экземпляр освобождается. После того, как это происходит, у нас не остается больше сильных ссылок, указывающих на экземпляр `CreditCard`, так что он тоже освобождается:

```
john = nil
// выводит "John Appleseed освобождается"
// выводит "Карта #1234567890123456
освобождается"
```

Последний кусок кода показывает нам, что инициализаторы экземпляров `Customer` и `CreditCard` напечатали свои сообщения деинициализации, после того, как переменной `john` был присвоен `nil`.

Бесхозные ссылки и неявно извлеченные опциональные свойства

Примеры, приведенные выше, для слабых и бесхозных ссылок, описывают два из самых распространенных

сценариев, где существует необходимость разрушения цикла сильных ссылок.

Пример с `Person`, `Apartment` показывает ситуацию, где два свойства, оба из которых могут иметь значение `nil`, имеют потенциальную возможность образования цикла сильных связей. Этот случай лучше всего решается с помощью слабой связи.

Пример с `Customer`, `CreditCard` демонстрирует ситуацию, где одному свойству разрешено иметь значение `nil`, другому - нет. Однако здесь так же существует потенциальная возможность образования цикла сильных ссылок. Такой случай лучше всего разрешается с помощью бесхозных ссылок.

Однако есть и третий вариант, в котором оба свойства должны всегда иметь значение, и ни одному из них нельзя иметь `nil`, после завершения инициализации. В этом случае лучше всего скомбинировать бесхозное свойство одного класса с неявно извлеченным опциональным свойством другого класса.

Это позволяет получить доступ к обоим свойствам напрямую (без опционального извлечения) после завершения инициализации, так же позволяя избегать взаимных сильных ссылок. В этой секции вы узнаете как создать такие взаимоотношения:

Пример внизу определяет два класса `Country`, `City`, каждый из которых хранит экземпляр другого класса в качестве свойства. В такой модели каждая страна должна иметь столицу, а каждый город, должен иметь страну. Для того, чтобы это отобразить, класс `Country` имеет свойство `capitalCity`, а класс `City` имеет свойство `country`:

```

class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name:
capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}

```

Для создания такой внутренней зависимости между этими двумя классами, инициализатор `City` берет экземпляр `Country` и сохраняет его в свойство `country`.

Инициализатор `City`, вызывается из инициализатора `Country`. Однако инициализатор `Country` не может передавать `self` в инициализатор `City` до тех пор, пока новый экземпляр `Country` не будет полностью инициализирован, что описано в разделе “Двухфазная инициализация”.

Объединив все с этим требованием, вы объявляете свойство `capitalCity` класса `Country` как неявно извлеченное опциональное свойство, отображаемое восклицательным знаком в конце аннотации типа (`City!`). Это значит, что свойство `capitalCity` имеет начальное значение равное `nil`, как и в случае с другими опционалами, но к

которому можно обратиться без предварительного развертывания значения, что описано в главе “Неявно извлеченные опционалы”.

Так как свойство `capitalCity` имеет значение по умолчанию `nil`, то новый экземпляр `Country` считается полностью инициализированным, как только экземпляр `Country` устанавливает свойство `name` с помощью своего инициализатора. Это значит, что инициализатор `Country` может ссылаться на неявное свойство `self` и раздавать его, как только свойство `name` получит корректное значение. Инициализатор `Country` может таким образом передать `self` в качестве одного из параметров для инициализатора `City`, когда инициализатор `Country` устанавливает свое собственное свойство `capitalCity`.

Из всего этого можно сделать вывод, что вы можете создать экземпляры `Country` и `City` единственным выражением, без создания цикла сильных ссылок друг на друга. Получить значение свойства `capitalCity` можно напрямую без использования восклицательного знака для извлечения опционального значения:

```
var country = Country(name: "Россия",
capitalName: "Москва")
println("Столицей страны \(country.name)
является \(country.capitalCity.name)")
// выводит "Столицей страны Россия является
Москва"
```

В примере выше использование неявно извлеченного опционала означает, что все требования двухфазного инициализатора класса выполнены. Свойство `capitalCity` может быть использовано как

неопциональное значение, после того как инициализация закончена, все так же избегая цикла сильных ссылок.

Циклы сильных ссылок для замыкания

Как вы видели ранее, циклы сильных ссылок могут быть созданы двумя экземплярами классов, когда они поддерживают друг на друга сильные ссылки. Вы так же видели как использовать слабые (*weak*) или бесхозные (*unowned*) ссылки для того, чтобы заменить ими сильные (*strong*).

Сильные ссылки так же могут образовываться, когда вы присваиваете замыкание свойству экземпляра класса, и тело замыкания захватывает экземпляр. Этот захват может случиться из-за того, что тело замыкания получает доступ к свойству экземпляра, например `self.someProperty`, или из-за того, что замыкание вызывает метод типа `self.someMethod()`. В обоих случаях эти доступы и вызывают тот самый “захват” `self`, при этом создавая цикл сильных ссылок.

Этот цикл возникает из-за того, что замыкания, как и классы, являются ссылочными типами. Когда вы присваиваете замыкание свойству, вы присваиваете ссылку на это замыкание. В общем, проблема та же, что и ранее: две сильные ссылки, которые не дают друг другу освободиться. Однако в отличии от предыдущих примеров здесь не два экземпляра классов, а замыкание и один экземпляр класса, которые поддерживают существование друг друга.

Swift предлагает элегантное решение этой проблемы, которые известно как список захвата замыкания (*closure*)

capture **list**). Однако до того, как вы узнаете, как разрушить такой цикл с помощью этого решения, давайте разберемся, что этот цикл может вызвать.

Пример ниже отображает, как вы можете создать цикл сильных ссылок, когда мы используем замыкание, которое ссылается на **self**. В этом примере определяем класс **HTMLElement**, который представляет модель простого элемента внутри HTML документа:

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "\ (text) \ (self.name) >"
        } else {
            return ""
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        println("\ (name) деинициализируется")
    }

}
```

Класс **HTMLElement** определяет свойство **name**, которое отображает имя элемента, например “**p**” тег для отображения параграфа или “**br**” для тэга перехода на следующую строку.

Класс `HTMLElement` также определяет опциональное свойство `text`, которому может быть присвоена строка, которая отображает текст, который может быть внутри `HTML` элемента.

В дополнение к этим двум простым свойствам класс `HTMLElement` определяет ленивое свойство `asHTML`. Это свойство ссылается на замыкание, которое комбинирует `name`, `text` во фрагмент HTML строки. Свойство `asHTML` имеет тип `() -> String`, или другими словами функция, которая не принимает параметров и возвращает строку.

По умолчанию свойству `asHTML` присвоено замыкание, которое возвращает строку, отображающую тэг HTML. Этот тэг содержит опциональный `text`, если таковой есть или не содержит его, если `text`, соответственно, отсутствует. Для элемента параграфа замыкание вернет `<p>some text</p>` или просто `<p />`, в зависимости от того, имеет ли свойство `text` какое либо значение или `nil`.

Свойство `asHTML` называется и используется несколько схоже с методом экземпляра. Однако из-за того что `asHTML` является свойством-замыканием, а не методом экземпляра, то вы можете заменить значение по умолчанию свойства `asHTML` на пользовательское замыкание, если вы хотите сменить отображение конкретного `HTML` элемента.

Заметка

Свойство `asHTML` объявлено как ленивое свойство, потому что оно нам нужно только тогда, когда элемент должен быть отображен в виде строкового значения для какого-либо HTML элемента выходного значения. Факт того, что свойство `asHTML` является ленивым, означает, что вы можете сослаться на `self` внутри дефолтного замыкания, потому что обращение к ленивому свойству невозможно до тех пор, пока

инициализация полностью не закончится и не будет известно, что `self` уже существует.

Класс `HTMLElement` предоставляет единственный инициализатор, который принимает аргумент `name` и (если хочется) аргумент `text` для инициализации нового элемента. Класс также определяет деинициализатор, который выводит сообщение, для отображения момента когда экземпляр `HTMLElement` освобождается.

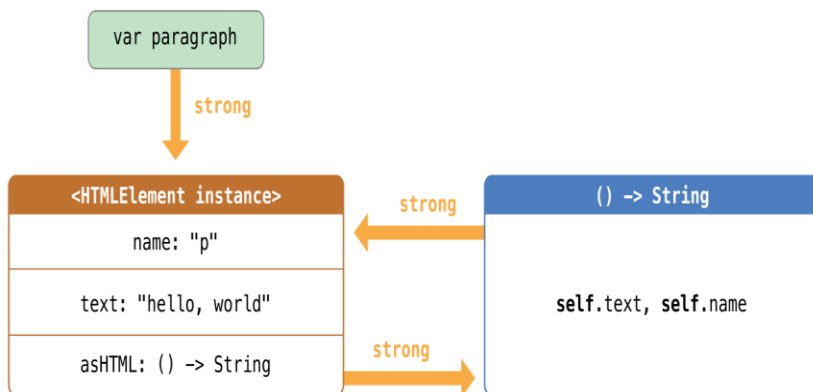
Вот как вы используете класс `HTMLElement` для создания и вывода нового экземпляра:

```
var paragraph: HTMLElement? =
  HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// выводит "<p>hello, world</p>"
```

Заметка

Переменная `paragraph` определена как опциональный `HTMLElement`, так что он может быть и `nil` для демонстрации цикла сильных ссылок.

К сожалению класс `HTMLElement`, который описан выше, создает цикл сильных ссылок между экземпляром `HTMLElement` и замыканием, использованным для его исходного значения `asHTML`. Вот как выглядит этот цикл:



Свойство `asHTML` экземпляра держит сильную ссылку на его замыкание. Однако из-за того, что замыкание ссылается на `self` внутри своего тела (`self.name`, `self.text`), оно захватывает `self`, что означает, что замыкание держит сильную ссылку обратно на экземпляр `HTMLElement`. Между ними двумя образуется цикл сильных ссылок. (Для более подробной информации по захвату значений в замыканиях читайте соответствующий раздел [“Захват значений”](#).)

Заметка

Даже несмотря на то, что замыкание ссылается на `self` несколько раз, оно все равно захватывает всего одну сильную ссылку на экземпляр `HTMLElement`.

Если вы установите значение `paragraph` на `nil`, чем разрушите сильную ссылку на экземпляр `HTMLElement`, то ни экземпляр `HTMLElement`, ни его замыкание не будут освобождены из-за цикла сильных ссылок:

```
paragraph = nil
```

Обратите внимание, что сообщение деинициализатора `HTMLElement` не выводится на экран, что и есть факт того, что этот экземпляр не освобожден.

Замена циклов сильных ссылок для замыканий

Заменить цикл сильных ссылок между замыканием и экземпляром класса можно путем определения списка захвата в качестве части определения замыкания. Список захвата определяет правила, которые нужно использовать при захвате одного или более ссылочного типа в теле замыкания. Что же касается циклов сильных связей между двумя экземплярами классов, то вы объявляете каждую захваченную ссылку как слабую или бесхозную (`weak` или `unowned`), вместо того, чтобы оставлять ее сильной (`strong`). Правильный выбор между слабой или бесхозной ссылками зависит от взаимоотношений между различными частями вашего кода.

Заметка

Swift требует от вас написания `self.someProperty` или `self.someMethod` (**вместо** `omeProperty`, `someMethod`), каждый раз, когда вы обращаетесь к члену свойства `self` внутри замыкания. Это помогает вам не забыть, что возможен случай случайного захвата `self`.

Определение списка захвата

Каждый элемент в списке захвата является парой ключевого слова `weak` или `unowned` и ссылки на экземпляр класса (например, `self` или `someInstance`). Эти пары вписываются в квадратные скобки и разделяются между собой запятыми.

Размещайте список захвата перед списком параметров замыкания и его возвращаемым типом:

```
lazy var someClosure: (Int, String) -> String
= {
    [unowned self] (index: Int,
stringToProcess: String) -> String in
    // тело замыкания
}
```

Если у замыкания нет списка параметров или возвращаемого типа, так как они могут быть выведены из контекста, то разместите список захвата в самом начале замыкания, перед словом `in`:

```
lazy var someClosure: () -> String = {
    [unowned self] in
    // тело замыкания
}
```

Слабые (weak) или бесхозные (unowned) ссылки

Определите список захвата в замыкании как бесхозную ссылку в том случае, когда замыкание и экземпляр, который оно захватывает, всегда будут ссылаться друг на друга, то они всегда будут освобождаться в одно и то же время.

Наоборот, определите список захвата в качестве слабой ссылки, когда захваченная ссылка может стать `nil` в какой-либо момент в будущем. Слабые ссылки всегда опционального типа и автоматически становятся `nil`, когда экземпляр, на который они ссылаются, освобождается. Это позволяет вам проверять их существование внутри тела замыкания.

Заметка

Если захваченная ссылка никогда не будет `nil`, то она должна быть всегда захвачена как `unowned` ссылка, а не `weak` ссылка.

Бесхозная ссылка является подходящим методом захвата для предотвращения существования цикла сильных ссылок в нашем примере с `HTMLElement`. Вот как можно записать класс `HTMLElement`, чтобы избежать цикла:

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return
            "<\(self.name)>\(text)</\(\(self.name))>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        println("\(name) освобождается")
    }

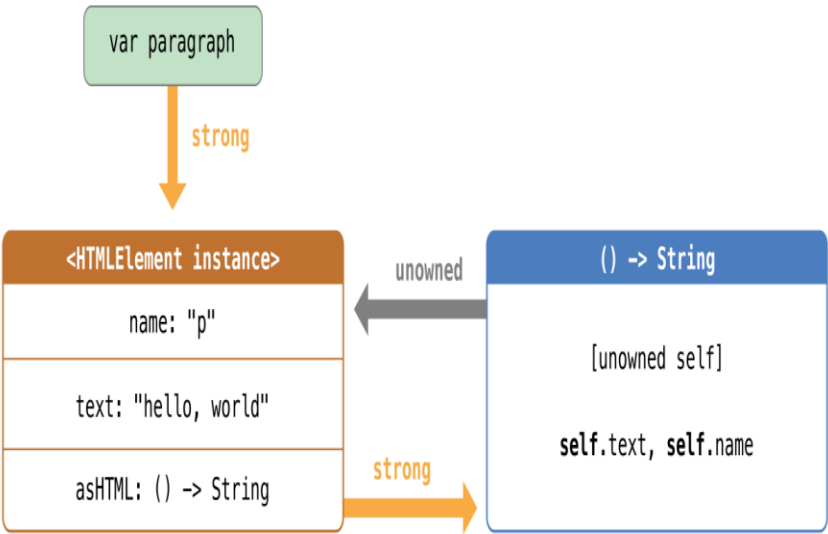
}
```

Эта реализация `HTMLElement` идентична предыдущей реализации, кроме дополнения списка захвата внутри замыкания `asHTML`. В этом случае список захвата `[unowned self]`, который означает: “захватить `self` как `unowned` ссылку, вместо `strong`”.

Вы можете создать и вывести экземпляр `HTMLElement` как и раньше:

```
var paragraph: HTMLElement? =
    HTMLElement(name: "p", text: "hello, world")
println(paragraph!.asHTML())
// выводит "<p>hello, world</p>"
```

Вот как теперь выглядят связи:



В этот раз захват `self` является бесхозной ссылкой и уже не поддерживает сильной связи с экземпляром `HTMLElement`, которого он захватил. Если вы установите сильную ссылку от переменной `paragraph` на значение `nil`, то экземпляр `HTMLElement` будет освобожден, что можно определить по выводимому сообщению в примере ниже:

```
paragraph = nil  
// выводит "p освобождается"
```

Опциональная

последовательность

Опциональная последовательность (optional chaining) - процесс запросов и вызовов свойств, методов, сабскриптов (индексов) у опционала, который может быть `nil`. Если опционал содержит какое-либо значение, то вызов свойства, метода или сабскрипта успешен, и наоборот, если опционал равен `nil`, то вызов свойства, метода или сабскрипта возвращает `nil`. Множественные запросы могут быть соединены вместе, и вся цепочка этих запросов не срабатывает, если хотя бы один запрос равен `nil`.

Заметка

Опциональная последовательность в Swift - аналог сообщению `nil` в Objective-C, но только она работает со всеми типами, и может быть проверена на успех или неудачу.

Опциональная последовательность как альтернатива принудительному развертыванию

Вы обозначаете опциональную последовательность, когда ставите вопросительный знак (?) после опционального значения, на которое вы хотите вызвать свойство, метод или индекс, если опционал не `nil`. Это очень похоже на установку восклицательного знака (!) после опционального значения для принудительного развертывания его значения. Основное отличие в том, что опциональная последовательность не выполняется, если опционал равен `nil`, в то время как принудительное развертывание приводит к runtime ошибке, когда опционал равен `nil`.

Факт того, что опциональная последовательность может быть вызвана и на значение `nil`, отражается в том, что результатом работы опциональной последовательности всегда является опциональная величина, даже в том случае, если свойство, метод или сабскрипт, к которым вы обращаетесь, возвращает неопциональное значение. Вы можете использовать это значение опционального возврата для проверки успеха (если возвращенный опционал содержит значение) или неудачи (если возвращенное значение опционала `nil`).

В частности, результат вызова опциональной последовательности того же типа, что и тип ожидаемого возвращаемого значения, только в завернутом в опционал

виде. Свойство, которое обычно возвращало `Int`, вернет `Int?`, когда обращаются к нему через опциональную последовательность.

Следующие несколько фрагментов кода покажут вам как отличаются опциональная последовательность от принудительного разворачивания, и как она позволяет вам проверить значение на успех.

Первые два класса `Person`, `Residence` определены как:

```
class Person {  
    var residence: Residence?  
}  
  
class Residence {  
    var numberOfRooms = 1  
}
```

Экземпляры `Residence` имеют единственное свойство `numberOfRooms` типа `Int`, со значением по умолчанию 1. Экземпляры `Person` имеют опциональное свойство `residence` типа `Residence?`.

Если вы создаете новый экземпляр `Person`, то его свойство `residence` по умолчанию имеет значение `nil`, в силу того, что оно является опционалом. В коде ниже `john` имеет свойство `residence`, значение которого `nil`:

```
let john = Person()
```

Если вы попытаетесь получить доступ к свойству `numberOfRooms` свойства `residence` экземпляра `Person`, поставив восклицательный знак после `residence`, для принудительного разворачивания, то вы получите runtime

ошибку, потому что `residence` не имеет значения для разворачивания:

```
let roomCount = john.residence!.numberOfRooms
// ошибка runtime
```

Код, представленный выше, срабатывает успешно, если `john.residence` имеет не `nil` значение и устанавливает корректное значение типа `Int` для `roomCount`. Однако этот код всегда всегда будет выдавать ошибку `runtime`, когда `residence` равен `nil`, что указано выше.

Опциональная последовательность предоставляет альтернативный способ получить доступ к значению `numberOfRooms`. Для использования опциональной последовательности используйте вопросительный знак, на месте восклицательного знака:

```
if let roomCount =
john.residence?.numberOfRooms {
    println("John's residence has \(roomCount)
room(s).")
} else {
    println("Unable to retrieve the number of
rooms.")
}
// выводит "Unable to retrieve the number of
rooms."
```

Это сообщает Swift “сцепиться” с опциональным свойством `residence` и получить значение `numberOfRooms`, если `residence` существует.

Так как попытка доступа к `numberOfRooms` имеет потенциальную возможность претерпеть неудачу, то опциональная последовательность возвращает значение типа `Int?` или “опциональный `Int`”. Когда `residence` равен `nil`, в примере выше, этот опциональный `Int` также будет `nil`, для отображения того факта, что было невозможно получить доступ к `numberOfRooms`.

Обратите внимание, что это верно даже если `numberOfRooms` неопциональный `Int`. Факт того, что запрос был через опциональную последовательность означает, что вызов `numberOfRooms` будет всегда возвращать `Int?`, вместо `Int`.

Вы можете присвоить экземпляр `Residence` в `john.residence`, так что оно больше не будет являться значением `nil`.

```
john.residence = Residence()
```

Теперь `john.residence` содержит экземпляр `Residence`, а не `nil`. Если вы попытаетесь получить доступ к `numberOfRooms` с той же последовательностью опционала, что и раньше, то теперь она вернет нам `Int?`, который содержит значение по умолчанию `numberOfRooms` равное 1:

```
if let roomCount =
john.residence?.numberOfRooms {
    println("John's residence has \ (roomCount)
room(s).")
} else {
    println("Unable to retrieve the number of
rooms.")
}
// выводит "John's residence has 1 room(s)."
```

Определение классовых моделей для опциональной последовательности

Вы можете использовать опциональную последовательность для вызовов свойств, методов, сабскриптов, которые находятся более чем на один уровень глубже. Это позволяет вам пробираться через подсвойства, внутри сложных моделей вложенных типов, и проверять возможность доступа свойств, методов и сабскриптов этих подсвойств.

Фрагмент кода ниже определяет четыре модели классов для использования в нескольких следующих примеров, включая примеры с многоуровневой опциональной последовательностью. Эти классы расширяют модели `Person`, `Residence` приведенные ранее, добавляя классы `Room`, `Address` со свойствами, методами и сабскриптами.

Класс `Person` объявляется так же как и раньше:

```
class Person {  
    var residence: Residence?  
}
```

Класс `Residence` стал намного сложнее, чем был раньше. В этот раз класс `Residence` определяет переменное свойство `rooms`, которое инициализировано пустым массивом `[Room]`:

```

class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        println("The number of rooms is
\ (numberOfRooms) ")
    }
    var address: Address?
}

```

Из-за того, что эта версия `Residence` хранит массив экземпляров `Room`, его свойство `numberOfRooms` реализовано как вычисляемое, а не как хранимое свойство. Вычисляемое свойство `numberOfRooms` просто возвращает значение свойства `count` массива `rooms`.

В качестве сокращенного варианта доступа к массиву `rooms`, эта версия класса `Residence` предлагает сабскрипт (доступный как для чтения, так и для записи), который предоставляет доступ к комнате под требуемым индексом в массиве `rooms`.

Эта версия `Residence` так же обеспечивает метод `printNumberOfRooms`, который просто выводит на экран количество комнат в резиденции.

И наконец, `Residence` определяет опциональное свойство `address` типа `Address?`. Тип класса `Address` для этого свойства определен ниже.

Класс `Room` используется для массива `rooms`, в качестве простого класса с одним свойством `name` и инициализатором, в котором устанавливается значение свойства `name` как подходящее имя комнаты:

```
class Room {  
    let name: String  
    init(name: String) { self.name = name }  
}
```

Последний класс в этой модели `Address`. Этот класс имеет три опциональных свойства типа `String?`. Первые два свойства `buildingName`, `buildingNumber` являются альтернативным вариантом определения конкретного здания как части адреса. Третье свойство `street` используется для названия улицы, для этого адреса:

```
class Address {  
    var buildingName: String?  
    var buildingNumber: String?  
    var street: String?  
    func buildingIdentifier() -> String? {  
        if buildingName != nil {  
            return buildingName  
        } else if buildingNumber != nil {  
            return buildingNumber  
        } else {  
            return nil  
        }  
    }  
}
```

Класс `Address` также предоставляет метод `buildingIdentifier`, который возвращает `String?`. Этот метод проверяет свойства `buildingName`, `buildingNumber` и возвращает `buildingName`, если у него есть значение или возвращает `buildingNumber`, если у него есть значение или `nil`, если ни у одного из свойств нет значения.

Доступ к свойствам через опциональную последовательность

Как было показано в разделе “[Опциональная последовательность как альтернатива принудительному разворачиванию](#)”, вы можете использовать опциональную последовательность для доступа к свойству опционального значения и проверить результат доступа к этому свойству на успешность

Используйте классы, определенные ранее, для создания нового экземпляра `Person` и попробуйте получить доступ к свойству `numberOfRooms`, как вы делали ранее:

```
let john = Person()
if let roomCount =
john.residence?.numberOfRooms {
    println("John's residence has \ (roomCount)
room(s).")
} else {
    println("Unable to retrieve the number of
rooms.")
}
// выводит "Unable to retrieve the number of
rooms."
```

Так как `john.residence` равен `nil`, этот вызов опциональной последовательности не будет успешен как и ранее.

Вы можете попробовать установить значение свойства через опциональную последовательность:

```
let someAddress = Address()  
someAddress.buildingNumber = "29"  
someAddress.street = "Acacia Road"  
john.residence?.address = someAddress
```

В этом примере попытка установить значение свойству `address` опциональному свойству `residence?` провалится, так как `john.residence` все еще `nil`.

Вызов методов через опциональную последовательность

Вы можете использовать опциональную последовательность для вызова метода опциональной величины, и проверить сам вызов метода на успешность. Вы можете сделать это, даже если этот метод не возвращает значения.

Метод `printNumberOfRooms` класса `Residence` выводит текущее значение `numberOfRooms`. Вот как выглядит этот метод:

```
func printNumberOfRooms() {  
    println("The number of rooms is  
    \ (numberOfRooms) ")  
}
```

Этот метод не определяет возвращаемого значения. Однако функции и методы без возвращаемого значения имеют неявный возвращаемый тип `Void`, как было описано в главе [“Параметры функции и возвращаемые значения”](#). Это означает, что они возвращают значение `()` или просто пустой кортеж.

Если вы вызовете этот метод на опциональном значении в опциональной последовательности, то он вернет тип не `Void`, а `Void?`, потому что возвращаемые значения всегда опционального типа, когда они вызываются через опциональную последовательность. Это позволяет вам использовать конструкцию `if` для проверки на возможность вызова метода `printNumberOfRooms`, даже если метод сам не определяет возвращаемого значения. Сравните возвращаемое значение от вызова `printNumberOfRooms` и `nil`, для того чтобы увидеть, что вызов метода прошел успешно:

```
if john.residence?.printNumberOfRooms() != nil
{
    println("It was possible to print the
number of rooms.")
} else {
    println("It was not possible to print the
number of rooms.")
}
// выводит "It was not possible to print the
number of rooms."
```

То же самое верно, если вы попытаетесь установить свойство через опциональную последовательность. Пример выше в [“Доступ к свойствам через опциональную последовательность”](#) пытается установить значение `address` в `john.residence`, хотя свойство `residence` равно `nil`. Любая попытка установить свойство через опциональную последовательность возвращает значение `Void?`, которое

позволяет вам сравнивать его с `nil`, для того, чтобы увидеть логический результат установки значения свойству (успех, провал):

```
if (john.residence?.address = someAddress) !=
nil {
    println("It was possible to set the
address.")
} else {
    println("It was not possible to set the
address.")
}
// выводит "It was not possible to set the
address."
```

Доступ к индексам через опциональную последовательность

Вы можете использовать опциональную последовательность для того, чтобы попробовать получить и установить значения из индекса опционального значения, и проверить успешность выполнения вызова сабскрипта.

Заметка

Когда вы получаете доступ к опциональному значению через опциональную последовательность, вы размещаете вопросительный знак до скобок сабскрипта (индекса), а не после. Вопросительный знак опциональной последовательности следует сразу после части выражения, которая является опционалом.

Пример ниже пробует получить значение имени первой комнаты в массиве `rooms` свойства `john.residence`, используя сабскрипт, определенный в классе `Residence`. Из-за того, что `john.residence` является `nil`, то вызов сабскрипта проваливается:

```
if let firstRoomName = john.residence?[0].name
{
    println("The first room name is
\ (firstRoomName) .")
} else {
    println("Unable to retrieve the first room
name.")
}
// выводит "Unable to retrieve the first room
name."
```

Вызов вопросительного знака опциональной последовательности в этом сабскрипте идет сразу после `john.residence`, но до скобок сабскрипта, потому что `john.residence` является опциональным значением, на которое применяется опциональная последовательность.

Аналогично вы можете попробовать установить новое значение через сабскрипт с помощью опциональной последовательности:

```
john.residence?[0] = Room(name: "Bathroom")
```

Это попытка установки значения через сабскрипт так же не срабатывает, так как `residence` все еще `nil`.

Если вы создадите и присвоите действительное значение экземпляру `Residence`, при помощи одного или нескольких экземпляров `Room` в массиве `rooms`, то вы сможете

использовать сабскрипт на экземпляре `residence` для того, чтобы получить доступ к массиву `rooms` через опциональную последовательность:

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name:
"Гостиная"))
johnsHouse.rooms.append(Room(name: "Кухня"))
john.residence = johnsHouse
if let firstRoomName = john.residence?[0].name
{
    println("Название первой комнаты
\(firstRoomName).")
} else {
    println("Никак не получить название
первой комнаты.")
}
// выводит "Название первой комнаты Гостиная."
```

Получение доступа к сабскрипту (индексу) опционального типа

Если сабскрипт возвращает значение опционального типа, например ключ словаря типа `Dictionary` в Swift, то мы должны поставить вопросительный знак после закрывающей скобки сабскрипта, для присоединения его опционального возвращаемого значения:

```
var testScores = ["Dave": [86, 82, 84], "Bev":
[79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0]++
testScores["Brian"]?[0] = 72
// массив "Dave" теперь имеет вид [91, 82,
84], массив "Bev" - [80, 94, 81]
```

Пример выше определяет словарь `testScores`, который содержит две пары ключ/значение, которые соединяют ключ типа `String` со значением типа `[Int]`. Пример использует опциональную последовательность для установки значения первого элемента ключа `"Dave"` равным 91, для увеличения первого элемента массива под ключом `"Bev"` на 1 и для попытки установить первое значение несуществующего массива, соответствующего ключу `"Brian"` равным 72. Первые два вызова завершились успешно, потому что их ключи находятся в `testScores`. Третий вызов завершился неудачей, так как такого ключа как `"Brian"` в словаре не оказалось.

Соединение нескольких уровней ОП

Вы можете соединить несколько уровней опциональных последовательностей вместе для того, чтобы пробраться до свойств, методов, сабскриптов, которые находятся глубже в модели. Однако многоуровневые опциональные последовательности не добавляют новых уровней опциональности к возвращаемым значениям:

Скажем другими словами:

- Если тип, который вы пытаетесь получить не опциональный, то он станет опциональным из-за опциональной последовательности.
- Если тип, который вы пытаетесь получить, уже опциональный, то более опциональным он уже не станет, даже по причине опциональной последовательности.

Таким образом:

- Если вы пытаетесь получить значение типа `Int` через опциональную последовательность, то получите `Int?`, и это не будет зависеть от того, сколько уровней в опциональной последовательности задействовано.
- Аналогично, если вы попытаетесь получить значение типа `Int?` через опциональную последовательность, то вы получите `Int?`, что опять таки не зависит от количества уровней, которые задействованы в опциональной последовательности.

Пример ниже пробует получить доступ к свойству `street` свойства `address` свойства `residence` экземпляра `john`. Здесь задействовано два уровня опциональной последовательности для того, чтобы соединить свойства `residence` и `address`, оба из которых опционального типа:

```
if let johnsStreet =  
john.residence?.address?.street {  
    println("John's street name is  
    \ (johnsStreet).")  
} else {  
    println("Unable to retrieve the address.")  
}  
// выводит "Unable to retrieve the address."
```

Значение `john.residence` на данный момент содержит корректный экземпляр класса `Residence`. Однако значение `john.residence.address` равно `nil`. Из-за этого вызов `john.residence?.address?.street` проваливается.

Обратите внимание, что в примере выше вы пытаетесь получить значение свойства `street`. Тип этого свойства `String?`. Возвращаемое значение `john.residence?.address?.street` также `String?`, даже если два уровня опциональной последовательности применены в дополнение к опциональному типу самого свойства.

Если вы установите фактический экземпляр класса `Address` как значение для `john.residence.address` и установите фактическое значение для свойства `street`, то вы можете получить доступ к значению свойства `street` через многоуровневую опциональную последовательность (цепочку):

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence!.address = johnsAddress
if let johnsStreet =
john.residence?.address?.street {
    println("John's street name is
\ (johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// выводит "John's street name is Laurel
Street."
```

Обратите внимание на использование восклицательного знака внутри `john.residence!.address`.

Свойство `john.residence` имеет опциональный тип, так что вам нужно развернуть его фактическое значение при помощи восклицательного знака до того, как вы попытаетесь получить доступ к свойству `address` свойства `residence`.

Прикрепление методов к ОП с опциональными возвращаемыми значениями

Предыдущие пример показал, как можно получить значение свойства опционального типа через опциональную последовательность. Вы так же можете использовать опциональную последовательность для вызова метода, который возвращает значение опционального типа, а затем к этой опциональной последовательности может прикрепить и возвращаемое значение самого метода, если это нужно.

Пример ниже вызывает метод `buildingIdentifier` класса `Address` через опциональную последовательность. Этот метод возвращает значение типа `String?`. Как было описано ранее, что возвращаемый тип этого метода после опциональной последовательности так же будет `String?`:

```
if let buildingIdentifier =  
john.residence?.address?.buildingIdentifier()  
{  
    println("John's building identifier is  
    \ (buildingIdentifier).")  
}  
// выводит "John's building identifier is The  
Larches."
```

Если вы хотите продолжить свою опциональную привязку и на возвращаемое значение метода, то разместите вопросительный знак после круглых скобок самого метода:

```
if let beginsWithThe =
john.residence?.address?.buildingIdentifier()?
.hasPrefix("The") {
    if beginsWithThe {
        println("John's building
identifier begins with \"The\".")
    } else {
        println("John's building
identifier does not begin with \"The\".")
    }
}
// выводит "John's building identifier begins
with "The"."
```

Заметка

В примере выше вы разместили вопросительный знак опциональной привязки после круглых скобок метода, потому что опциональная величина, которую вы присоединяете к последовательности, является возвращаемой величиной метода `buildingIdentifier`, а не самим методом `buildingIdentifier`.

Приведение типов

Приведение типов - это способ проверить тип экземпляра и/или способ обращения к экземпляру так, как если бы он был экземпляром суперкласса или подкласса откуда-либо из своей собственной классовой иерархии.

Приведение типов в Swift реализуется с помощью операторов `is` и `as`. Эти два оператора предоставляют простой и выразительный способ проверки типа значения или преобразование значения к другому типу.

Вы так же можете использовать приведение типов для проверки соответствия типа протоколу.

Определение классовой иерархии для приведения типов

Вы можете использовать приведение типов с иерархией классов и подклассов, чтобы проверить тип конкретного экземпляра класса и преобразовать тип этого экземпляра в тип другого класса в той же иерархии. Следующих три фрагмента кода определяют иерархию классов и массив, который содержит экземпляры этих классов, для использования в примере приведения типов.

Первый кусок кода определяет новый базовый класс `MediaItem`, Этот класс предоставляет базовую функциональность для любого вида элемента, который

появляется в цифровой медиа библиотеке. А именно, он определяет свойство `name` типа `String` и инициализатор `init name`. (В этом примере считается, что все фильмы, песни или другие медиа штуковины имеют свойство `name` или попросту имя.)

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

Следующий фрагмент определяет два подкласса класса `MediaItem`. Первый подкласс - `Movie`, он инкапсулирует дополнительную информацию о фильмах. Он добавляет свойство `director` поверх базового класса `MediaItem` с соответствующим инициализатором. Второй подкласс - `Song`. Этот подкласс добавляет свойство `artist` и инициализатор поверх базового класса:

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

Последний отрывок кода создает неизменяемый массив `library`, который содержит два экземпляра `Movie` и три экземпляра `Song`. Тип `library` выведен во время инициализации массива литералом массива. Механизм проверки типов Swift делает вывод, что `Movie`, `Song` имеют общий суперкласс `MediaItem`, так что тип массива `library` становится `[MediaItem]`:

```
let library = [  
    Movie(name: "Casablanca", director:  
"Michael Curtiz"),  
    Song(name: "Blue Suede Shoes", artist:  
"Elvis Presley"),  
    Movie(name: "Citizen Kane", director:  
"Orson Welles"),  
    Song(name: "The One And Only", artist:  
"Chesney Hawkes"),  
    Song(name: "Never Gonna Give You Up",  
artist: "Rick Astley")  
]  
// тип "library" выведен как [MediaItem]
```

Элементы, которые хранятся в `library` все еще экземпляры `Movie` и `Song` на самом деле. Однако, если вы переберете элементы массива, то они все будут одного типа `MediaItem`, а не `Movie` или `Song`. Для того чтобы работать с ними как с исходными типами, вам нужно проверить их типы или привести к другому типу, как указано далее.

Проверка типов

Используйте оператор проверки типа для проверки того, соответствует ли тип экземпляра типом какого-то определенного подкласса. Оператор проверки типа возвращает `true`, если экземпляр имеет тип конкретного подкласса, `false`, если нет.

Пример ниже определяет две переменные `movieCount` и `songCount`, которые считают число экземпляров `Movie` и экземпляров `Song` в массиве `library`:

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        ++movieCount
    } else if item is Song {
        ++songCount
    }
}

println("В Media библиотеке содержится
\u(movieCount) фильма и \u(songCount) песни")
// выводит "В Media библиотеке содержится 2
фильма и 3 песни"
```

Пример перебирает все элементы массива `library`. На каждую итерацию цикла `for-in` константа `item` инициализируется следующим значением типа `MediaItem` из массива `library`.

Выражение `item is Movie` возвращает `true`, в том случае, если текущий `item` типа `MediaItem` **является** экземпляром `Movie`, и это выражение возвращает `false`, если не является экземпляром `Movie`. Аналогично `item is Song` проверяет является ли `item` экземпляром `Song`. В конце цикла `for-in` значения `movieCount` и `songCount` содержат количество экземпляров `MediaItem` каждого типа.

Понижающее приведения

Константа или переменная определенного класса может фактически ссылаться на экземпляр подкласса. Чему вы будете верить в данном случае, вы можете попробовать привести тип к типу подкласса при помощи оператора понижающего приведения (`as!`).

Из-за того, что понижающее приведение может провалиться, оператор приведения имеет две формы. Опциональная форма (`as?`), которая возвращает опциональное значение типа, к которому вы пытаетесь привести. И принудительная форма (`as!`), которая принимает попытки понижающего приведения и принудительного разворачивания результата в рамках одного составного действия.

Используйте опциональную форму оператора понижающего приведения (`as?`), когда вы не уверены, что ваше понижающее приведение выполнится успешно. В этой форме оператор всегда будет возвращать опциональное значение, и значение будет `nil`, если понижающее приведение будет не выполнимо. Так же это позволяет вам проверить успешность понижающего приведения типа.

Используйте принудительную форму оператора понижающего приведения (`as!`), но только в тех случаях,

когда вы точно уверены, что понижающее приведение будет выполнено успешно. Эта форма оператора вызовет runtime ошибку, если вы попытаетесь таким образом привести к некорректному типу класса.

Пример ниже перебирает элементы `MediaItem` в массиве `library` и выводит соответствующее описание для каждого элемента. Чтобы сделать это, ему нужно получить доступ к каждому элементу как `Movie` или `Song`, а не просто как к `MediaItem`. Это необходимо для того, чтобы был доступ к свойствам `director`, `artist`, которые пригодятся нам в описании.

В этом примере каждый элемент массива может быть либо `Movie`, либо `Song`. Вы не знаете наперед какой класс вам нужно использовать для каждого элемента, так что логично будет использовать опциональную форму оператора понижающего приведения (`as?`) для проверки возможности понижающего приведения к конкретному подклассу конкретного элемента массива:

```
for item in library {
    if let movie = item as? Movie {
        println("Movie: '\(movie.name)', dir.
\ (movie.director)")
    } else if let song = item as? Song {
        println("Song: '\(song.name)', by
\ (song.artist)")
    }
}
// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick
Astley
```

Пример начинается с попытки понижающего приведения текущего элемента `item` в качестве `Movie`. Так как `item` является экземпляром `MediaItem`, то этот элемент может быть `Movie`, но он и так же может быть и `Song` или даже просто базовым `MediaItem`. Из-за этой неопределенности мы и используем опциональную форму оператора (`as?`). Результат выражения `item as? Movie` является тип `Movie?` или “опциональный `Movie`”.

Понижающее приведение к `Movie` проваливается, когда оно применимо к экземплярам `Song` в массиве `library`. Объединив это все, мы получаем, что пример выше использует опциональную привязку для проверки наличия значения у опционального `Movie` (то есть, чтобы выяснить успешность проведенной операции). Опциональная привязка записана выражением вида “`if let movie = item as? Movie`”, что может быть прочитано так:

“Пробуем получить доступ к `item` в качестве `Movie`. Если доступ успешен, то присвоим временную константу `movie` значению, которое мы получили из опционального `Movie`.”

Если понижающее приведение прошло успешно, то свойства `movie` используются в качестве вывода описания для экземпляра `Movie`, включая имя `director`. Аналогичный принцип используется при проверке экземпляров `Song` и при выводе описания (включая имя `artist`), как только находится элемент `Song` в массиве `library`.

Заметка

Приведение не изменяет экземпляра или его значений. Первоначальный экземпляр остается тем же. Просто после приведения типа с экземпляром можно обращаться (и использовать свойства) именно так как с тем типом, к которому его привели.

Приведение типов для Any и AnyObject

Swift предлагает два типа псевдонима типа для работы с неопределенными типами:

- `AnyObject` может отобразить экземпляр любого класса.
- `Any` может отобразить экземпляр любого типа, включая функциональные типы.

Заметка

Используйте `Any` и `AnyObject` только тогда, когда вам явно нужно поведение и особенности, которые они предоставляют. Всегда лучше быть конкретным насчет типов, с которыми вы ожидаете работать в вашем коде.

AnyObject

Когда вы работаете с Cocoa API очень часто вы можете получить массив типа `[AnyObject]` или “массив значений типа `any object`”. Это из-за того, что Objective-C не имеет массивов с явно указанными типами. Однако часто вы можете быть уверены в типе элементов массива из той информации, которую вы знаете об этом API.

В этих ситуациях вы можете использовать версию принудительного оператора (`as!`) для понижающего приведения типа до класса `AnyObject`, без необходимости принудительного развертывания.

Пример ниже определяет массив типа `[AnyObject]`, который содержит три экземпляра класса `Movie`:

```
let someObjects: [AnyObject] = [  
    Movie(name: "2001: A Space Odyssey",  
director: "Stanley Kubrick"),  
    Movie(name: "Moon", director: "Duncan  
Jones"),  
    Movie(name: "Alien", director: "Ridley  
Scott")  
]
```

Так как мы знаем, что этот массив может содержать экземпляры только класса `Movie`, вы используете понижающее приведение и разворачиваете напрямую неопциональную версию `Movie` с помощью оператора (`as!`) в принуждающей форме:

```
for object in someObjects {  
    let movie = object as! Movie  
    println("Movie: '\(movie.name)', dir.  
\(movie.director)")  
}  
// Movie: '2001: A Space Odyssey', dir.  
Stanley Kubrick  
// Movie: 'Moon', dir. Duncan Jones  
// Movie: 'Alien', dir. Ridley Scott
```

Для еще более короткой формы этого цикла используем понижающее приведение типа на сам массив к типу `[Movie]`, а не на каждый его элемент:

```
for movie in someObjects as! [Movie] {  
    println("Movie: '\ (movie.name) ', dir.  
    \ (movie.director)")  
}  
// Movie: '2001: A Space Odyssey', dir.  
Stanley Kubrick  
// Movie: 'Moon', dir. Duncan Jones  
// Movie: 'Alien', dir. Ridley Scott
```

Any

Ниже приведен пример использования `Any` для работы с различными типами, включая функциональные типы и внеклассовые типы. Пример создает массив `things`, который может хранить тип `Any`:

```
var things = [Any]()  
things.append(0)  
things.append(0.0)  
things.append(42)  
things.append(3.14159)  
things.append("hello")  
things.append((3.0, 5.0))  
things.append(Movie(name: "Ghostbusters",  
director: "Ivan Reitman"))  
things.append({ (name: String) -> String in  
"Hello, \ (name) " })
```

Массив `things` содержит для значения типа `Int`, два значения типа `Double`, значение типа `String`, кортеж типа `(Double, Double)`, кино "Ghostbusters" и замыкание типа `(String) -> (String)`.

Вы можете использовать операторы `is` и `as` в случаях (`case`) конструкции `switch` для определения типа константы или переменной, когда известно только то, что она принадлежит типу `Any` или `AnyObject`. Пример ниже перебирает элементы в массиве `things` и запрашивает тип у каждого элемента с помощью конструкции `switch`. Несколько случаев конструкции `switch` привязывают их совпавшие значения к константе определенного типа, для того, чтобы потом можно было вывести значение на экран:

```
for thing in things {
    switch thing {
        case 0 as Int:
            println("zero as an Int")
        case 0 as Double:
            println("zero as a Double")
        case let someInt as Int:
            println("an integer value of
\ (someInt) ")
            case let someDouble as Double where
someDouble > 0:
                println("a positive double value of
\ (someDouble) ")
            case is Double:
                println("some other double value that
I don't want to print")
            case let someString as String:
                println("a string value of
\" \ (someString) \")
            case let (x, y) as (Double, Double):
                println("an (x, y) point at \ (x),
\ (y) ")
```

```

    case let movie as Movie:
        println("a movie called
'\(movie.name)', dir. \(movie.director)")
        case let stringConverter as String ->
String:
            println(stringConverter("Michael"))
        default:
            println("something else")
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called 'Ghostbusters', dir. Ivan
Reitman
// Hello, Michael

```

Заметка

Случаи в конструкции `switch` используют принуждающую версию оператора (`as`, а не `as?`) для проверки определенного типа. Эта проверка всегда безопасна внутри контекста случая конструкции `switch`.

Вложенные типы

Зачастую перечисления создаются для дополнительной поддержки функциональности определенного класса. Аналогично может быть полезным создание вспомогательных классов или структур предназначенных для контекста более сложного типа. Для достижения этой цели Swift предлагает вам определить *вложенные типы*, в которые вы вкладываете вспомогательные перечисления, классы и структуры, внутри определения типа, которые они поддерживают.

Чтобы вложить тип в другой тип, вам нужно написать свое определение во внешних фигурных скобках типа, который он поддерживает. Типы могут быть вложены на столько уровней, на сколько это необходимо.

Вложенные типы в действе

Пример ниже определяет структуру `BlackjackCard`, которая создает модель игральных карт игры `Blackjack`. Структура `BlackjackCard` содержит два вложенных перечисления типов `Suit`, `Rank`.

В игре `Blackjack` карта `Ace` (Туз) имеет значение либо один, либо одиннадцать. Это особенность отображается структурой `Values`, которая вложена внутрь перечисления `Rank`:

```

struct BlackjackCard {

    // вложенное перечисление Suit
    enum Suit: Character {
        case Spades = "♠", Hearts = "♥",
Diamonds = "♦", Clubs = "♣"
    }

    // вложенное перечисление Rank
    enum Rank: Int {
        case Two = 2, Three, Four, Five, Six,
Seven, Eight, Nine, Ten
        case Jack, Queen, King, Ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .Ace:
                return Values(first: 1,
second: 11)
            case .Jack, .Queen, .King:
                return Values(first: 10,
second: nil)
            default:
                return Values(first:
self.rawValue, second: nil)
            }
        }
    }

    // свойства и методы BlackjackCard
    let rank: Rank, suit: Suit
    var description: String {
        var output = "масть:
\"(suit.rawValue), "

```

```

        output += " значение:
\ (rank.values.first) "
        if let second = rank.values.second {
            output += " или \ (second) "
        }
        return output
    }
}

```

Перечисления `Suit` описывают четыре масти при помощи символа типа `Character` для их отображения.

Перечисление `Rank` вместе со значением `Int` описывает тринадцать возможных рангов карт, для отображения их номинальной стоимости. (Значение `Int` не используется для карт `Jack`, `Queen`, `King`, `Ace` или по-русски Валета, Королевы, Короля и Туза).

Как уже упоминалось ранее, структура `Rank` определяет вложенную внутри себя структуру `Values`. Эта структура инкапсулирует факт того, что большинство карт имеют одно значение, но Туз имеет два значения своей карты. Структура `Values` определяет два свойства для того, чтобы отобразить это.

- `first` типа `Int`
- `second` типа `Int?` или “опциональный `Int`”

`Rank` так же определяет вычисляемое свойство `values`, которое возвращает экземпляр структуры `Values`. Это вычисляемое свойство учитывает ранг карты и инициализирует новый экземпляр `Values` с соответствующими значениями, основываясь на своем ранге. Оно использует специальные значения для карт `Jack`, `Queen`, `King`, `Ace`. Для карт с цифрами, оно использует значение ранга типа `Int`.

Сама структура `BlackjackCard` имеет два свойства `rank`, `suit`. Она так же определяет вычисляемое свойство `description`, которое использует значение, которое храниться в `rank` и `suit`, для того, чтобы создать описание имени и значения карты. Свойство `description` использует опциональную привязку для проверки наличия второго значения, и если оно есть, то вставляет дополнительное описание для второго значения.

Из-за того что `BlackjackCard` является структурой без пользовательских инициализаторов, она имеет неявный почленный инициализатор, что описано в главе [“Почленная инициализация структурных типов”](#). Вы можете использовать этот инициализатор для инициализации новой константы `theAceOfSpades`:

```
let theAceOfSpades = BlackjackCard(rank: .Ace,
suit: .Spades)
println("theAceOfSpades:
\ (theAceOfSpades.description) ")
// выводит "theAceOfSpades: масть: ♠,
значение: 1 или 11"
```

Даже если `Rank` и `Suit` являются вложенными в `BlackjackCard`, их типы могут наследоваться из контекста, таким образом инициализация этого экземпляра может сослаться на члены перечисления по их именам (`.Ace` и `.Spades`). В примере выше свойство `description` корректно отображает то, что Туз (`Ace`) масти пики (`Spades`) имеет значение либо 1, либо 11.

Ссылка на вложенные типы

Для того, чтобы использовать вложенные типы снаружи определяющего их контекста, нужно поставить префикс имени типа, внутри которого он вложен, затем его имя:

```
let heartsSymbol =  
BlackjackCard.Suit.Hearts.rawValue  
// heartsSymbol равен "♥"
```

Приведенный выше пример позволяет именам `Suit` и `Rank` быть намеренно короткими, потому что их имена изначально подобраны под контекст, в котором они определены.

Расширения

Расширения добавляют новую функциональность существующему типу класса, структуры или перечисления. Это включает в себя возможность расширять типы, к исходным кодам которых у вас нет доступа (известно как ретроактивное моделирование). Расширения очень похожи на категории из Objective-C. (В отличие от категорий из Objective-C, расширения в Swift не имеют имен.)

Расширения в Swift могут:

- Добавлять вычисляемые свойства и вычисляемые `static` свойства
- Определять метода экземпляра и методы типа
- Предоставлять новые инициализаторы
- Определять сабскрипты
- Определять новые вложенные типы
- Обеспечить соответствие существующего типа протоколу

Заметка

Расширения могут добавлять новую функциональность типу, но они не могут переписать существующую функциональность.

Синтаксис расширений

Расширение объявляется с помощью ключевого слова `extension`:

```
extension SomeType {  
    // описываем новую функциональность для  
    типа SomeType  
}
```

Расширение может расширить существующий тип для того, чтобы он соответствовал одному или более протоколам. Там где это имеет место, имена протоколов записываются точно так же, как и в случае с классами или структурами:

```
extension SomeType: SomeProtocol,  
AnotherProtocol {  
    // реализация требования протокола тут  
}
```

Заметка

Если вы определяете расширение для добавления новой функциональности существующему типу, то новая функциональность будет доступна всем экземпляром этого типа, даже если они были созданы до того, как было определено расширение

Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа к существующим типам. В примере мы добавляем пять вычисляемых свойств экземпляра во встроенный тип `Double` языка Swift, для обеспечения работы данного типа с единицами длины:

```
extension Double {  
    var km: Double { return self * 1000.0 }  
    var m: Double { return self }  
    var cm: Double { return self / 100.0 }  
    var mm: Double { return self / 1000.0 }  
    var ft: Double { return self / 3.28084 }  
}  
let oneInch = 25.4.mm  
println("Один фут - это \(oneInch) метра")  
// выводит "Один фут - это 0.0254 метра"  
let threeFeet = 3.ft  
println("Три фута - это \(threeFeet) метра")  
// выводит "Три фута - это 0.914399970739201 метра"
```

Эти вычисляемые свойства объясняют, что тип `Double` должен считаться как конкретная единица измерения длины. Хотя они реализованы как вычисляемые свойства, имена этих свойств могут быть добавлены к литералу чисел с плавающей точкой через точечный синтаксис, как способ использовать значения литерала для проведения преобразований длины.

В этом примере, значение `1.0` типа `Double` отображает “один метр”. Это причина, по которой `m` возвращает `self`, что равно `1.m`, то есть посчитать `Double` от числа `1.0`.

Другие единицы требуют некоторых преобразований, чтобы выражать свое значение через метры. Один километр то же самое что и `1000` метров, так что `km` - вычисляемое свойство, которое умножает значение на `1000.0`, чтобы отобразить величину в метрах. По аналогии поступаем и с остальными свойствами, как например, с футом, которых в одном метре насчитывается `3.28024`, так что для выражения числа в метрах, нам нужно поделить его на `3.28024`.

Эти свойства являются вычисляемыми свойствами только для чтения, так что они могут быть выражены без ключевого слова `get`. Их возвращаемое значение является типом `Double` и может быть использовано в математических вычислениях, где поддерживается тип `Double`:

```
let aMarathon = 42.km + 195.m
println("Марафон имеет длину \ (aMarathon)
метров")
// выводит "Марафон имеет длину 42195.0
метров"
```

Заметка

Расширения могут добавлять новые вычисляемые свойства, но они не могут добавить хранимые свойства или наблюдателей свойства к уже существующим свойствам.

Инициализаторы в расширениях

Расширения могут добавить новые инициализаторы существующему типу. Это позволяет вам расширить другие типы для принятия ваших собственных типов в качестве параметров инициализатора, или для обеспечения дополнительных опций инициализации, которые не были включены как часть первоначальной реализации типа.

Расширения могут добавлять вспомогательные инициализаторы классу, но они не могут добавить новый назначенный инициализатор или деинициализатор классу. Назначенные инициализаторы и деинициализаторы должны всегда предоставляться реализацией исходного класса.

Заметка

Если вы используете расширения для того, чтобы добавить инициализатор к типу значений, который обеспечивает значения по умолчанию для всех своих хранимых свойств и не определяет какого-либо пользовательского инициализатора, то вы можете вызвать дефолтный инициализатор и почленный инициализатор для того типа значений изнутри инициализатора вашего расширения. Это не будет работать, если вы уже написали инициализатор как часть исходной реализации значения типа.

Пример ниже определяет структуру `Rect` для отображения геометрического прямоугольника. Пример так же определяет две вспомогательные структуры `Size` и `Point`, обе из которых предоставляют значения по умолчанию `0 . 0` для всех своих свойств:

```

struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}

```

Из-за того, что структура `Rect` предоставляет значения по умолчанию для всех своих свойств, она автоматически получает инициализатор по умолчанию и почленный инициализатор, что описано в главе “[Дефолтные инициализаторы](#)”. Эти инициализаторы могут быть использованы для создания экземпляров `Rect`:

```

let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x:
2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))

```

Вы можете расширить структуру `Rect` для предоставления дополнительного инициализатора, который принимает определенную точку и размер:

```

extension Rect {

    init(center: Point, size: Size) {
        let originX = center.x - (size.width /
2)
        let originY = center.y - (size.height
/ 2)
        self.init(origin: Point(x: originX, y:
originY), size: size)
    }
}

```

Этот новый инициализатор начинается с вычисления исходной точки, основываясь на значениях свойств `center` и `size`. Потом инициализатор вызывает почленный инициализатор структуры `init(origin: size:)`, который хранит новую исходную точку и размеры в соответствующих свойствах:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// исходная точка centerRect (2.5, 2.5) и его
размер (3.0, 3.0)
```

Заметка

Если вы предоставляете новый инициализатор вместе с расширением, вы все еще ответственны за то, что каждый экземпляр должен быть полностью инициализирован, когда инициализатор кончает свою работу.

Методы в расширениях

Расширения могут добавить новые методы экземпляра или методы типа к уже существующим типам. Следующий пример добавляет новый метод экземпляра `repetitions` к типу `Int`:

```
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0..
```

Метод `repetitions` принимает единственный аргумент типа `() -> ()`, который указывает на функцию, которая не принимает ни одного параметра и которая не возвращает значения.

После определения расширения вы можете вызвать метод `repetitions` на любом целом числе, чтобы выполнить определенное задание целое число раз:

```
3.repetitions({
    println("Вкусно!")
})
// Вкусно!
// Вкусно!
// Вкусно!
```

Используйте последующие замыкания для еще более краткого вызова:

```
3.repetitions {
    println("Ну очень вкусно!")
}
// Ну очень вкусно!
// Ну очень вкусно!
// Ну очень вкусно!
```

Изменяющиеся методы экземпляра

Методы экземпляров, добавленные в расширении так же могут менять и сам экземпляр. Методы структуры и перечисления, которые изменяют `self` или его свойства, должны быть отмечены как `mutating`.

Пример ниже добавляет новый изменяющийся (`mutating`) метод `square` для типа `Int`, который возводит в квадрат исходное значение:

```
extension Int {  
    mutating func square() {  
        self = self * self  
    }  
}  
var someInt = 3  
someInt.square()  
// теперь переменная someInt имеет значение 9
```

Сабскрипты а

расширениях

Расширения могут добавить новые сабскрипты к существующему типу. Этот пример добавляет сабскрипт целого числа во встроенный тип `Int` языка Swift. Этот сабскрипт `[n]` возвращает цифру, которая стоит на `n` позиции справа:

- `123456789[0]` возвращает 9
- `123456789[1]` возвращает 8

и так далее:

```
extension Int {
    subscript(var digitIndex: Int) -> Int {
        var decimalBase = 1
        while digitIndex > 0 {
            decimalBase *= 10
            --digitIndex
        }
        return (self / decimalBase) % 10
    }
}

746381295[0]
// возвращает 5
746381295[1]
// возвращает 9
746381295[2]
// возвращает 2
746381295[8]
// возвращает 7
```

Если значение `Int` не имеет достаточно количество цифр для требуемого индекса, то сабскрипт возвращает `0`, как если бы вместо этого числа стоял `0`:

```
746381295[9]
// возвращает 0, как если бы вы запросили вот
так:
0746381295[9]
```

Вложенные типы в расширениях

Расширения могут добавлять новые вложенные типы к существующим классам, структурам и перечислениям:

```
extension Int {
    enum Kind {
        case Negative, Zero, Positive
    }
    var kind: Kind {
        switch self {
            case 0:
                return .Zero
            case let x where x > 0:
                return .Positive
            default:
                return .Negative
        }
    }
}
```

Этот пример добавляет новое перечисление в тип `Int`. Это перечисление `Kind` описывает значение, которое отображает данное целое число. В частности оно определяет является ли число положительным, отрицательным или нулем.

Так же этот пример добавляет новое вычисляемое свойство `kind` к типу `Int`, которое возвращает соответствующий член перечисления `Kind` для этого числа.

Вложенное перечисление может быть использовано типом `Int`:

```
func printIntegerKinds(numbers: [Int]) {
    for number in numbers {
        switch number.kind {
            case .Negative:
                print("- ")
            case .Zero:
                print("0 ")
            case .Positive:
                print("+ ")
        }
    }
    print("\n")
}

printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Выводит "+ + - 0 - 0 +"
```

Эта функция `printIntegerKinds` принимает параметр в виде массива значений `Int`, затем перебирает по очереди все эти значения. Для каждого целого числа в массиве, функция смотрит на его вычисляемое свойство `kind` и выводит соответствующее описание.

Заметка

Как нам уже известно что `number.kind` имеет тип `Int.Kind`. Значит все значения членов `Int.Kind` могут быть записаны в короткой форме внутри конструкции `switch`, как `Negative`, а не `Int.Kind.Negative`.

Протоколы

Протокол определяет образец метода, свойства или другие требования, которые соответствуют определенному конкретному заданию или какой-то функциональности. Протокол фактически не предоставляет реализацию для любого из этих требований, он только описывает как реализация должна выглядеть. Протокол может быть *принят* классом, структурой или перечислением для обеспечения фактической реализации этих требований. Любой тип, который удовлетворяет требованиям протокола, имеет указание *соответствовать* этому протоколу.

Протоколы могут требовать, чтобы соответствующие им типы имели специфические свойства экземпляра, методы экземпляра, методы типа, операторы и сабскрипты.

Ситаксис протокола

Вы определяете протокол очень похоже на то, как вы определяете классы, структуры и перечисления:

```
protocol SomeProtocol {  
    // определение протокола..  
}
```

Пользовательские типы утверждают, что они принимают протокол, когда они помещают имя протокола после имени типа и разделяются с этим именем двоеточием, то есть указывают эти протоколы как часть их определения. После двоеточия вы можете указывать множество протоколов, перечисляя их имена через запятую:

```
struct SomeStructure: FirstProtocol,  
AnotherProtocol {  
    // определение конструкции...  
}
```

Если у класса есть суперкласс, то вписывайте имя суперкласса до списка протоколов, которые он принимает, так же разделите имя суперкласса и имя протокола запятой:

```
class SomeClass: SomeSuperclass,  
FirstProtocol, AnotherProtocol {  
    // определение класса...  
}
```

Требуемые свойства

Протокол может потребовать соответствующий ему тип предоставить свойство экземпляра или свойство типа конкретного типа и имени. Протокол не уточняет какое должно быть свойство, хранимое или вычисляемое, только лишь указывает на требование имени свойства и типа. Протокол так же уточняет должно ли быть доступным, или оно должно быть доступным и устанавливаемым.

Если протокол требует от свойства быть доступным и устанавливаемым, то это требование не может полностью быть удовлетворено константой или вычисляемым свойством только для чтения (read only). Если протокол только требует от свойства доступности (get), то такое требование может быть удовлетворено любым свойством, и это так же справедливо для устанавливаемого свойства, если это необходимо в вашем коде.

Требуемые свойства всегда объявляются как переменные свойства, с префиксом `var`. Свойства, значения которых вы

можете получить или изменить маркируются `{get set}` после объявления типа свойства, а свойства, значения которых мы можем только получить, но не изменить `{get}`.

```
protocol SomeProtocol {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
}
```

Перед требуемыми свойствами типов пишите префикс `class`, когда вы определяете их в протоколе. Это правило распространяется даже тогда, когда свойство имеет префикс `static`, когда мы реализуем его в структурах или перечислениях:

```
protocol AnotherProtocol {  
    class var someTypeProperty: Int { get set }  
}
```

Пример протокола с единственным требуемым свойством экземпляра:

```
protocol FullyNamed {  
    var fullName: String { get }  
}
```

Протокол `FullyNamed` требует соответствующий ему тип предоставить полное имя. Протокол больше не уточняет ничего, кроме того, что тип этого свойства должен быть в состоянии предоставить свое полное имя. Протокол утверждает, что любой тип `FullyNamed` должен иметь свойство `fullName`, значение которого может быть получено, и это значение должно быть типа `String`.

Ниже приведен пример структуры, которая принимает и полностью соответствует протоколу `FullyNamed`:

```
struct Person: FullyNamed {  
    var fullName: String  
}  
let john = Person(fullName: «John Appleseed»)  
// john.fullName равен «John Appleseed»
```

Этот пример определяет структуру `Person`, которая отображает персону с конкретным именем. Она утверждает, что она принимает протокол `FullyNamed`, в качестве первой строки собственного определения.

Каждый экземпляр `Person` имеет единственное свойство `fullName` типа `String`. Это удовлетворяет единственному требованию протокола `FullyNamed`, и это значит, что `Person` корректно соответствует протоколу. (Swift сообщает об ошибке во время компиляции, если требования протокола выполняются не полностью.)

Ниже представлен сложный класс, который так же принимает и соответствует **протоколу** `FullyNamed`:

```
class Starship: FullyNamed {
    var prefix: String?
    Var name: String
    init(name: String, prefix: String? = nil)
{
    self.name = name
    self.prefix = prefix
}
    var fullName: String {
        return (prefix != nil ? prefix! + « »
: «») + name
    }
}
var ncc1701 = Starship(name: «Enterprise»,
prefix: «USS»)
// ncc1701.fullName равен «USS Enterprise»
```

Класс реализует требуемое свойство `fullName`, в качестве вычисляемого свойства только для чтения (для космического корабля). Каждый экземпляр класса `Starship` хранит обязательный `name` и опциональный `prefix`.

Свойство `fullName` использует значение `prefix`, если оно существует и устанавливает его в начало `name`, чтобы получилось целое имя для космического корабля.

Требуемые методы

Протоколы могут требовать реализацию определенных методов экземпляра и методов типа, соответствующими типами протоколу. Эти методы написаны как часть определения протокола в точности в такой же форме как и методы экземпляра или типа, но только в них отсутствуют фигурные скобки или тело метода целиком. Вариативные параметры допускаются точно так же как и в обычных методах.

Заметка

Протоколы используют обычный синтаксис как и обычные методы, но они не могут определять значения по умолчанию (дефолтные значения) для параметров метода.

Так же как и в случае со требуемым свойством типа, так и при требуемых методах типа, вы всегда пишете префикс **class**, когда определяете метод в протоколе. Это верно, даже если требуемые методы типа начинаются с помощью ключевого слова **static**, когда они реализуются структурами или перечислениями:

```
protocol SomeProtocol {  
    class func someTypeMethod()  
}
```

Следующий пример определяет протокол с единственным требуемым методом экземпляра:

```
protocol RandomNumberGenerator {  
    func random() -> Double  
}
```

Этот протокол **RandomNumberGenerator** требует любой соответствующий ему тип иметь метод экземпляра `random`, который при вызове возвращает значение типа **Double**. Хотя это и не указано как часть протокола, но предполагается, что значение будет числом от **0.0** и до **1.0** (не включительно).

Протокол **RandomNumberGenerator** не делает никаких предположений по поводу того, как будет находиться это случайное число, он просто требует генератор предоставить стандартный способ генерации нового рандомного числа.

Ниже приведена реализация класса, который принимает и соответствует протоколу **RandomNumberGenerator**. Этот класс реализует алгоритм генератора псевдослучайных чисел, известный как алгоритм *линейного конгруэнтного генератора*:

```
class LinearCongruentialGenerator:
RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c) %
m)

        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
println("Случайное число:
(generator.random())")
// prints "Случайное число: 0.37464991998171"
println("Другое случайное число:
(generator.random())")
// prints "Другое случайное число:
0.729023776863283"
```


Требуемые изменяющиеся

методы

Иногда необходимо для метода изменить (или мутировать) экземпляр, которому он принадлежит. Для методов экземпляра типа значения (структура, перечисление) вы располагаете ключевое слово `mutating` до слова метода `func`, для индикации того, что этому методу разрешено менять экземпляр, которому он принадлежит, и/или любое свойство этого экземпляра. Этот процесс описан в главе “[Изменение типов значений методами экземпляра](#)”.

Если вы определяете требуемый протоколом метод экземпляра, который предназначен менять экземпляры любого типа, которые принимают протокол, то поставьте ключевое слово `mutating` перед именем метода, как часть определения протокола. Это позволяет структурам и перечислениям принимать протокол и удовлетворять требованию метода.

Заметка

Если вы поставили ключевое слово `mutating` перед требуемым протоколом методом экземпляра, то вам не нужно писать слово `mutating` при реализации этого метода для класса. Слово `mutating` используется только структурами или перечислениями.

Пример ниже определяет протокол `Toggable`, который определяет единственный требуемый метод экземпляра `toggle`. Как и предполагает имя метода, он переключает или инвертирует состояние любого типа, обычно меняя свойство этого типа.

Метод `toggle` имеет слово `mutating` как часть определения протокола `Togglable`, для отображения того, что этот метод меняет состояние соответствующего протоколу экземпляра при своем вызове:

```
protocol Togglable {
    mutating func toggle()
}
```

Если вы реализуете протокол `Togglable` для структур или перечислений, то эта структура или перечисление может соответствовать протоколу предоставляя реализацию метода `toggle`, который так же будет отмечен словом `mutating`.

Пример ниже определяет перечисление `OnOffSwitch`. Это перечисление переключается между двумя состояниями, отмеченными двумя случаями перечисления `on` и `off`. Реализация метода `toggle` перечисления отмечена словом `mutating`, чтобы соответствовать требованию протокола:

```
enum OnOffSwitch: Togglable {
    case Off, On
    mutating func toggle() {
        switch self {
            case Off:
                self = On
            case On:
                self = Off
        }
    }
}

var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
// lightSwitch теперь равен .On
```

Требуемые инициализаторы

Иногда протоколы могут требовать реализацию конкретного инициализатора типами соответствующими протоколу. Вы пишете эти инициализаторы как часть определения протокола, точно так же как и обычные инициализаторы, но только без фигурных скобок или без тела инициализатора:

```
protocol SomeProtocol {  
    init(someParameter: Int)  
}
```

Реализация класса соответствующему протоколу с требованием инициализатора

Мы можете реализовать требуемый инициализатор в соответствующем классе протоколу в качестве назначенного инициализатора или вспомогательного. В любом случае вам нужно отметить этот инициализатор ключевым словом `required`:

```
class SomeClass: SomeProtocol {  
    required init(someParameter: Int) {  
        // реализация инициализатора...  
    }  
}
```

Использование модификатора `required` гарантирует, что вы проведете явную или унаследованную реализацию требуемого инициализатора на всех подклассах соответствующего класса протоколу, так, чтобы они тоже соответствовали протоколу. Подробнее вы можете прочитать в главе [“Требуемые инициализаторы.”](#)

Заметка

Вам не нужно обозначать реализацию инициализаторов протокола модификатором `required` в классах, где стоит модификатор `final`, потому что конечные классы не могут иметь подклассы. Для более полной информации по модификатору `final` читайте главу [“Предотвращение переопределения.”](#)

Если подкласс переопределяет назначенный инициализатор суперкласса и так же реализует соответствующий инициализатор протоколу, то обозначьте реализацию инициализатора сразу двумя модификаторами `required` и `override`:

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // реализация инициализатора...
    }
}

class SomeSubClass: SomeSuperClass,
SomeProtocol {
    // "required" от соответствия протоколу
    SomeProtocol; "override" от суперкласса
    SomeSuperClass
    required override init() {
        // реализация инициализатора...
    }
}
```

Требуемые проваливающиеся инициализаторы

Протоколы могут определять требования проваливающихся инициализаторов для соответствующих протоколу типов, что определено в главе Проваливающиеся инициализаторы".

Требование проваливающегося инициализатора может быть удовлетворено проваливающимся инициализатором или непроваливающимся инициализатором соответствующего протоколу типа. Требование непроваливающегося инициализатора может быть удовлетворено непроваливающимся инициализатором или неявно развернутым проваливающимся инициализатором.

Протоколы как типы

Протоколы сами по себе не несут какой-то новой функциональности. Тем не менее любой протокол, который вы создаете становится полноправным типом, который вы можете использовать в вашем коде.

Так как это тип, то вы можете использовать протокол во многих местах, где можно использовать другие типы:

- Как тип параметра или возвращаемый тип в функции, методе, инициализаторе
- Как тип константы, переменной или свойства
- Как тип элементов массива, словаря или другого контейнера

Из-за того что протоколы являются типами, то их имена начинаются с заглавной буквы (как в случае `FullyNamed` или `RandomNumberGenerator`) для соответствия имен с другими типами Swift (`Int`, `String`, `Bool`, `Double`...)

Вот пример использования протокола в качестве типа:

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator:
RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() *
Double(sides)) + 1
    }
}
```

Этот пример определяет новый класс `Dice`, который отображает игральную кость с `n` количеством сторон для настольной игры. Экземпляры `Dice` имеют свойство `sides`, которое отображает количество сторон, которое они имеют, так же кубики имеют свойство `generator`, которое предоставляет генератор случайных чисел, из которого и берутся значения броска игрового кубика.

Свойство `generator` типа `RandomNumberGenerator`, таким образом вы можете установить его экземпляру любого типа, который соответствует протоколу `RandomNumberGenerator`. Больше ничего не требуется от экземпляра, присеваемого

этому свойству, кроме того, что этот экземпляр должен принимать протокол `RandomNumberGenerator`.

`Dice` так же имеет инициализатор для установки начальных значений. Этот инициализатор имеет параметр `generator`, который так же является типом `RandomNumberGenerator`. Вы можете передать значение любого соответствующего протоколу типа в этот параметр, когда инициализируете новый экземпляр `Dice`.

`Dice` предоставляет один метод экземпляра - `roll`, который возвращает целое значение от 1 и до количества сторон на игровой кости. Этот метод вызывает генератор метода `random`, для создания нового случайного числа от 0.0 и до 1.0, а затем использует это случайное число для создания значения броска игровой кости в соответствующем диапазоне (1...n). Так как мы знаем, что генератор принимает `RandomNumberGenerator`, то это гарантирует нам, что у нас будет метод `random`.

Вот как используется класс `Dice` для создания шестигранный игровой кости с экземпляром `LinearCongruentialGenerator` в качестве генератора случайных чисел:

```
var d6 = Dice(sides: 6, generator:
LinearCongruentialGenerator())
for _ in 1...5 {
    println("Бросок игровой кости
\ (d6.roll()) ")
}
// Бросок игровой кости равен 3
// Бросок игровой кости равен 5
// Бросок игровой кости равен 4
// Бросок игровой кости равен 5
// Бросок игровой кости равен 4
```

Делегирование

Делегирование - это шаблон, который позволяет классу или структуре передавать (или делегировать) некоторую ответственность экземпляру другого типа. Этот шаблон реализуется определением протокола, который инкапсулирует делегируемые полномочия, таким образом, что соответствующий протоколу тип (делегат) гарантировано получит функциональность, которая была ему делегирована. Делегирование может быть использовано для ответа на конкретное действие или для получения данных из внешнего источника без необходимости знания типа источника.

Пример ниже определяет два протокола для использования в играх, основанных на бросках игральные костей:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate {
    func gameDidStart(game: DiceGame)
    func game(game: DiceGame,
didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(game: DiceGame)
}
```

Протокол `DiceGame` является протоколом, который может быть принят любой игрой, которая включает игральную кость. Протокол `DiceGameDelegate` может быть принят любым типом для отслеживания прогресса `DiceGame`.

Вот версия игры “Змеи и лестницы”, которая первоначально была представлена в разделе “[Управление потоком](#)”. Эта версия адаптирована под использование экземпляра `Dice`

для своих бросков кости, для соответствия протоколу `DiceGame` и для уведомления `DiceGameDelegate` о прогрессе:

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator:
LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = [Int](count: finalSquare + 1,
repeatedValue: 0)
        board[03] = +08; board[06] = +11;
board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11;
board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare
{
            let diceRoll = dice.roll()
            delegate?.game(self,
didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare
> finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
    }
}
```

```

    }
    delegate?.gameDidEnd(self)
  }
}

```

Для описания процесса игры “Змеи и лесницы”, посмотрите раздел оператора [“Break”](#).

Эта версия игры обернута в класс `SnakesAndLadders`, который принимает протокол `DiceGame`. Он предоставляет свойство `dice` и метод `play` для соответствия протоколу. (Свойство `dice` объявлено как константное свойство, потому что оно не нуждается в изменении значения после инициализации, а протокол требует только чтобы оно было доступным.)

Настройка игры “Змеи и лестницы” происходит в инициализаторе класса `init()`. Все логика игры перемещается в метод `play` протокола, который использует требуемое свойство протокола для предоставления значений броска игровой кости.

Обратите внимание, что свойство `delegate` определено как опциональное `DiceGameDelegate`, потому что делегат не требуется для игры. Так как оно является опциональным типом, свойство `delegate` автоматически устанавливает начальное значение равное `nil`. Таким образом, у каждого экземпляра игры есть установки свойства подходящему делегату.

`DiceGameDelegate` предоставляет три метода для отслеживания прогресса игры. Эти три метода были включены в логику игры внутри метода `play`, и вызываются когда начинается новая игра, начинается новый ход или игра кончается.

Так как свойство `delegate` является опциональным `DiceGameDelegate`, метод `play` использует опциональную последовательность каждый раз, как вызывается этот метод у делегата. Если свойство `delegate` равно `nil`, то этот вызов этого метода делегатом проваливается без возникновения ошибки. Если свойство `delegate` не `nil`, вызываются методы делегата, которые передаются в экземпляр `SnakesAndLadders` в качестве параметра.

Следующий пример показывает класс `DiceGameTracker`, который принимает протокол `DiceGameDelegate`:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            println("Начали новую игру Змеи и
лестницы")
        }
        println("У игровой кости
\(game.dice.sides) граней")
    }
    func game(game: DiceGame,
didStartNewTurnWithDiceRoll diceRoll: Int) {
        ++numberOfTurns
        println("Выкинули \(diceRoll)")
    }
    func gameDidEnd(game: DiceGame) {
        println("Длительность игры
\(numberOfTurns) хода")
    }
}
```

`DiceGameTracker` реализует все три метода, которые требует `DiceGameDelegate`. Он использует эти методы для отслеживания количества ходов, которые были сделаны в игре. Он сбрасывает значение свойства `numberOfTurns` на ноль, когда начинается игра и увеличивает каждый раз, как начинается новый ход и выводит общее число ходов, как только кончается игра. Реализация `gameDidStart`, показанная ранее, использует параметр `game` для отображения вступительной информации об игре, в которую будут играть. Параметр `game` имеет тип `DiceGame`, но `HESnakesAndLadders`, так что `gameDidStart` может получить и использовать только те методы и свойства, которые реализованы как часть протокола `DiceGame`. Однако метод все еще может использовать приведение типов для обращения к типу основного (исходного) экземпляра. В этом примере, он проверяет действительно ли `game` является экземпляром `SnakesAndLadders` или нет, а потом выводит соответствующее сообщение.

`gameDidStart` так же получает доступ к свойству `dice`, передаваемого параметра `game`. Так как известно, что `game` соответствует протоколу `DiceGame`, то это гарантирует наличие свойства `dice`, таким образом метод `gameDidStart` может получить доступ и вывести сообщение о свойстве кости `sides`, независимо от типа игры, в которую играют.

Теперь давайте взглянем на то, как выглядит `DiceGameTracker` в действии:

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Начали новую игру Змеи и лестницы
// У игровой кости 6 граней
// Выкинули 3
// Выкинули 5
// Выкинули 4
// Выкинули 5
// Длительность игры 4 хода
```

Добавление соответствий протоколу через расширения

Вы можете расширить существующий тип для того, чтобы он соответствовал протоколу, даже если у вас нет доступа к источнику кода для существующего типа. Расширения могут добавлять новые свойства, методы и сабскрипты существующему типу, что таким образом может удовлетворить любым требованиям протокола. Для более полной информации читайте [“Расширения”](#).

Заметка

Существующие экземпляры типа автоматически принимают и отвечают требованиям протокола, когда опции, необходимые для соответствия добавляются через расширение типа.

К примеру, этот протокол `TextRepresentable` может быть реализован любым типом, который может отображать текст. Это может быть собственное описание или текстовая версия текущего состояния:

```
protocol TextRepresentable {  
    func asText() -> String  
}
```

Класс `Dice`, о котором мы говорили ранее, может быть расширен для принятия и соответствия протоколу `TextRepresentable`:

```
extension Dice: TextRepresentable {  
    func asText() -> String {  
        return "Игральная кость с \$(sides)  
гранями"  
    }  
}
```

Это расширение принимает новый протокол в точности так же, как если бы `Dice` был представлен внутри его первоначальной реализации. Имя протокола предоставляется после имени типа и разделяются между собой двоеточием, и реализация всех требований протокола обеспечивается внутри фигурных скобок расширения.

Теперь экземпляр `Dice` может быть использован как `TextRepresentable`:

```
let d12 = Dice(sides: 12, generator:  
LinearCongruentialGenerator())  
println(d12.asText())  
// выводит "Игральная кость с 12 гранями"
```

Аналогично игровой класс `SnakesAndLadders` может быть расширен для того, чтобы смог принять и соответствовать протоколу `TextRepresentable`:

```
extension SnakesAndLadders: TextRepresentable
{
    func asText() -> String {
        return "Игра Змеи и Лестницы с полем в
\\(finalSquare) клеток"
    }
}

println(game.asText())
// выводит "Игра Змеи и Лестницы с полем в 25
клеток"
```

Объявление принятия протокола через расширение

Если тип уже соответствует всем требованиям протоколу, но еще не заявил, что он принимает этот протокол, то вы можете сделать это через пустое расширение:

```
struct Hamster {
    var name: String
    func asText() -> String {
        return "Хомяка назвали \\(name)"
    }
}

extension Hamster: TextRepresentable {}
```

Экземпляры `Hamster` теперь могут быть использованы в тех случаях, когда нужен тип `TextRepresentable`:

```
let simonTheHamster = Hamster(name: "Фруша")
let somethingTextRepresentable:
TextRepresentable = simonTheHamster
println(somethingTextRepresentable.asText())
// выводит "Хомяка назвали Фруша"
```

Заметка

Типы не принимают протоколы автоматически, если они удовлетворяют их требованиям. Принятие протокола должно быть объявлено в явной форме.

Коллекции типов протокола

Протоколы могут использовать в качестве типов, которые хранятся в таких коллекциях как массивы или словари, что упоминалось ранее в ["Протоколы как типы"](#).

Пример ниже создает массив из элементов типа `TextRepresentable`:

```
let things: [TextRepresentable] = [game, d12,
simonTheHamster]
```

Теперь мы можем перебирать элементы массива и выводить текстовое отображение каждого из них:

```
for thing in things {
    println(thing.asText())
}
// Игра Змеи и Лестницы с полем в 25 клеток
// Игральная кость с 12 гранями
// Хомяка назвали Фруша
```


Обратите внимание, что константа `thing` является типом `TextRepresentable`. Она не является типом `Dice`, или `DiceGame`, или `Hamster`, даже в том случае, если базовый тип является одним из них. Тем не менее из-за того, что она типа `TextRepresentable`, а все что имеет тип `TextRepresentable`, имеет метод `asText`, что значит, что можно безопасно вызывать `thing.asText` каждую итерацию цикла.

Наследование протокола

Протокол может наследовать один или более других протоколов и может добавлять требования поверх тех требований протоколов, которые он наследует. Синтаксис наследования протокола аналогичен синтаксису наследования класса, но с возможностью наследовать сразу несколько протоколов, которые разделяются между собой запятыми:

```
protocol InheritingProtocol: SomeProtocol,
AnotherProtocol {
    // определение протокола...
}
```

Ниже приведен пример протокола, который наследует протокол `TextRepresentable`, о котором мы говорили ранее:

```
protocol PrettyTextRepresentable:
TextRepresentable {
    func asPrettyText() -> String
}
```

Этот пример определяет новый протокол `PrettyTextRepresentable`, который наследует **из** `TextRepresentable`. Все, что соответствует протоколу `PrettyTextRepresentable`, должно удовлетворять всем требованиям `TextRepresentable`, плюс дополнительные требования введенные от

протокола `PrettyTextRepresentable`. В этом примере `PrettyTextRepresentable` добавляет единственное требование обеспечить метод экземпляра `asPrettyText`, который возвращает `String`.

Класс `SnakesAndLadders` может быть расширен, чтобы иметь возможность принять и соответствовать `PrettyTextRepresentable`:

```
extension SnakesAndLadders:
  PrettyTextRepresentable {
    func asPrettyText() -> String {
      var output = asText() + ":\n"
      for index in 1...finalSquare {
        switch board[index] {
          case let ladder where ladder > 0:
            output += "▲ "
          case let snake where snake < 0:
            output += "▼ "
          default:
            output += "○ "
        }
      }
      return output
    }
  }
```

Это расширение утверждает, что оно принимает протокол `PrettyTextRepresentable` и реализует метод `asPrettyText` для типа `SnakesAndLadders`. Все, что является типом `PrettyTextRepresentable`, так же должно быть и `TextRepresentable`, таким образом, реализация `asPrettyText` начинается с вызова метода `asText` из протокола `TextRepresentable` для начала вывода строки. Затем он добавляет двоеточие и символ разрыва строки для начала текстового отображения. Затем он проводит перебор элементов массива (клеток доски) и

добавляет их геометрические формы для отображения контента:

- Если значение клетки больше нуля, то это является началом лестницы, и это отображается символом ▲.
- Если значение клетки меньше нуля, то это голова змеи, и эта ячейка имеет символ ▼.
- И наоборот, если значение клетки равно нулю, то это “свободная” клетка, которая отображается символом ○.

Эта реализация метода может быть использована для вывода текстового описания любого экземпляра `SnakesAndLadders`:

```
println(game.asPrettyText())  
// Игра Змеи и Лестницы с полем в 25 клеток:  
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼  
○ ▼ ○
```

Классовые протоколы

Вы можете ограничить протокол так, чтобы его могли принимать только классы (но не структуры или перечисления), добавив ключевое слово `class` к списку наследования протокола. Слово `class` всегда должно появляться на первом месте списка наследования, до того, как будут вписаны наследуемые протоколы:

```
protocol SomeClassOnlyProtocol: class,  
SomeInheritedProtocol {  
    // определение протокола типа class-only  
}
```

В примере выше `SomeClassOnlyProtocol` может быть принят только классом. Если вы попытаетесь принять протокол `SomeClassOnlyProtocol` структурой или перечислением, то получите ошибку компиляции.

Заметка

Используйте протоколы `class-only`, когда поведение, определяемое протоколом, предполагает или требует, что соответствующий протоколу тип должен быть ссылочного типа, а не типом значения. Для более детального исследования с вашей стороны прочитайте главы: "[Структуры и перечисления - типы значения](#)" и "[Классы - ссылочный тип](#)".

Композиция протоколов

Иногда бывает удобно требовать тип соответствовать нескольким протоколам за раз. Вы можете скомбинировать несколько протоколов в одно единственное требование при помощи *композиции протоколов*. Композиции протоколов имеют форму `protocol<SomeProtocol, AnotherProtocol>`. Вы можете перечислить столько протоколов в угловых скобках (`<>`), сколько вам нужно, разделяя их запятыми.

Ниже приведен пример, который комбинирует два протокола `Named` и `Aged` в одно единственное требование композиции протоколов в качестве параметра функции:

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator:
protocol<Named, Aged>) {
    println("С Днем Рождения,
\\(celebrator.name)! Тебе уже
\\(celebrator.age)!")
}
let birthdayPerson = Person(name: "Сашка",
age: 21)
wishHappyBirthday(birthdayPerson)
// выводит "С Днем Рождения, Сашка! Тебе уже
21!"
```

В этом примере мы определяем протокол `Named` с единственным требованием свойства `name` типа `String`, значение которого мы можем получить. Так же мы определяем протокол `Aged` с единственным требованием свойства `age` типа `Int`, значение которого мы так же должны мочь получить. Оба этих протокола принимаются структурой `Person`.

Этот пример определяет функцию `wishHappyBirthday`, которая принимает единственный параметр `celebrator`. Тип

этого параметра `protocol<Named, Aged>`, что означает “любой тип, который соответствует сразу двум протоколам `Aged` и `Named`”. Это не важно какой тип передается в качестве параметра функции до тех пор, пока он соответствует этим протоколам.

Далее в примере мы создает экземпляр `birthdayPerson` класса `Person` и передаем этот новый экземпляр в функцию `wishHappyBirthday`. Из-за того что `Person` соответствует двум протоколам, то функция `wishHappyBirthday` может вывести поздравление с днем рождения.

Заметка

Композиции протоколов не определяют нового, постоянного типа протокола. Вместо этого они определяют временный локальный протокол, который имеет комбинированные требования всех протоколов композиции.

Проверка соответствия протоколу

Вы можете использовать операторы `is` и `as`, которые описаны в главе “[Приведение типов](#)”, для проверки соответствия протоколу и приведению к определенному протоколу. Приведение к протоколу проходит точно так же как и приведение к типу:

- Оператор `is` возвращает значение `true`, если экземпляр соответствует протоколу и возвращает `false`, если нет.
- Опциональная версия оператора понижающего приведения `as?` возвращает опциональное значение типа протокола, и это значение равно `nil`, если оно не соответствует протоколу.

- Принудительная версия оператора понижающего приведения `as` осуществляет принудительное понижающее приведение, и если оно не завершается успешно, то выскакивает runtime ошибка.

Этот пример определяет протокол `HasArea` с единственным требованием свойства `area` типа `Double` (доступное свойство):

```
protocol HasArea {  
    var area: Double { get }  
}
```

Заметка

Вы можете проверить соответствие протоколу только в том случае, если ваш протокол отмечен атрибутом `@objc`, как вы это видели в примере выше. Этот атрибут является индикатором того, что протокол должен быть представлен в коде на Objective-C. И если вы не взаимодействуете с Objective-C, то вы должны указывать ваши протоколы атрибутом `@objc`, если вы хотите проверить соответствие протоколу.

Так же обратите внимание `@objc` протоколы могут быть приняты только классами, но не структурами или перечислениями. Если вы обозначите свой протокол как `@objc` для того, чтобы проверить соответствие, то вы сможете использовать этот протокол только в классах.

Ниже представлены два класса `Circle`, `Country`, оба из которых соответствуют протоколу `HasArea`:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius *
radius }
    init(radius: Double) { self.radius =
radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Класс реализует требование свойства `area` в качестве вычисляемого свойства, основываясь на хранимом свойстве `radius`. Класс `Country` реализует требование `area` напрямую в качестве хранимого свойства. Оба класса корректно соответствуют протоколу `HasArea`.

Ниже приведен класс `Animal`, который не соответствует протоколу `HasArea`:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```


Классы `Circle`, `Country`, `Animal` не имеют общего базового класса. Тем не менее они все являются классами, и их экземпляры могут быть использованы для инициализации массива, который хранит значения типов `AnyObject`:

```
let objects: [AnyObject] = [  
    Circle(radius: 2.0),  
    Country(area: 243_610),  
    Animal(legs: 4)  
]
```

Массив `objects` инициализирован при помощи литерала, содержащего экземпляры `Circle`, который имеет `radius` равный 2, экземпляр типа `Country`, который инициализирован площадью Великобритании в квадратных километрах, и экземпляром класса `Animal`, который инициализирован количеством ног.

Массив `objects` может быть перебран, и каждый элемент массива может быть проверен на соответствие протоколу `HasArea`:

```
for object in objects {  
    if let objectWithArea = object as? HasArea  
    {  
        println("Площадь равна  
        \ (objectWithArea.area) ")  
    } else {  
        println("Что-то такое, что не имеет  
        площади")  
    }  
}  
  
// Площадь равна 12.5663708  
// Площадь равна 243610.0  
// Что-то такое, что не имеет площади
```

Каждый раз, когда объект массива соответствует протоколу `HasArea`, возвращается опциональное значение при помощи оператора `as?`, которое разворачивается при помощи опциональной связи в константу `objectWithArea`. Константа `objectWithArea` является типом `HasArea`, таким образом, свойство `area` может быть доступно и выведено на экран способом вывода через сам тип.

Обратите внимание, что базовые объекты не меняются в процессе приведения типа. Они остаются `Circle`, `Country` и `Animal`. Однако в момент, когда они хранятся в константе `objectWithArea`, известно лишь то, что они являются типом `HasArea`, так что мы можем получить доступ только к свойству `area`.

Опциональные требования протоколу

Вы можете определить опциональные требования для протокола. Эти требования не обязательно должны быть реализованы для соответствия протоколу. Опциональные требования должны иметь префиксный модификатор `optional` в качестве части определения протокола.

Опциональное требование протокола может быть вызвано при помощи опциональной последовательности, чтобы учесть возможность того, что требование не будет реализовано типом, который соответствует протоколу. Для более полной информации о опциональной последовательности читайте “[Опциональная последовательность](#)”.

Вы проверяете реализацию опционального требования, написав вопросительный знак после имени требования, когда оно вызывается,

например `someOptionalMethod?(someArgument)`.

Опциональное требование свойства и опциональное требование метода, которые возвращают значения, будут всегда возвращать опциональное значение соответствующего типа, когда вы считываете их значения или вызываете их, для отображения того факта, что опциональное требование может быть и не реализовано.

Заметка

Опциональное требование протокола может указано только в том случае, если ваш протокол имеет маркировку атрибутом `@objc`. Даже если вы не работаете с Objective-C, вам все равно нужно маркировать ваши протоколы при помощи атрибута `@objc`, если вы хотите указать опциональные требования.

Так же обратите внимание, что `@objc` протоколы могут быть приняты только классами, а не структурами или перечислениями. Если вы отметите ваш протокол при помощи `@objc` для того, чтобы указать опциональные требования, то вы можете применить этот протокол только с классами.

Следующий пример определяет класс `Counter`, который использует источник внешних данных для предоставления значение их инкремента. Этот источник внешних данных определен протоколом `CounterDataSource`, который имеет два опциональных требования:

```
@objc protocol CounterDataSource {
    optional func incrementForCount(count:
Int) -> Int
    optional var fixedIncrement: Int { get }
}
```

Протокол `CounterDataSource` определяет опциональное требование метода `incrementForCount` и опциональное требование свойства `fixedIncrement`. Эти требования определяют два разных способа для источника данных для предоставления подходящего значения инкремента для экземпляра `Counter`.

Заметка

Строго говоря, вы можете написать пользовательский класс, который соответствует протоколу `CounterDataSource` без реализации какого-либо требования этого протокола. Они оба опциональные, в конце концов. Хотя технически это допускается, но это не будет реализовываться для хорошего источника данных.

Класс `Counter`, определенный ниже, имеет опциональное свойство `dataSource` типа `CounterDataSource?`:

```
@objc class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount =
dataSource?.incrementForCount?(count) {
            count += amount
        } else if let amount =
dataSource?.fixedIncrement? {
            count += amount
        }
    }
}
```

Класс `Counter` хранит свое текущее значение в переменном свойстве `count`. Класс `Counter` так же определяет метод `increment`, который увеличивает свойство `count`, каждый раз, когда вызывается этот метод.

Метод `increment` сначала пытается получить значение инкремента, загляывая в реализацию метода `incrementForCount` в его источнике данных. Метод `increment` использует опциональную последовательность для попытки вызвать `incrementForCount` и передает текущее значение `count` как единственный аргумент метода.

Обратите внимание, что здесь всего два уровня опциональной последовательности. Первый - возможный источник данных `dataSource`, который может быть `nil`, так что `dataSource` имеет вопросительный знак после имени для индикации того, что метод `incrementForCount` может быть вызван только в том случае, если `dataSource` не `nil`. Второй уровень говорит нам о том, что даже если `dataSource` существует, у нас все равно нет гарантии того, что он реализует метод `incrementForCount`, потому что это опциональное требование. Именно по этой причине `incrementForCount` записан с вопросительным знаком после своего имени.

Так как вызов `incrementForCount` может провалиться по одной из этих двух причин, вызов возвращает нам значение типа опционального `Int`. Это верно даже если `incrementForCount` **определено** как возвращающее неопциональное значение `Int` в определении `CounterDataSource`.

После вызова `incrementForCount` опциональный `Int`, который он возвращает, разворачивается в константу `amount`, при помощи опциональной связки. Если опциональный `Int` содержит значения, то есть, если делегат и метод существуют, и метод вернул значение, то неразвернутое значение `amount` прибавляется в свойство `count`, и на этом реализация завершается.

Если же нет возможности получить значение из метода `incrementForCount` по причине `dataSource` равен `nil` или из-за того что у источника данных нет реализации метода `incrementForCount`, а следовательно вместо этого метод `increment` пытается получить значение от источника данных `fixedIncrement`.

Свойство `fixedIncrement` опциональное требование свойства, так что его имя так же написано в опциональной последовательности с вопросительным знаком, что служит индикатором того, что попытка получить значение этого свойства может привести к провалу. Как и раньше, возвращаемое значение является опциональной `Int`, даже тогда `fixedIncrement` определен как свойство типа неопционального `Int`, в качестве части определения протокола `CounterDataSource`.

Ниже приведена простая реализация `CounterDataSource`, где источник данных возвращает постоянное значение 3, каждый раз, как получает запрос. Это осуществляется благодаря тому, что реализуется опциональное требование свойства `fixedIncrement`:

```
class ThreeSource: CounterDataSource {  
    let fixedIncrement = 3  
}
```

Вы можете использовать экземпляр `ThreeSource` в качестве источника данных для новых экземпляров `Counter`:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    println(counter.count)
}
// 3
// 6
// 9
// 12
```

Код, приведенный выше, создает новый экземпляр `Counter`, устанавливает его источник данных как экземпляр `ThreeSource` и вызывает метод счетчика `increment` четыре раза. Как и ожидалось, свойство счетчика увеличивается на три каждый раз, когда вызывается `increment`.

Ниже приведен более сложный источник данных `TowardsZeroSource`, который заставляет экземпляр `Counter` считать в сторону увеличения или уменьшения по направлению к нулю от текущего значения `count`:

```
class TowardsZeroSource: CounterDataSource {
    func incrementForCount(count: Int) -> Int
    {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

Класс `TowardsZeroSource` реализует опциональный метод `incrementForCount` из протокола `CounterDataSource` и использует значение аргумента `count` для определения направления следующего счета. Если `count` уже ноль, то метод возвращает `0`, для отображения того, что дальнейших вычислений не требуется.

Вы можете использовать экземпляр `TowardsZeroSource` с уже существующим экземпляром `Counter` для отсчета от -4 и до 0. Как только счетчик достигает 0, вычисления прекращаются:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {

    counter.increment()
    println(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```


Универсальные шаблоны

Универсальный код позволяет вам писать гибкие, общего назначения функции и типы, которые могут работать с любыми другими типами, с учетом требований, которые вы определили. Вы можете написать код, который не повторяется и выражает свой контент в ясной абстрактной форме.

Универсальные шаблоны одна из самых мощных особенностей Swift, и большая часть всех библиотек Swift построена на основе универсального кода. На самом деле вы используете универсальный код все время, даже если вы этого не осознаете. Например, коллекции Swift `Array` или `Dictionary` являются универсальными. Вы можете создать массив, который содержит значения типа `Int` или массив, который содержит значения `String`, или на самом деле любой другой массив, который может содержать любой другой тип. Аналогично вы создаете словарь, который может содержать значения разных типов, и нет никакого ограничения по типу хранящихся значений.

Проблема, которую решают универсальные шаблоны

Приведем обычную, стандартную, неуниверсальную функцию `swapTwoInts`, которая меняет два `Int` местами:

```
func swapTwoInts(inout a: Int, inout b: Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

Эта функция использует сквозные параметры для замещения значения `a` и `b`, что описано в [“Имена параметров функций”](#).

Функция `swapTwoInts` обменивает начальные значения переменных `a` и `b` местами. Вы можете использовать эту функцию для замещения двух значений типа `Int`:

```
var someInt = 3  
var anotherInt = 107  
swapTwoInts(&someInt, &anotherInt)  
println("someInt теперь \(someInt), а  
anotherInt теперь \(anotherInt)")  
// выводит "someInt теперь 107, а anotherInt  
теперь 3"
```

Функция `swapTwoInts` полезная, но она применима только для значений типа `Int`. Если вы хотите поменять местами два значения типа `String` или два значения `Double`, то вам

придется написать больше функций, к примеру, `swapTwoStrings` или `swapTwoDoubles`, которые показаны ниже:

```
func swapTwoStrings(inout a: String, inout b:
String) {
    let temporaryA = a
    a = b
    b = temporaryA
}
func swapTwoDoubles(inout a: Double, inout b:
Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Вы может быть заметили, что тела функций `swapTwoStrings` и `swapTwoDoubles` идентичны. Единственное отличие в том, что они поддерживают значения различных типов.

Было бы намного удобнее написать одну более гибкую функцию, которая бы могла заменить значения двух переменных любого типа. Универсальный код позволяет вам написать такую функцию. (Универсальная версия этой функции приведена ниже.)

Заметка

Во всех трех функциях есть важный момент того, что типы `a` и `b` должны быть одинаковыми по отношению друг к другу. Если `a` и `b` не являются значениями одного типа, то будет невозможно поменять их значения местами. Swift является языком типо-безопасным языком и не позволяет переменным с разными типами меняться значениями друг с другом. Попытка сделать это приведет к ошибке компиляции.

Универсальные шаблоны

Универсальные функции могут работать с любыми типами. Ниже приведена универсальная версия функции `swapTwoInts`, которая теперь называется `swapTwoValues`:

```
func swapTwoValues<T>(inout a: T, inout b: T)
{
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Тело функции `swapTwoValues` идентично телу функции `swapTwoInts`. Однако первая строка функции `swapTwoValues` немного отличается от аналогичной строки функции `swapTwoInts`. Вот как можно сравнить первые строки этих функций:

```
func swapTwoInts(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

Универсальная версия использует заполнитель имени типа (называется `T` в нашем случае) вместо текущего имени типа (`Int`, `String`, `Double`...). Заполнитель имени типа ничего не говорит о том, чем должно являться `T`, но он говорит о том, что и `a` и `b` должны быть одного типа `T`, не зависимо от того, что такое `T`. Текущий тип `T` будет определяться каждый раз, как вызывается функция `swapTwoValues`.

Другое отличие в том, что за именем универсальной функции идет заполнитель имени типа в угловых скобках `()`. Угловые скобки говорят Swift, что `T` является заполнителем имени

типа внутри определения функции `swapTwoValues`. Так как `T` является заполнителем, то Swift не смотрит на текущее значение `T`.

Функция `swapTwoValues` теперь может быть вызвана точно так же как и функция `swapTwoInts`, за исключением того, что в нее можно передавать значения любого типа, до тех пор пока они одинаковые. Каждый раз при вызове функции `swapTwoValues` используется тип, который выводится из значений переданных функции.

В двух примерах ниже `T` имеет значение типа `Int` и `String` соответственно:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt теперь 107, а anotherInt теперь 3
var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString теперь "world", а anotherString
теперь "hello"
```

Заметка

Определенная функция `swapTwoValues` на самом деле имеется в стандартной библиотеке и называется `swap`, так что вы можете автоматически использовать ее в ваших приложениях, а не писать свою собственную `swapTwoValues`, например.

Параметры типа

В примере выше в функции `swapTwoValues` заполнитель имени типа `T` пример *параметра типа*. Параметры типа определяют и называют тип наполнителя, и пишутся сразу после имени функции, между угловыми скобками (например `<T>`).

Как только вы определили параметр типа, то вы можете использовать его в качестве типа параметра функции (как например, параметры `a` и `b` в функции `swapTwoValues`) или как возвращаемый функциональный тип, или как аннотация типа внутри тела функции. В каждом случае заполнитель типа отображается параметром типа, который заменяется на актуальное значение типа при вызове функции. (В нашем примере выше произошло замещение заполнителя типа на `Int`, а во втором случае на `String`.)

Вы можете использовать несколько параметров типа, просто вписав их в угловых скобках через запятую.

Именованние параметров

типа

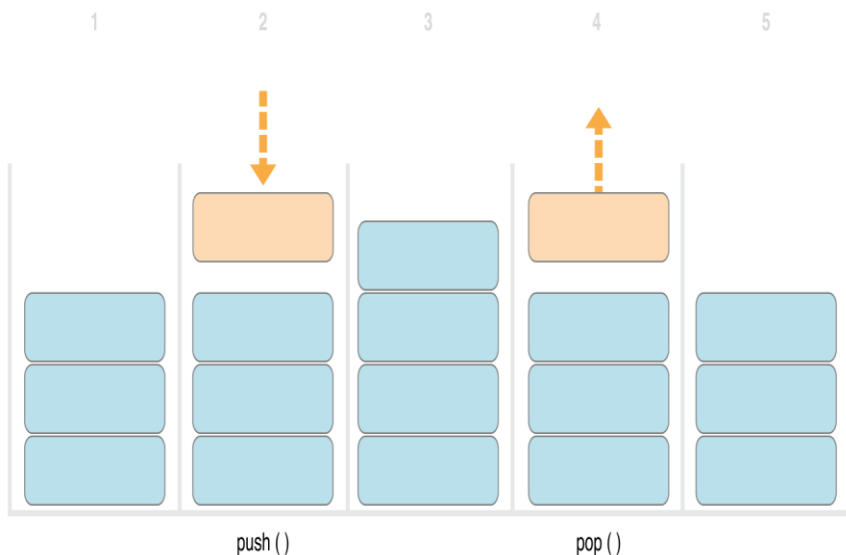
В дополнение к универсальным функциям, Swift позволяет вам определять ваши *универсальные типы*. Это к примеру универсальные классы, структуры и перечисления, которые могут работать с любыми типами, на подобии тому, как работают массивы или словари.

Эта секция покажет вам как создать вашу универсальную коллекцию типа `Stack`. `Stack` - упорядоченная коллекция значений, аналогичная массиву, но с более строгим набором операций, чем имеет типа `Array` языка Swift. Массив позволяет вам вставлять и удалять элементы с любой позиции массива. Однако, `Stack` позволяет добавлять новые элементы только в конец коллекции (известно как *заталкивание* (или `pushing` на англ) нового значения в стек). Аналогично стек позволяет удалять элементы только с конца коллекции (известно как *выстреливать значение из стека* (или `poping` по англ)).

Заметка

Концепция стека используется классом `UINavigationController` для моделирования контроллеров видов в его иерархии навигации. Вы вызываете метод `pushViewController:animated:` класса `UINavigationController` для добавления контроллера вида на стек навигации, а метод `popViewControllerAnimated:` для удаления контроллера вида из стека навигации. Стек - полезная модель коллекции, когда вам нужен строгий принцип “последний на вход - первый на выход”.

Ниже приведена иллюстрация поведения добавления и удаления элемента из стека:



1. На данный момент у нас три значения в стеке.
2. Четвертое значение “затолкнули” на самый верх стека.
3. На этот момент в стеке находится три значения, самое свежее значение находится наверху.
4. Последнее значение удалено или “выстреляно” из стека.
5. После удаления значения, стек снова имеет три значения.

Вот как написать не универсальную версию стека, в этом случае мы используем стек для хранения `Int` значений:

```
struct IntStack {
    var items = [Int]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

Эта структура использует свойство `items` типа `Array` для хранения значений в стеке. `Stack` предоставляет нам два метода, `push` и `pop` для добавления последнего элемента в стек и для удаления последнего элемента из стека. Эти методы отмечены как `mutating`, потому как они вынуждены менять массив `items`.

Тип `IntStack`, показанный выше, может быть использован только со значениями `Int`. Но он будет куда полезнее, если мы определим его как универсальный класс `Stack`, который может управлять стеком любого типа. Вот универсальная версия структуры:

```
struct Stack<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

Обратите внимание как универсальная версия `Stack` похожа на не универсальную, вообще отличаясь только тем, что мы используем заполнитель типа, вместо указания конкретного типа `Int`. Этот параметр типа написан внутри угловых скобок , сразу после имени структуры.

`T` определяет заполнитель имени типа для “какого-то типа `T`”, который будет предоставлен позже. Этот будущий тип может быть сослан на “`T`” в любом месте определения структуры. В этом случае наш некоторый тип “`T`” используется в трех местах:

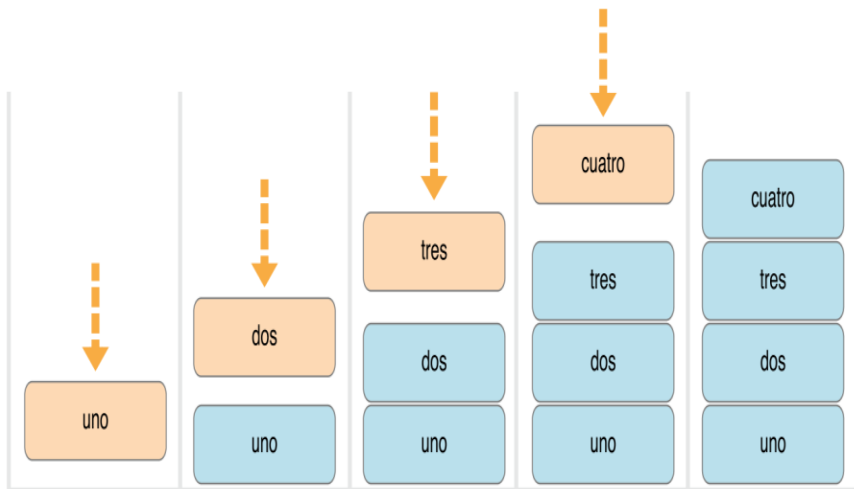
- Для создания свойства `items`, которое инициализируется пустым массивом типа `T`.
- Для указания того, что метод `push` имеет единственный параметр `item`, который должен быть типа `T`.
- Для указания типа возвращаемого значения методом `pop`, которое должно быть типом `T`.

Из-за того, что это является универсальным типом, то `Stack` может быть использован для создания стека любых корректных типов в Swift, аналогичным образом как это осуществляют типы `Array` или `Dictionary`.

Вы создаете новый экземпляр `Stack`, вписав тип хранимых значений стека в угловые скобки. Например, создадим новый стек строк, вы напишите `Stack()` :

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

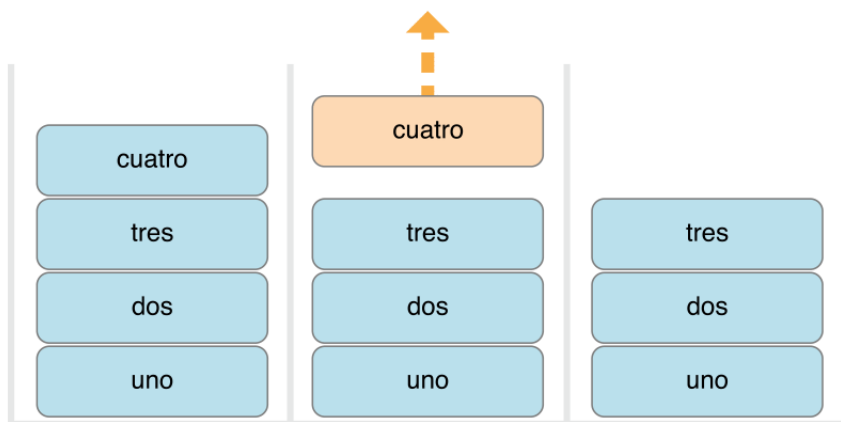
Теперь `stackOfStrings` выглядит вот так после добавления последних четырех значений:



Удаляя последнее значение, он возвращает его и удаляет его из стека "cuatro".

```
let fromTheTop = stackOfStrings.pop()  
// fromTheTop равно "cuatro", теперь стек  
содержит 3 строки
```

После удаления верхней величины, стек выглядит так:



Ограничения типа

Функция `swapTwoValues` и тип `Stack` могут работать с любыми типами. Однако иногда бывает нужно внедрить определенные ограничения типа на типы, которые могут быть использованы вместе универсальными функциями или универсальными типами. Ограничения типа указывают на то, что параметры типа должны наследовать от определенного класса или соответствовать определенному протоколу или композиции протоколов.

Для примера возьмем тип `Dictionary`, который имеет некоторые ограничения типов, которые могут быть использованы в качестве ключей. Как было описано в главе “Словари”, тип ключа словаря должен быть хешируемым. Таким образом он должен предоставить способ представить себя уникальным. `Dictionary` нужно, чтобы его ключи были хешируемыми, таким образом он может проверить, содержит ли конкретный ключ какое-либо значение. Без этого

требования, `Dictionary` не в состоянии понять, должен ли он заменить или вставить значение для конкретного ключа, и не в состоянии найти значение для конкретного ключа, которое уже есть в словаре.

Такое требование внедряется ограничениями типа для типа ключа словаря, которое определяет, что каждый ключ должен соответствовать протоколу `Hashable`, специальному протоколу, который определен в стандартной библиотеке Swift. Все базовые типа Swift (`String`, `Int`, `Double`, `Bool`) по умолчанию являются хешируемыми типами.

Вы можете определить свои собственные ограничения типа, когда создаете пользовательские универсальные классы, и эти ограничения предоставляют еще больше возможностей универсальному программированию. Абстрактные понятия, как `Hashable`, характеризуют типы с точки зрения их концептуальных характеристик, а не их явного типа.

Синтаксис ограничения типа

Вы пишете ограничения типа, поместив ограничение единственного класса или протокола после имени параметра типа, и разделив их между собой запятыми, обозначая их в качестве части списка параметров. Базовый синтаксис для ограничений типа универсальной функции показан ниже (хотя синтаксис для универсальных типов такой же):

```
func someFunction<T: SomeClass, U:
SomeProtocol>(someT: T, someU: U) {
    // тело функции...
}
```

Выше описанная гипотетическая функция имеет два параметра типа. Первый параметр типа - `T`, имеет

ограничение типа, которое требует чтобы `T`, было подклассом класса `SomeClass`. Второй параметр типа - `U`, имеет ограничение типа, которое требует чтобы `U` соответствовал протоколу `SomeProtocol`.

Ограничение типа в действии

Ниже приведена не универсальная функция `findStringIndex`, которая получает значение типа `String` для того, чтобы его найти, и массив значений типа `String`, внутри которого и будет происходить поиск. Функция `findStringIndex` возвращает опциональное значение `Int`, которое является индексом первого совпадения строки с элементом внутри массива или `nil`, которое означает отсутствие совпадения строки с каким-либо элементом массива:

```
func findStringIndex(array: [String], valueToFind: String) -> Int?
{
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Функция `findStringIndex` может быть использована для поиска строкового значения в массиве строк:

```
let strings = ["cat", "dog", "llama",
"parakeet", "terrapiin"]
if let foundIndex = findStringIndex(strings,
"llama") {
    println("Индекс llama равен
\ (foundIndex) ")
}
// выводит "Индекс llama равен 2"
```

Однако нахождение индекса совпадения значения в массиве бывает полезным не только для строк. Вы можете написать ту же функцию, но только в универсальной форме. Давайте напишем такую функцию и назовем ее `findIndex`, а все упоминания типа `String` заменим на тип `T`.

Вот как будет выглядеть версия функции `findStringIndex` в универсальной форме `findIndex`. Обратите внимание, что возвращаемый функцией тип все еще равен `Int?`, потому что функция возвращает опциональное значение индекса, а не опциональное значение элемента массива. Но будьте осторожны, так как эта функция не компилируется, по причинам, указанным после примера:

```
func findIndex<T>(array: [T], valueToFind: T)
-> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Как мы и сказали, эта функция не компилируется. Проблема находится в строке `if value == valueToFind`. Не каждый тип в Swift может быть сравнен оператором равенства (`==`). Если вы создаете свой класс или структуру для отображения сложной модели данных, например, то смысл выражения “равен чему-то” для этого класса или структуры Swift не может додумать за вас. Из-за этого нет никакой гарантии того, что этот код будет работать для любого возможного класса `T`, и соответствующая ошибка компиляции выскакивает, когда вы пытаетесь скомпилировать код.

Но не все еще потеряно. Стандартная библиотека Swift определяет протокол `Equatable`, который требует любой соответствующий ей тип реализовывать равенство оператору равенства (`==`) и реализовывать неравенство оператору неравенства (`!=`), для того, чтобы значения этих типов можно было сравнивать между собой. Все стандартные типы Swift автоматически поддерживают протокол `Equatable`.

Любой тип, который удовлетворяет протоколу `Equatable`, может быть безопасно использован в функции `findIndex`, потому что гарантирована поддержка оператора равенства и неравенства. Для отображения этого факта, вы пишете ограничение типа `Equatable`, как часть определения параметра типа, когда вы определяете функцию:

```
func findIndex<T: Equatable>(array: [T],
valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```


Единственный параметр типа для функции `findIndex` записывается как `T: Equatable`, что означает “любой тип, который соответствует протоколу `Equatable`”.

Теперь функция `findIndex` благополучно компилируется и может быть использована с любыми типами `Equatable`, например, `String`, `Double`:

```
let doubleIndex = findIndex([3.14159, 0.1,  
0.25], 9.3)
```

```
// doubleIndex является опциональным Int без  
значения, так как значения 9.3 не существует в  
массиве
```

```
let stringIndex = findIndex(["Mike",  
"Malcolm", "Andrea"], "Andrea")
```

```
// stringIndex является опциональным Int  
который равен 2
```

Расширяем универсальный тип

Когда вы расширяете универсальный тип, вы не обеспечиваете список параметров в качестве определения расширения. Вместо этого, список параметров типа, из исходного определения типа, доступен внутри тела расширения, а имена исходных параметров типа используются для ссылки на параметры типа из исходного определения.

Следующий пример расширяет универсальный тип `Stack`, для добавления вычисляемого свойства только для чтения `topItem`, которое возвращает верхний элемент стека, без “выстреливания” его из этого стека:

```
extension Stack {  
    var topItem: T? {  
        return items.isEmpty ? nil :  
items[items.count - 1]  
    }  
}
```

Свойство `topItem` возвращает опциональное значение типа `T`. Если стек пустой, то `topItem` возвращает `nil`. Если стек не пустой, то `topItem` возвращает последний элемент массива `items`.

Обратите внимание, что расширение не определяет списка параметров типа. Вместо этого, имя существующего параметра типа `Stack` - `T` используется внутри расширения для отображения опционального типа вычисляемого свойства `topItem`.

Вычисляемое свойство теперь может быть использовано внутри экземпляра `Stack` для доступа к значению и для запроса к последнему элементу стека, без дальнейшего его удаления:

```
if let topItem = stackOfStrings.topItem {  
  
    println("Верхний элемент стека -  
\ (topItem) .")  
}  
// выводит "Верхний элемент стека - tres."
```

Связанные типы

При определении протокола бывает нужно определить еще один или более *связанных типов* в качестве части определения протокола. Связанный тип дает заполнитель имени (или алиас) типу, который используется как часть протокола. Фактический тип, который будет использоваться связанным типом не указывается до тех пор, пока не будет принят протокол. Связанные типы указываются при помощи ключевого слова `typealias`.

Связанные типы в действии

Ниже приведен пример протокола `Container`, который объявляет связанный тип `ItemType`:

```
protocol Container {  
    typealias ItemType  
    mutating func append(item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}
```

Протокол `Container` определяет три требуемых возможности, которые должен иметь любой контейнер:

- Должна быть возможность добавлять новый элемент в контейнер при помощи метода `append`.
- Должна быть возможность получить доступ к количеству элементов в контейнере через свойство `count`, которое возвращает значение типа `Int`.
- Должна быть возможность получить значение через индекс элемента, который принимает значение типа `Int`.

Этот протокол не указывает количество и способ хранения элементов в контейнере или какого типа они должны быть. Протокол только лишь указывает три “кусочка” функциональности, которые должны быть предоставлены контейнером, чтобы он считался `Container`. Соответствующий тип может предоставлять дополнительную функциональность, пока он удовлетворяет этим трем требованиям.

Любой тип, который удовлетворяет протоколу `Container` должен иметь возможность указывать на тип хранящихся элементов. Конкретно, он должен гарантировать, что только элементы правильного типа будут добавлены в контейнер, и должно быть ясно какой тип элементов будет возвращаться сабскриптом.

Для определения этих требований, протокол `Container` должен иметь способ ссылаться на тип элементов, которые он будет хранить, без указания типа элементов, которые может хранить конкретный контейнер. Протокол `Container` должен указать, что любое значение переданное в метод `append` должно иметь тот же тип, что и тип элементов контейнера, и что значение, возвращаемое сабскриптом контейнера, должно быть того же типа, что и элементы контейнера.

Чтобы добиться этого, протокол `Container` объявляет связанный тип `ItemType`, который записывается как `typealias ItemType`. Протокол не определяет для чего конкретно нужен алиас `ItemType`, потому что эта информация остается для любого соответствующего класса протоколу. Тем не менее, алиас `ItemType` предоставляет способ сослаться на тип элементов в `Container` и определить тип для использования метода `append` и сабскрипта, для того, чтобы гарантировать, что желаемое поведение любого `Container` имеет силу.

Ниже приведена версия не универсального типа `IntStack`, который адаптирован под протокол `Container`:

```
struct IntStack: Container {
    // исходная реализация IntStack
    var items = [Int]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // соответствие протоколу Container
    typealias ItemType = Int
    mutating func append(item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

Тип `IntStack` реализует все три требования протокола `Container`, и в каждом случае оборачивает часть существующей функциональности типа `IntStack` для удовлетворения этих требований.

Более того, `IntStack` указывает, что для этой реализации контейнера, подходящий тип `ItemType` будет `Int`. Определение `typealias ItemType = Int` преобразует абстрактный тип `ItemType` в конкретный тип `Int` для этой реализации протокола `Container`.

Благодаря выводу типов Swift, вам фактически не нужно указывать конкретный тип `Int` для `ItemType` как часть определения `IntStack`. Так как `IntStack` соответствует протоколу `Container`, Swift может вывести соответствующий тип для `ItemType`, просто посмотрев на тип параметра `item` метода `append` и на тип возвращаемого значения сабскрипта. И на самом деле, если удалить строку кода `typealias ItemType = Int`, все будет продолжать работать, потому что все еще ясно какой тип должен быть использован для `ItemType`.

Вы так же можете создать универсальный тип `Stack`, который соответствует протоколу `Container`:

```
struct Stack<T>: Container {
    // исходная реализация Stack<T>
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // соответствие протоколу Container
    mutating func append(item: T) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> T {
        return items[i]
    }
}
```

В этот раз заполнитель имени типа `T` использован в качестве параметра `item` метода `append` и в качестве возвращаемого типа сабскрипта. Таким образом Swift может вывести, что `T` подходящий тип для использования его в качестве `ItemType` для этого конкретного контейнера.

Расширение существующего типа для указания связанного типа

Вы можете расширить существующий тип для того, чтобы добавить соответствие протоколу, как описано в [“Добавление соответствия протоколу через расширение”](#). Это включает в себя протокол со связанным типом.

Тип `Array` уже предоставляет нам метод `append`, свойство `count` и сабскрипт со значением индекса типа `Int` для получения его элементов. Эти три возможности удовлетворяют совпадают с требованиями протокола `Container`. Это означает, что вы можете расширить тип `Array`, чтобы он соответствовал протоколу `Container`, просто указав, что `Array` принимает протокол `Container`. Вы можете сделать это при помощи пустого расширения, которое подробнее описано в подразделе главы [“Добавление соответствия протоколу через расширение”](#):

```
extension Array: Container {}
```

Существующий метод `append` типа `Array` и сабскрипт позволяют Swift выводить соответствующий тип для `ItemType`, точно так же как и для универсального типа `Stack`, который был приведен ранее. После определения расширения вы можете использовать `Array` как `Container`.

Оговорка where

Ограничения типа, как описано в главе “[Ограничения типа](#)”, позволяют вам определять требования параметров типа связанных с универсальными функциями или типами.

Так же бывает полезно определять требования для связанных типов. Чтобы сделать это, вы определяете оговорку `where`, как часть списка параметров типа.

Оговорка `where` позволяет вам требовать, чтобы связанный тип, соответствовал определенному протоколу, и/или чтобы конкретные параметры типа и связанные типы были одними и теми же. Вы пишете оговорку `where`, поместив ключевое слово `where` сразу после списка параметров типа, за которым следует одно или более ограничений для связанных типов, и/или один или более отношений равенства между типами и связанными типами.

В примере ниже определяем универсальную функцию `allItemMatch`, которая проверяет, чтобы увидеть содержат ли два экземпляра `Container` одни и те же элементы в одной и той же последовательности. Функция возвращает значение типа `Bool`, то есть, если у нас все элементы и их последовательность совпадает, то функция возвращает `true`, если нет - `false`.

Контейнеры не должны быть одного типа для того чтобы их проверить, хотя они и могут, но они должны содержать элементы одного и того же типа. Это требование выражается через комбинацию ограничений типа и оговоркой `where`:

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType,
    C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2)
-> Bool {

    // проверяет контейнеры на одинаковое
    количество элементов
    if someContainer.count !=
anotherContainer.count {
        return false
    }

    // проверяет каждую пару элементов на
    их эквивалентность
    for i in 0..
```

Эта функция принимает два аргумента `someContainer` и `anotherContainer`. Аргумент `someContainer` имеет тип `C1`, аргумент `anotherContainer` имеет тип `C2`. И `C1` и `C2` являются

заполнителями имен типов для двух контейнеров, которые будут определены, когда будет вызвана функция.

Список типов параметров функции размещает следующие требования на два параметра типа:

- `C1` должен соответствовать протоколу `Container` (`C1 : Container`)
- `C2` должен соответствовать протоколу `Container` (`C2 : Container`)
- `ItemType` для `C1` должен быть тем же, что и `ItemType` для `C2` (`C1.ItemType == C2.ItemType`)
- `ItemType` для `C1` должен соответствовать протоколу `Equatable` (`C1.ItemType : Equatable`)

Треть и четвертое требование определены как часть оговорки `where`, и записаны после ключевого слова `where`, в качестве части списка типов параметров функции.

Эти требования означают:

- `someCharacter` является контейнером типа `C1`.
- `anotherCharacter` является контейнером типа `C2`.
- `someCharacter` и `anotherCharacter` содержат значения одного типа
- Элементы в `someContainer` могут быть проверены при помощи оператора неравенства (`!=`), чтобы увидеть, что они отличаются друг от друга.

Третье и четвертое требование комбинируются так, чтобы элементы в `anotherContainer` так же могли бы быть проверены оператором `!=`, потому что они в точности одного и того же типа, что и `BsomeContainer`.

Эти требования позволяют функции `allItemsMatch` сравнивать два контейнера, даже если они являются контейнерами разного типа.

Функция `allItemsMatch` начинается с проверки количества элементов в этих контейнерах. Если они содержат разное количество элементов, то эти контейнеры уже не могут быть одинаковыми, функция возвращает `false`.

После проведения этой проверки, функция перебирает все элементы в `someContainer` при помощи `for-in` цикла и полукратного оператора диапазона (`..). Для каждого элемента someContainer функция проверяет равенство элемента соответствующему элементу в контейнере anotherContainer. Если два элемента не равны друг другу, то эти два контейнера не считаются одинаковыми, функция возвращает false.`

Если цикл закончился без каких-либо несоответствий элементов, то два контейнера считаются одинаковыми, и функция возвращает `true`.

Вот как выглядит функция `allItemsMatch` в действии:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
var arrayOfStrings = ["uno", "dos", "tres"]
if allItemsMatch(stackOfStrings,
arrayOfStrings) {
    println("Все элементы совпали.")
} else {
    println("Не все элементы совпали.")
}
// выводит "Все элементы совпали."
```

Пример выше создает экземпляр `Stack` для хранения значений типа `String` и добавляет три значения на стек. Так пример создает массив `arrayOfStrings`, который инициализируется литералом массива. Даже тогда стек и массив имеют разные типы, но оба они соответствуют протоколу `Container`, и оба они содержат одинаковый тип значений. Тем не менее вы можете вызвать функцию `allItemsMatch` с этими двумя контейнерами в качестве своих аргументов. В примере выше функция `allItemsMatch` корректно извещает нас, что все элементы этих двух контейнеров одинаковые.

Контроль доступа

Контроль доступа ограничивает доступ к частям вашего кода из кода других исходных файлов и модулей. Эта особенность позволяет вам прятать детали реализации вашего кода и указывать на предпочтительный интерфейс, через который можно получить доступ к вашему коду.

Вы можете присвоить определенные уровни доступа как к индивидуальным типам (классы, структуры и перечисления), так и к свойствам, методам, инициализаторам и сабскриптам, принадлежащим этим типам. Протоколы могут быть ограничены в определенном контексте, так же как могут быть ограничены глобальные переменные или функции.

В дополнение к предложению контроля различных уровней доступа, Swift уменьшает необходимость указания явного уровня контроля доступа тем, что сам обеспечивает уровни доступа по умолчанию для типичных сценариев. И на самом деле, если вы пишете простое приложение, то вам может и не понадобится указывать явно уровень контроля доступа вообще.

Заметка

Различные аспекты вашего кода, к которым применим контроль уровня доступа (свойства, типы, функции и т.д.) будем называть “объектами”, для краткости.

Модули и исходные файлы

Модель контроля доступа Swift основывается на концепции модулей и исходных файлов.

Модуль представляет из себя единый блок распределения кода - фреймворк или приложение, которое построено и поставляется в качестве единого блока и которое может быть импортировано другим модулем с ключевым словом `import`.

Каждая цель сборки (например, набор приложений или фреймворк) в Xcode обрабатывается как отдельный модуль. Если вы объедините вместе аспекты кода вашего приложения в качестве отдельного фреймворка, то их возможно будет инкапсулировать и использовать заново во множестве других приложений. Таким образом, все, что вы определите в рамках этого фреймворка будет считаться частью отдельного модуля, когда это будет импортировано и использовано внутри приложения, или когда это будет использовано внутри другого фреймворка.

Исходный файл - исходный код файла в пределах одного модуля (в сущности это и есть один файл вашего приложения или фреймворка). Хотя в большинстве случаев определение типов происходит в отдельных исходных файлах, но фактически исходный файл может содержать определения множества различных типов, функций и т.д.

Уровни доступа

Swift предлагает три различных уровня доступа для объектов вашего кода. Эти уровни доступа относительно исходному файлу, в котором определен объект, и так же они относительно модулю, которому принадлежит исходный файл:

- *Открытый* (**public**). Этот уровень доступа позволяет использовать объекты внутри любого исходного файла из определяющего их модуля и так же в любом исходном файле из другого модуля, который импортирует определяющий модуль. Вы обычно используете открытый доступ, когда указываете общий интерфейс фреймворка.
- *Внутренний* (**internal**). Этот уровень доступа позволяет использовать объекты внутри любого исходного файла из их определяющего модуля, но не исходного файла не из этого модуля. Вы обычно указываете внутренний доступ, когда определяете внутреннюю структуру приложения или фреймворка.
- *Частный* (**private**). Этот уровень доступа позволяет использовать объект в пределах его исходного файла. Используйте частный доступ для того, чтобы спрятать детали реализации определенной функциональности.

Открытый доступ - самый высокий уровень доступа (наименее строгий), и частный уровень доступа является самым низким уровнем доступа (самый строгий).

Руководящий принцип по выбору уровня доступа

Уровни доступа в Swift следуют общему руководящему принципу: никакой объект не может быть определен в

пределах другого объекта, который имеет более низкий (более строгий) уровень доступа.

Например:

- Переменная с уровнем доступа `public` не может быть определена как будто она имеет уровень доступа `private`, потому что этот уровень доступа не может быть использован везде, где доступен `public`.
- Функция не может иметь уровень доступа выше чем у ее параметров или возвращаемого типа, потому что функция не может использоваться там, где ее параметры не доступны.

Более глубокий смысл данного руководящего принципа для различных аспектов раскрывается ниже.

Уровень доступа по умолчанию (дефолтный)

Все объекты вашего кода (кроме двух исключений, о которых мы поговорим чуточку позже) имеют дефолтный уровень доступа - внутренний (`internal`), если вы явно не указываете другой уровень. В результате во многих случаях вам не нужно указывать явный уровень доступа в вашем коде.

Уровень доступа для простых однозадачных Вприложений

Когда вы пишете простое однозадачное приложение, то код вашего приложения обычно самодостаточен и не требует доступа к нему из внешних источников. По умолчанию уровень доступа стоит внутренний, так что это полностью удовлетворяет требованию кода. Таким образом, вам не нужно указывать явно этот уровень доступа. Однако, если вам все таки нужно, то вы можете некоторые части вашего

кода обозначить как `private`, для того чтобы спрятать детали реализации от другого кода этого же модуля.

Уровень доступа для фреймворка

Когда вы разрабатываете фреймворк, обозначьте внешний интерфейс фреймворка как `public`, так чтобы его можно было посмотреть и получить к нему доступ из других модулей, так например, чтобы приложение могло импортировать его. Внешний интерфейс - интерфейс прикладного программирования (API) для фреймворка.

Заметка

Любые внутренние детали реализации вашего фреймворка могут использовать с дефолтным уровнем доступа `internal`, или они могут быть отмечены как `private`, если вы хотите их спрятать от остального внутреннего кода фреймворка. Вам нужно отметить объект как `public`, если вы хотите сделать его частью интерфейса прикладного программирования фреймворка.

Синтаксис уровня доступа

Определите уровень доступа для объекта, установив одно из ключевых слов (`private`, `internal`, `public`) перед вступительным словом объекта (`func`, `property`, `class`, `struct`...):

```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

Если вы не укажете уровень доступа, то он будет `internal` по умолчанию, о чем было поведено в главе “[Уровни доступа](#)”. Это значит, что `SomeInternalClass` и `someInternalConstant` могут быть записаны без явного указания модификатора уровня доступа, и они все равно будут все еще иметь `internal` уровень доступа:

```
class SomeInternalClass {}           //  
неявно internal  
var someInternalConstant = 0         //  
неявно internal
```

Пользовательские типы

Если вы хотите указать явно уровень доступа для пользовательского типа, то сделайте это на этапе определения типа. Новый тип может быть использован там, где позволяет его уровень доступа. К примеру, если вы определите класс с уровнем доступа `private`, то он сможет быть использован как тип свойства или параметр функции или возвращаемый тип в исходном файле, в котором определен этот класс.

Контроль уровня доступа типа так же влияет на уровень доступа для этих членов по умолчанию (его свойств, методов, инициализаторов и сабскриптов). Если вы определяете уровень доступа типа как `private`, то дефолтный уровень доступа его членов так же будет `private`. Если вы определите уровень доступа как `internal` или `public` (или будете использовать дефолтный уровень доступа, без явного указания `internal`), то уровень доступа членов типа по умолчанию будет `internal`.

Заметка

Как уже было сказано выше типы с уровнем доступа `public` по умолчанию имеют члены с уровнем доступа `internal`. Если вы хотите чтобы члены типа имели уровень доступа `public`, то вы должны явно указать его. Такое требование гарантирует, что внешняя часть API - эта та часть, которую вы выбираете сами и исключает тот случай, когда вы можете по ошибке забыть указать `internal` для внутреннего кода по ошибке.

```
public class SomePublicClass { //
    явный public класс
        public var somePublicProperty = 0 //
    явный public член класса
        var someInternalProperty = 0 //
    неявный internal член класса
        private func somePrivateMethod() {} //
    явный private член класса
}
class SomeInternalClass { //
    неявный internal класс
        var someInternalProperty = 0 //
    неявный internal член класса
        private func somePrivateMethod() {} //
    явный private член класса
}
private class SomePrivateClass { //
    явный private класс
        var somePrivateProperty = 0 //
    неявный private член класса
        func somePrivateMethod() {} //
    неявный private член класса
}
```

Кортежи типов

Уровень доступа для кортежей типов имеет самый строгий уровень доступа типа из всех используемых типов в кортеже. Например, если вы скомпонуете кортеж из двух разных типов, один из которых будет иметь уровень доступа как `internal`, другой как `private`, то кортеж будет иметь уровень доступа как `private`.

Заметка

Кортежи типов не имеют отдельного определения в отличие от классов, структур, перечислений или функций. Уровень доступа кортежей типов вычисляется автоматически, когда используется кортеж, и не может быть указан явно.

Типы функций

Уровень доступа для типов функции вычисляется как самый строгий уровень доступа из типов параметров функции и типа возвращаемого значения. Вы должны указывать уровень доступа явно как часть определения функции, если вычисляемый уровень доступа функции не соответствует контекстному по умолчанию.

Пример ниже определяет глобальную функцию `someFunction`, без явного указания уровня доступа самой функции. Вы можете ожидать, что эта функция будет иметь уровень доступа по умолчанию `internal`, но только не в нашем случае. На самом деле функция, которая описана ниже вообще не будет компилироваться:

```
func someFunction() -> (SomeInternalClass,  
SomePrivateClass) {  
    // реализация функции...  
}
```

Возвращаемый тип функцией является кортежем, который составлен из двух пользовательских классов, которые были определены ранее в этом разделе. Один из этих классов был определен как `internal`, другой - как `private`. Таким образом, общий уровень доступа кортежа будет вычислен как `private` (минимальный уровень доступа из всех элементов кортежа).

Из-за того, что уровень доступа функции `private`, то вы должны установить общий уровень доступа как `private` во время определения функции:

```
private func someFunction() ->
    (SomeInternalClass, SomePrivateClass) {
    // реализация функции...
}
```

Это не правильно ставить маркер уровня доступа функции `someFunction` как `internal` или `private`, или использовать уровень доступа по умолчанию, потому что пользователи функции с уровнем доступа `public` или `internal` не смогут получить соответствующий доступ к `private` классу, который используется в качестве части возвращаемого значения функции.

Типы перечислений

Каждый случай в перечислении автоматически получает тот же уровень доступа, что и само перечисление. Вы не можете указать другой уровень доступа для какого-то определенного случая (`case`) перечисления.

В примере ниже перечисление `CompassPoint` имеет явный уровень доступа `public`. Случаи

перечисления `North`, `South`, `East` и `West` так же получают такой же уровень доступа, то есть `public`:

```
public enum CompassPoint {  
    case North  
    case South  
    case East  
    case West  
}
```

Сырые значения и связанные значения

Типы, используемые для любых сырых значений или связанных значений в перечислении, должны иметь как минимум такой же высокий уровень доступа как и перечисление. Вы мне можете использовать тип `private` для типа сырого значения перечисления, которое имеет `internal` уровень доступа.

Вложенные типы

Вложенные типы определенные внутри типа с уровнем доступа `private`, автоматически получают уровень доступа `private`. Вложенные типы внутри `public` типов или `internal` типов, автоматически получают уровень доступа как `internal`. Если вы хотите, чтобы вложенный тип внутри `public` типа имел уровень доступа как `public`, то вам нужно явно указать этот тип самостоятельно.

Уровень доступа класса и подкласса

Вы можете создать подкласс любого класса, который может быть доступен в текущем контексте. Подкласс не может иметь более высокого уровня доступа, чем его суперкласс. Например, вы не можете написать подклассу `public`, если его суперкласс имеет `internal` доступ.

В дополнение вы можете переопределить любой член класса (метод, свойство, инициализатор или сабскрипт), который будет виден в определенном контексте доступа.

Переопределение может сделать член унаследованного класса более доступным, чем его версия суперкласса. В примере ниже класс `A` имеет доступ `public` и имеет метод `someMethod` с уровнем доступа `private`.

Класс `B` является подклассом класса `A`, который имеет урезанный уровень доступа до `internal`. Тем не менее, класс `B` предоставляет переопределение метода `someMethod` с уровнем доступа `internal`, который выше, чем первоначальное определение метода `someMethod`:

```
public class A {  
    private func someMethod() {}  
}  
internal class B: A {  
    override internal func someMethod() {}  
}
```

Это так же справедливо и для членов подкласса, которые вызывают член суперкласса, который имеет более низкий уровень доступа, чем они, до тех пор пока вызов члена суперкласса попадает под допустимый уровень доступа

контекста (то есть, внутри одного исходного файла суперкласс может вызвать член с уровнем доступа `private` или в пределах одного модуля суперкласс может вызвать член с уровнем доступа `internal`):

```
public class A {  
    private func someMethod() {}  
}  
  
internal class B: A {  
    override internal func someMethod() {  
        super.someMethod()  
    }  
}
```

Из-за того, что суперкласс `A` и подкласс `B` определены в одном исходном файле, то будет корректно для реализации `B` записать вызов метода `someMethod` как `super.someMethod()`.

Константы, переменные, свойства и сабскрипт

Константы, переменные, свойства не могут быть более открытыми, чем их тип. Это не правильно писать свойство `public` для `private` типа. Аналогично дело обстоит и с сабскриптом: сабскрипт не может быть более открытым, чем тип индекса или возвращаемый тип.

Если константа, переменная, свойство или сабскрипт используют тип `private`, то они должны быть отмечены ключевым словом `private`:

```
private var privateInstance =  
SomePrivateClass()
```

Геттеры и сеттеры

Геттеры и сеттеры для констант, переменных и сабскриптов автоматически получают тот же уровень доступа как и константа, переменная, свойство или сабскрипт, которому они принадлежат.

Вы можете задать сеттер более низкого уровня доступа чем его соответствующий геттер, для ограничения области `read-write` этой переменной, свойства или сабскрипта. Вы присваиваете более низкий уровень доступа написав `private (set)` или `internal (set)` до вступительного `var` или `subscript`.

Заметка

Это правило применяется как к хранимым свойствам так и к вычисляемым свойствам. Даже если вы не пишете явного геттера и сеттера для хранимого свойства, Swift все еще синтезирует для вас неявный геттер и сеттер, чтобы вы могли получить доступ к хранимым свойствам. Используйте `private (set)` и `internal (set)` для изменения уровня доступа этого синтезированного сеттера в точно такой же форме как и в случае явного сеттера вычисляемого свойства.

Пример ниже определяет структуру `TrackedString`, которая отслеживает число изменений строкового свойства:

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits++
        }
    }
}
```

Структура `TrackedString` определяет хранимое свойство `value` с начальным значением `""` (пустая строка). Структура так же определяет хранимое свойство `numberOfEdits`, которое используется для отслеживания количества изменений значения `value`. Эта модификация отслеживания реализована в наблюдателе `didSet` свойства `value`, которое увеличивает `numberOfEdits` каждый раз, как `value` получает новое значение.

Структура `TrackedString` и свойство `value` не указывают явного уровня доступа, таким образом они оба получают дефолтный уровень доступа `internal`. Однако уровень доступа `numberOfEdits` обозначен как `private(set)`, что означает, что это свойство может устанавливать значение только в пределах этого исходного файла, то есть в пределах структуры `TrackedString`. Геттер свойства все еще имеет дефолтный уровень доступа `internal`, но его сеттер теперь уже `private` к исходному файлу, где определен `TrackedString`. Это позволяет `TrackedString` изменять свойство `numberOfEdits` скрытно, но тем не менее позволяет свойству быть read-only (только для чтения), когда оно

используется в других исходных файлах в пределах того же модуля.

Если вы создаете экземпляр `TrackedString` и изменяете его строковое значение несколько раз, то вы можете увидеть, что свойство `numberOfEdits` изменяется, чтобы соответствовать количеству фактических изменений значения:

```
var stringToEdit = TrackedString()  
stringToEdit.value = "This string will be tracked."  
stringToEdit.value += " This edit will increment  
numberOfEdits."  
stringToEdit.value += " So will this one."  
println("Количество изменений равно  
\"(stringToEdit.numberOfEdits)\")  
// выводит "Количество изменений равно 3"
```

Хотя вы и можете обращаться в текущему значению свойства `numberOfEdits` в пределах одного исходного файла, но вы не можете изменять его из других исходных файлов. Это ограничение защищает детали реализации функциональности `TrackedString`, в то же время обеспечивая удобный доступ к аспекту этой функциональности.

Обратите внимание, что вы можете присвоить явный уровень доступа и к геттеру, и к сеттеру, если это необходимо. Пример ниже показывает версию структуры `TrackedString`, где она определена с явным указанием открытого уровня доступа. Таким образом, элементы структуры (включая свойство `numbersOfEdits`) получают уровень доступа `internal` по умолчанию. Вы можете сделать уровень доступа геттера параметра `numberOfEdits` открытым, а сеттера этого же свойства сделать частным, таким образом вы комбинируете и `public`, и `private(set)` модификаторы уровней доступа:

```
public struct TrackedString {

    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits++
        }
    }
}

public init() {}
}
```

Инициализаторы

Пользовательским инициализаторам может быть присвоен уровень доступа ниже или равный уровню доступа самого типа, который они инициализируют. Единственное исключение составляют [требуемые инициализаторы](#). Требуемый инициализатор может иметь тот же уровень доступа как и класс, которому он принадлежит.

Что же касается параметров функций и методов, типов параметров инициализатора, то они не могут быть более частными, чем собственный уровень доступа инициализатора.

Дефолтные инициализаторы

Как было описано в главе “[Дефолтные инициализаторы](#)”, Swift автоматически предоставляет *дефолтный инициализатор*, который не имеет никаких аргументов, для любой структуры или базового класса, который предоставляет значения по умолчанию для всех своих свойств и который не имеет ни одного собственного инициализатора.

Дефолтный инициализатор имеет тот же уровень доступа, что и тип, который он инициализирует, если только тип не имеет доступа `public`. Для типа, у которого уровень доступа установлен `public`, дефолтный инициализатор имеет уровень доступа `internal`. Если вы хотите, чтобы открытый (`public`) тип был инициализируемым при помощи инициализатора, который не имеет аргументов, когда используется в другом модуле, то вы должны явно указать такой инициализатор как часть определения типа.

Дефолтные почленные инициализаторы для типов структур

Дефолтные почленные инициализаторы для типов структур считаются частными (`private`), если есть свойства, которые имеют уровень доступа как `private`. В противном случае, инициализатор имеет уровень доступа `internal`.

Как и с дефолтным инициализатором выше, если вы хотите открытый тип структуры, который может быть инициализирован при помощи почленного инициализатора, когда используется в другом модуле, то вы должны предоставить открытый почленный инициализатор самостоятельно, как часть определения типа.

Протоколы и уровень доступа

Если вы хотите присвоить явный уровень доступа протоколу, то вы должны указать его во время определения протокола. Это позволяет вам создавать протоколы, которые могут быть приняты только внутри определенного уровня доступа контекста.

Уровень доступа каждого требования в процессе определения протокола устанавливается на тот же уровень, что и сам протокол. Вы не можете установить уровень доступа требований протокола другим, чем поддерживает сам протокол. Это гарантирует, что все требования протокола будут видимы любому типу, который принимает протокол.

Заметка

Если вы определяете открытый протокол, то требования протокола требуют открытого уровня доступа для тех требований, которые они реализуют. Это поведение отличается от поведений других типов, где определение открытого типа предполагает наличие уровня `internal` у элементов этого типа.

Наследование протокола

Если вы определяете новый протокол, который наследует из другого существующего протокола, то новый протокол может иметь не более высокий уровень доступа, чем протокол, от которого он наследует. Вы не можете писать открытый протокол, который наследует из внутреннего протокола, к примеру.

Соответствие протоколу

Тип может соответствовать протоколу с более низким уровнем доступа, чем сам тип. Например, вы можете определить открытый тип, который может быть использован в других модулях, но чье соответствие внутреннему протоколу может быть использовано только внутри модуля, где определен сам внутренний протокол.

Контекст в котором тип соответствует конкретному протоколу является минимумом из доступов протокола и типа. Если тип является открытым (`public`), но протокол, которому он

соответствует является внутренним (`internal`), то соответствие типа этому протоколу будет тоже внутреннего типа (`internal`).

Когда вы пишете или расширяете тип для того, чтобы он соответствовал протоколу, вы должны быть уверены, что реализация этого типа каждому требованию протокола, по крайней мере имеет один и тот же уровень доступа, что и соответствие типа этому протоколу. Например, если тип `public` соответствует протоколу `internal`, то реализация каждого требования протокола должна быть как минимум `internal`.

Заметка

В Swift как и в Objective-C соответствие протоколу является глобальным. И типа не может соответствовать протоколу двумя разными способами в пределах одной программы.

Расширения и уровень доступа

Вы можете расширить класс, структуру или перечисление в любом контексте, в котором класс, структура или перечисление доступны. Любой элемент типа, добавленный в расширение, имеет тот же дефолтный уровень доступа, что и типы, объявленные в исходном типе, будучи расширенными. Например, если вы расширяете тип `public`, то любые новые элементы этого типа, которые вы добавили, будут иметь уровень доступа равный `internal`.

Аналогично вы можете отметить расширение, явно указав модификатор уровня доступа (например, `private extension`), для того чтобы указать новый дефолтный уровень доступа, который будут иметь элементы, определенные в этом расширении. Этот новый уровень доступа может быть переопределен для отдельных элементов расширением.

Добавление соответствия протоколу

Вы не можете предоставлять явный модификатор уровня доступа для расширения, если вы используете расширение для добавления соответствия протоколу. Вместо этого, собственный уровень доступа протокола используется для предоставления дефолтного уровня доступа для каждой реализации требования протокола внутри расширения.

Универсальный код.

Алиасы типов

Уровень доступа для универсального типа или универсальной функции вычисляется как минимальный уровень доступа универсального типа или самой функции и уровень доступа ограничений любого типа ограничений для параметров типа.

Алиасы типов

Любой алиас типа, который вы определяете, рассматривается как отдельный тип для целей и контроля доступа. Алиас типа может иметь уровень доступа такой же или ниже, чем уровень доступа типа, псевдоним которого он создает. Например, алиас с уровнем доступа `private`, может быть алиасом для типа с уровнем доступа `private`, `internal`, `public`, но если у алиаса уровень доступа стоит `public`, то он не может быть алиасом типа, у которого уровень доступа стоит как `internal` или `private`.

Заметка

Это правило так же применимо для алиасов типа связанных типов, используемых для удовлетворения несоответствий протоколу.

Продвинутые операторы

В дополнение к операторам, которые мы рассматривали в главе “[Базовые операторы](#)”, Swift предоставляет нам еще несколько продвинутых операторов, которые позволяют нам проводить более сложные манипуляции со значениями. Они включают в себя побитовые и операторы разрядного смещения, с которыми вы возможно знакомы из языков C или Objective-C.

В отличие от арифметических операторов C, арифметические операторы в Swift не переполняются по умолчанию. Переполнения отслеживаются и выводятся как ошибка. Для того, чтобы этого избежать, вы можете использовать оператор из второго набора арифметических операторов Swift (&+). Все операторы переполнения начинаются с амперсанда (&).

Когда вы определяете ваши собственные структуры, классы или перечисления, то может быть полезным обеспечивать ваши собственные реализации стандартных операторов Swift, для этих пользовательских типов. Swift позволяет создавать адаптивные реализации этих операторов, так что вы можете определить их поведения для каждого конкретного типа, который вы создаете.

Вы не ограничены в предопределенных операторах. Swift дает вам свободу определять ваши собственные префиксные, инфиксные, постфиксные операторы и операторы присваивания, которым вы можете задавать собственный приоритет и ассоциативность значений. Эти операторы могут быть использованы и приняты вашим кодом, как и любой другой предопределенный оператор, вы так же можете расширить уже существующие типы, для того, чтобы они могли поддерживать ваши пользовательские операторы.

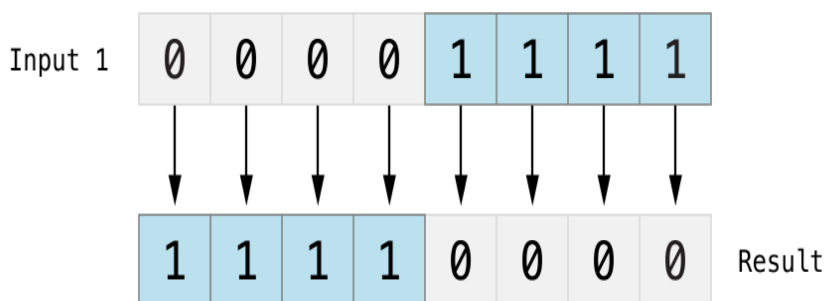
Побитовые операторы

Побитовые операторы позволяют вам манипулировать отдельными битами необработанных данных внутри структуры данных. Они часто используются в низкоуровневом программировании, например программирование графики или создание драйвера для устройства. Побитовые операторы так же могут быть полезны, когда вы работаете с необработанными данными из внешних ресурсов, например, шифрование или дешифрование данных для связи через собственный протокол.

Swift поддерживает все побитовые операторы, которые были основаны в C и которых мы поговорим далее.

Побитовый оператор NOT

Побитовый оператор NOT (\sim) инвертирует все битовые числа:



Побитовый оператор NOT является префиксным оператором и ставится прямо перед значением (без пробела), над которым он оперирует.

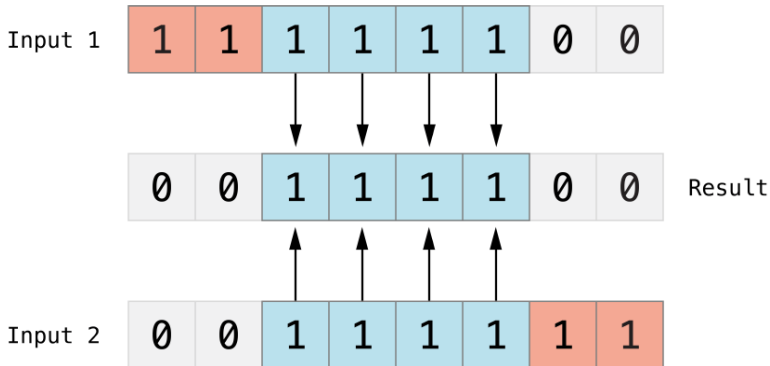
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits // равен
11110000
```

Целые числа типа `UInt8` имеют восемь бит и могут хранить значения от 0 до 255. В этом примере инициализируем число типа `UInt8`, которое имеет бинарное значение `00001111`, которое имеет первые четыре бита равные 0, а вторая четверка битов равна 1. Это эквивалент числа 15.

Далее используем побитовый оператор `NOT` для создания новой константы `invertedBits`, которая равна `initialBits`, но только с перевернутыми битами. То есть теперь все единицы стали нулями, а нули единицами. Значение числа `invertedBits` равно `11110000`, что является эквивалентом 240.

Побитовый оператор AND

Побитовый оператор `AND` (`&`) комбинирует два бита двух чисел. Он возвращает новое число, чье значение битов равно 1, если только оба бита из входящих чисел были равны 1:

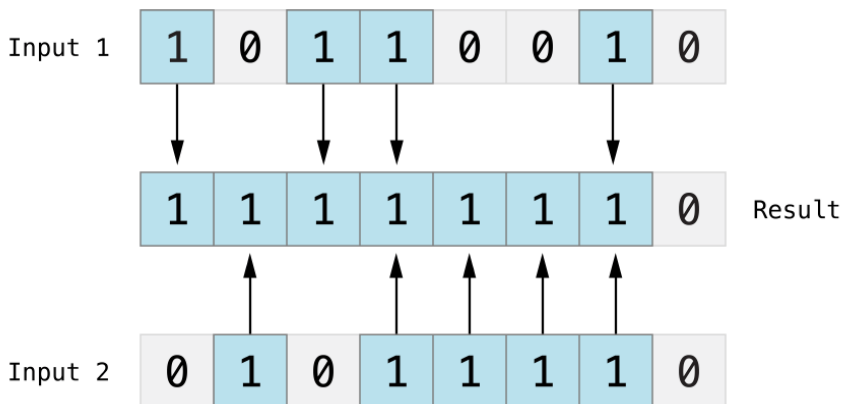


В примере ниже, значения `firstSixBits` и `lastSixBits` имеют четыре бита по середине равными 1. Побитовый оператор AND комбинирует их для создания числа `001111100`, которое равно беззнаковому целому числу 60:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits &
lastSixBits // равен 00111100
```

Побитовый оператор OR

Побитовый оператор OR (|) сравнивает биты двух чисел. Оператор возвращает новое число, чьи биты устанавливаются на 1, если один и пары битов этих двух чисел имеет бит равный 1:

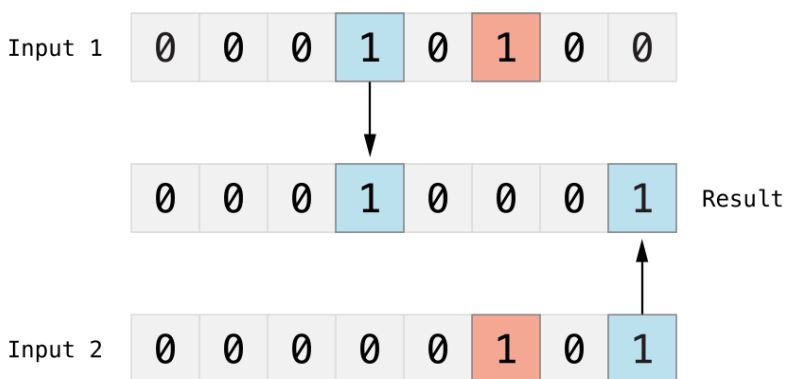


В примере ниже значения `someBits` и `moreBits` имеют разные биты со значениями 1. Побитовый оператор OR комбинирует их для создания числа `11111110`, что равно беззнаковому целому числу 254:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits //
равен 11111110
```

Побитовый оператор XOR

Побитовый оператор XOR или “оператор исключающего OR” (^), который сравнивает биты двух чисел. Оператор возвращает число, которое имеет биты равные 1, когда биты входных чисел разные, и возвращает 0, когда биты одинаковые:



В примере ниже, значения `firstBits` и `otherBits` каждый имеет один бит в том месте, где у другого 0. Побитовый оператор XOR устанавливает оба этих бита в качестве выходного значения. Все остальные биты повторяются, поэтому оператор возвращает 0:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits //
равен 00010001
```

Операторы побитового левого и правого сдвига

Оператор побитового левого сдвига (\ll) и оператор побитового правого сдвига (\gg) двигают все биты числа влево или вправо на определенное количество мест, в зависимости от правил, которые определены ниже.

Побитовые операторы левого и правого сдвига имеют эффект умножения или деления числа на 2. Сдвигая биты целого числа влево на одну позицию, мы получаем удвоенное первоначальное число, в то время как, двигая его вправо на одну позицию, мы получаем первоначальное число поделённое на 2.

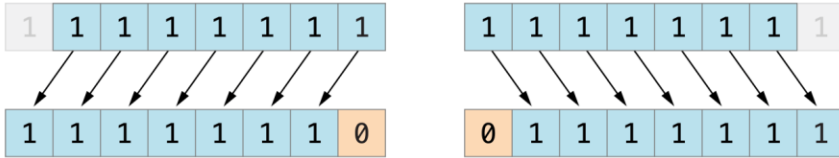
Поведение сдвига для беззнаковых целых чисел

Поведение побитового сдвига имеет следующие правила:

- Существующие биты сдвигаются направо или налево на требуемое число позиций.
- Любые биты, которые вышли за границы числа, отбрасываются.
- На пустующие позиции сдвинутых битов вставляются нули.

Такой подход называется *логическим сдвигом*.

Иллюстрация внизу отображает результат смещения $11111111 \ll 1$ (что означает 11111111 сдвинутые влево на 1), и $11111111 \gg 1$ (что означает 11111111 сдвинутые на 1 вправо). Голубые цифры - сдвинутые, серые - отброшенные, оранжевые - вставленные:



Вот как выглядит побитовый сдвиг в виде Swift кода:

```
let shiftBits: UInt8 = 4    // 00000100
бинарный вид
```

```
shiftBits << 1    // 00001000
shiftBits << 2    // 00010000
shiftBits << 5    // 10000000
shiftBits << 6    // 00000000
shiftBits >> 2    // 00000001
```

Вы можете использовать побитовый сдвиг для шифрования и дешифрования значений внутри других типов данных:

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16
// redComponent равен 0xCC, или 204
let greenComponent = (pink & 0x00FF00) >> 8
// greenComponent равен 0x66, или 102
let blueComponent = pink & 0x0000FF
// blueComponent равен 0x99, или 153
```

Этот пример использует `UInt32`, который называется `pink`, для хранения значение розового цвета из файла CSS. Значение розового цвета `#CC6699`, что записывается в виде шестнадцатеричном представлении Swift как `0xCC6699`. Этот цвет затем раскладывается на его красный (`CC`), зеленый (`66`) и голубой (`99`) компоненты при помощи побитового оператора AND (`&`) и побитового оператора правого сдвига (`>>`).

Красный компонент получен с помощью побитового оператора AND между числами `0xCC6699` и `0xFF0000`. Нули в `0xFF0000` фактически являются “маской” для третьего и четвертого бита `0xCC6699`, тем самым заставляя игнорировать `6699`, и оставляя `0xCC0000` в качестве результата.

После этого число сдвигается на 16 позиций вправо (`>> 16`). Каждая пара символов в шестнадцатеричном числе использует 8 битов, так что сдвиг вправо на 16 позиций преобразует число `0xCC0000` в `0x0000CC`. Это то же самое, что и `0xCC`, которое имеет целое значение равное 204.

Аналогично с зеленым компонентом, который получается путем использования побитового оператора AND между числами `0xCC6699` и `0x00FF00`, который в свою очередь дает нам выходное значение `0x006600`. Это выходное значение затем сдвигается на восемь позиций вправо, давая нам значение `0x66`, что имеет целое значение равное 102.

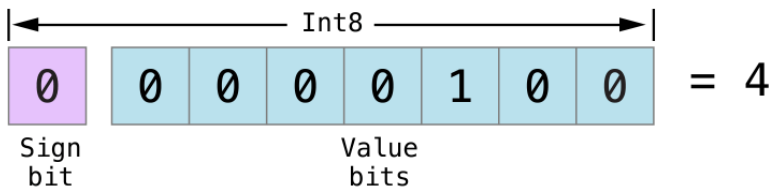
Ну а теперь последний синий компонент, который получается при использовании побитового оператора AND между числами `0xCC6699` и `0x0000FF`, что в свою очередь дает нам выходное значение равное `0x000099`. Таким образом, нам не нужно сдвигать это вправо, так как `0x000099` уже равно `0x99`, что имеет целое значение равное 153.

Поведение побитового сдвига для знаковых целых чисел

Поведение побитового сдвига для знаковых целых чисел более сложное, чем для беззнаковых, из-за того, как они представлены в бинарном виде. (Пример ниже основан на восьми битовом знаковом целом числе для простоты примера, однако этот принцип применим к знаковым целым числам любого размера).

Знаковые целые числа используют первый бит (известный как знаковый бит) для индикации того, является ли число положительным или отрицательным. Значение знакового бита равно 0 свидетельствует о положительном числе, 1 - отрицательном.

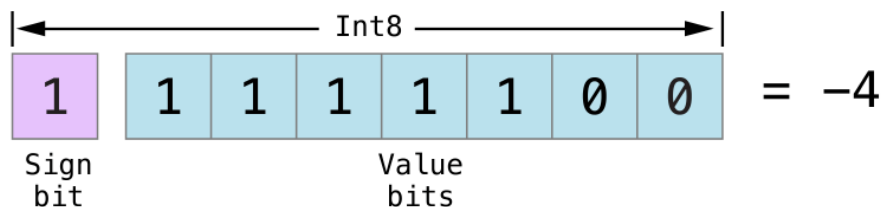
Остальные биты (известные как биты значения) хранят фактическое значение. Положительные числа хранятся в точности так же как и беззнаковые целые числа, считая от 0. Вот как выглядят биты внутри `Int8` для числа 4:



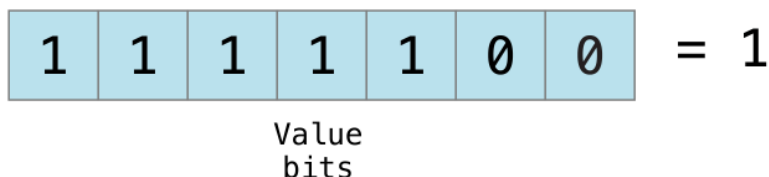
Знаковый бит равен 0 (число положительное), остальные семь битов означают число 4, записанное в бинарной форме.

Однако отрицательные числа хранятся иначе. Они хранятся путем вычитания их абсолютного значения из 2 в степени n , где n - количество битов значения.

Вот как выглядит биты внутри `Int8` для числа -4 :

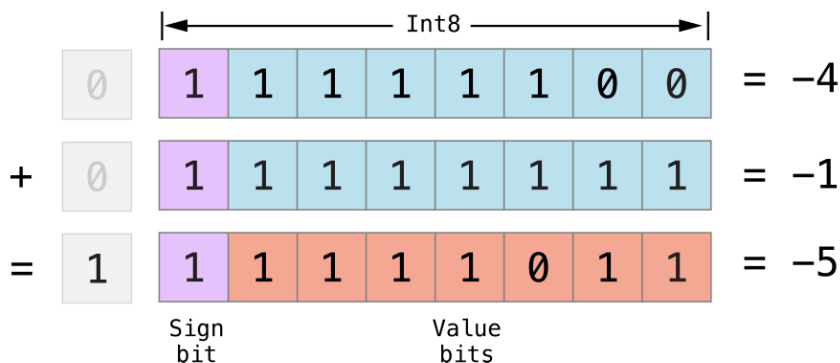


В этот раз, знаковый бит равен 1 (число отрицательное), а остальные семь знаковых бита имеют бинарное значение числа `124` (что означает $128 - 4$):



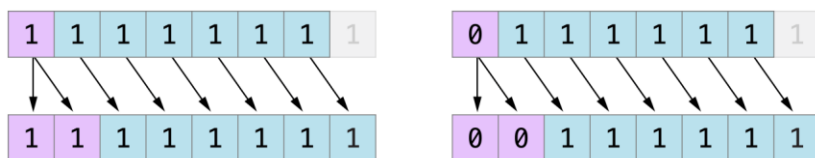
Шифрование отрицательных чисел известно под названием *дополнение до двух*. Это может показаться необычным способом для отображения отрицательных чисел, но в нем есть несколько преимуществ.

Первое. Вы можете добавить -1 к -4 , просто выполняя стандартное сложение всех восьми битов (включая и восьмой бит), и отбрасывая все, что не поместится в ваши восемь бит:



Второе. Представление “дополнения до 2” так же позволяет вам сдвигать биты отрицательных чисел влево и вправо, как в случае с положительными, и все так же умножая их при сдвиге влево или уменьшая их в два раза, при сдвиге на 1 место вправо. Для того чтобы обеспечить такое поведение при движении знаковых чисел вправо, мы должны применить дополнительное правило:

- Когда вы сдвигаете знаковое число вправо, используйте тоже самое правило, что и для беззнаковых чисел, но заполняйте освободившиеся левые биты знаковыми битами, а не нулями.



Эти действия гарантируют вам, что знаковые числа имеют тот же знак, после того как они сдвинуты вправо, и эти действия известны как *арифметический сдвиг*.

Из-за такого способа хранения положительных и отрицательных чисел, сдвиг вправо двигает их значение ближе к нулю. Оставляя знаковый бит тем же самым в течение побитового сдвига, означает, что ваше отрицательное число, так же и остается отрицательным, в то время как его значение так же движется к нулю.

Операторы переполнения

Если вы попытаетесь ввести число в целочисленную константу или переменную, которая не может держать это число, то по умолчанию Swift выдаст сообщение об ошибке, а не будет создавать недействительное значение. Это поведение дает дополнительную безопасность, когда вы работаете с числами, которые слишком велики или слишком малы.

Для примера, целочисленный тип `Int16` может держать любой знаковое целое число от `-32768` и до `32767`. Если вы попытаетесь установить число (константу или переменную) типа `Int16` за границы приведенного диапазона, то вы получите ошибку:

```
var potentialOverflow = Int16.max
// potentialOverflow равняется 32767, что
// является самым большим значением, которое
// может содержаться в Int16
potentialOverflow += 1
// это вызовет ошибку
```

Обеспечивая обработку ошибки, когда значение является слишком большим или слишком маленьким, вы получаете намного большую гибкость, для кодирования краевых условий.

Однако, когда вы специально хотите осуществить условие переполнения, чтобы обрезать количество доступных битов, то вы можете получить именно такое поведение, вместо отчета об ошибке переполнения. Swift предоставляет пять арифметических операторов переполнения, которые помогают перейти к поведению переполнения для целочисленных вычислений. Все эти операторы начинаются с символа амперсанда (&):

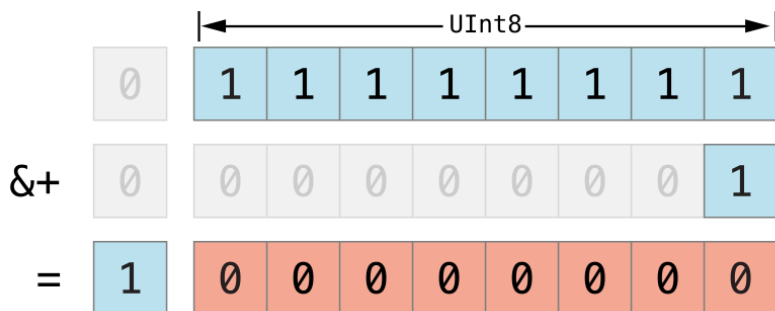
- Оператор переполнения с добавлением (&+)
- Оператор переполнения с вычитанием (&-)
- Оператор переполнения с умножением (&*)
- Оператор переполнения с делением (&/)
- Оператор переполнения с остатком (&%)

Переполнение значения

Ниже приведен пример того, что случится, когда беззнаковое значение позволяет переполнить себя, с использованием оператора (&+):

```
var willOverflow = UInt8.max
// willOverflow равняется 255, что является
// наибольшим числом, которое может держать UInt8
willOverflow = willOverflow &+ 1
// willOverflow теперь равно 0
```

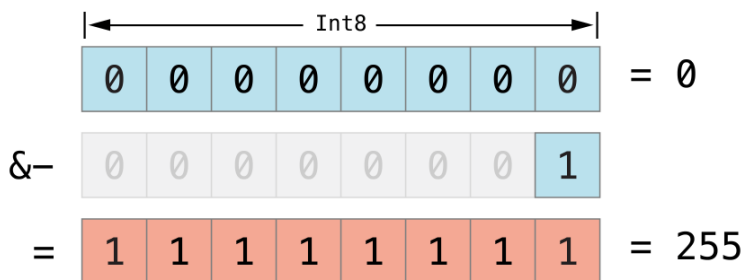
Переменная `willOverflow` инициализирована самым большим числом, которое может держать `UInt8` (255 или в бинарном виде `11111111`). Затем оно увеличивается на 1 при помощи оператора переполнения (&+). Это выталкивает бинарное представление размерности `UInt8`, вызывая тем самым переполнение границ, что отображено на диаграмме ниже. Значение, которое остается в пределах границ значения типа `UInt8` после переполнения и добавления, выглядит как `00000000`, или попросту `0` в десятичной форме:



Оператор недополнения

Числа также могут быть слишком маленькими, чтобы соответствовать определенному типу. Приведем пример.

Самое маленькое значение, которое может держать `UInt8` равно 0 (что отображается как 00000000 в восьмибитной бинарной форме). Если вы из 00000000 вычитите 1, с использованием оператора недополнения, число переполнится в обратную сторону к 11111111, или к 255 в десятичной форме:



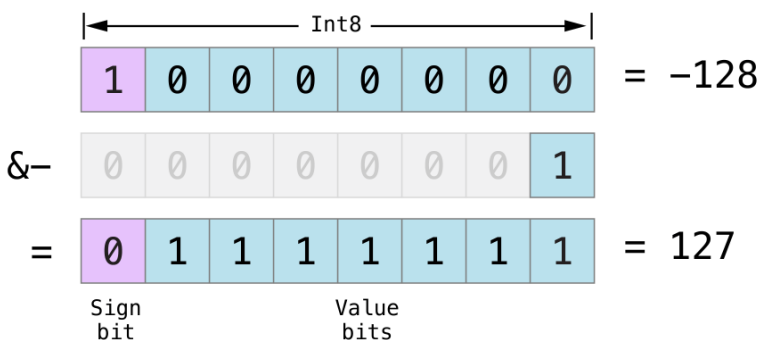
Вот как это будет выглядеть на Swift:

```

var willUnderflow = UInt8.min
// willUnderflow равняется 0, что является
наименьшим значением, которое может держать
UInt8
willUnderflow = willUnderflow &- 1
// willUnderflow теперь равен 255

```

Аналогичное недополнение случается и с знаковыми целыми числами. Все вычитание для знаковых целых чисел проводится как прямое бинарное вычитание с учетом знакового бита, в качестве части вычитаемых чисел, что описано в [“Операторы побитового левого и правого сдвига”](#). Самым маленьким числом, которое может держать `Int8`, является `-128`, что записывается в бинарной форме как `10000000`. Вычитая `1` из этого бинарного числа с оператором недополнения, дает нам значение `01111111`, что переключает наш знаковый бит на противоположный и дает нам положительное `127`, что является самым большим числом, которое может держать `Int8`:



То же самое запишем в Swift коде:

```
var signedUnderflow = Int8.min
// signedUnderflow равняется -128, что
// является самым маленьким числом, которое может
// держать Int8
signedUnderflow = signedUnderflow &- 1
// signedUnderflow теперь равняется 127
```

Конечный результат поведения переполнения и недополнения описан выше и одинакового работает как для знаковых, так и для беззнаковых целых чисел. Переполнение всегда переворачивает значение с самого большого на самое маленькое, недополнение всегда переворачивает самое маленькое число на самое большое.

Деление на нуль

Деление на нуль ($i / 0$), или вычисление остатка от деления на нуль ($i \% 0$) вызывает ошибку:

```
let x = 1
let y = x / 0
```

Однако эти версии операторов переполнения ($\&/$ и $\&\%$) возвращают значение равное нулю, если вы делите на нуль:

```
let x = 1
let y = x &/ 0
// y равен 0
```

Приоритет и ассоциативность

Оператор *приоритета* дает некоторым операторам более высокий приоритет по сравнению с остальными. В выражении сначала применяются эти операторы, затем все остальные.

Оператор *ассоциативности* определяет то, как операторы одного приоритета сгруппированы вместе (или *ассоциированы* друг с другом), то есть либо они сгруппированы слева, либо справа. Думайте об этом как “они связаны с выражением налево” или “они связаны с выражением направо”.

Это важно учитывать приоритет и ассоциативность каждого оператора, когда работаете с порядком, в котором должно считаться выражение. Вот простой пример. Почему данное выражение имеет равенство 4?

2 + 3 * 4 % 5
// это равно 4

Если вы прочитаете это выражение строго слева направо, то вы можете ожидать действия в таком порядке:

- 2 плюс 3 равняется 5
- 5 умножить на 4 равно 20
- остаток от деления 20 на 5 равен 0

Однако, как не крути, правильный ответ равен 4, а не 0. Операторы более высокого приоритета выполняются раньше операторов более низкого приоритета. В Swift, как и в

C, оператор умножения (*) и оператор остатка (%) имеют более высокий приоритет, чем оператор сложения (+). В результате они оба вычисляются раньше, чем вычисляется оператор сложения.

Однако оператор умножения и оператор остатка имеют один и тот же приоритет по отношению друг к другу. Для выяснения точного порядка вычисления вы должны обратиться к их ассоциативности. Операторы умножения и остатка оба ассоциируются с выражением слева от себя. Представляйте это, как будто вы добавили скобки вокруг этих частей выражения, начиная слева:

$2 + ((3 * 4) \% 5)$

$(3 * 4)$ равно 12, значит можно записать:

$2 + (12 \% 5)$

$(12 \% 5)$ равно 2, и значит мы можем записать:

$2 + 2$

Таким образом наш конечный результат равен 4.

Заметка

Правила приоритета и ассоциативности операторов Swift проще и более предсказуемые чем в C или Objective-C. Однако это означает, что они ведут себя не так же как они вели себя в этих C-языках. Будьте внимательны с тем, как ведут себя операторы взаимодействия при переносе кода в Swift.

Операторные функции

Классы и структуры могут предоставлять свои собственные реализации существующих операторов. Действие переопределения оператора известно как перегрузка существующего оператора.

Пример ниже отображает как можно реализовать арифметический оператор сложения (+) для пользовательской структуры. Арифметический оператор сложения является бинарным оператором, потому что он оперирует с двумя операндами, то есть он является инфиксным, потому как вставляется между двумя операндами.

Пример определяет структуру `Vector2D` для двухмерного вектора положения (x, y) , за которой идет операторная функция, которая добавляет друг другу экземпляры структуры `Vector2D`:

```
struct Vector2D {  
    var x = 0.0, y = 0.0  
}  
func + (left: Vector2D, right: Vector2D) ->  
Vector2D {  
    return Vector2D(x: left.x + right.x, y:  
left.y + right.y)  
}
```

Операторная функция определена как глобальная функция с именем функции, которое совпадает с именем оператора, который перегружают (+). Так как арифметический оператор сложения является бинарным оператором, то этот оператор

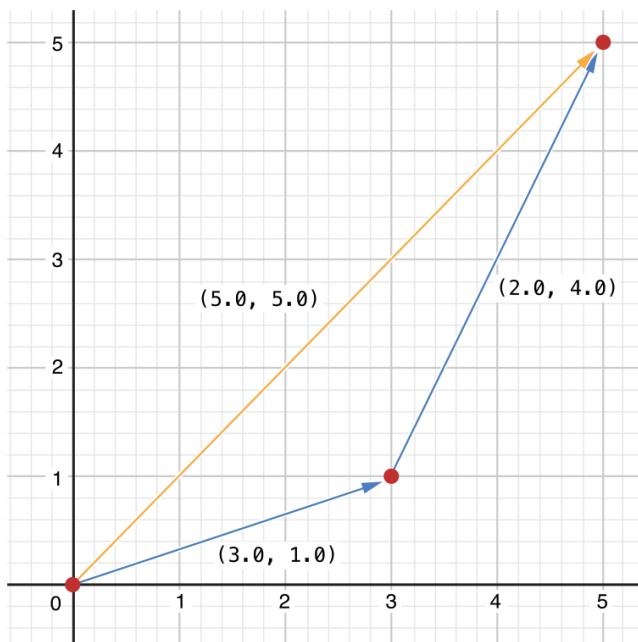
принимает два параметра типа `Vector2D` и возвращает единственное выходное значение, которое тоже имеет тип `Vector2D`.

В этой реализации входные параметры имеют имена `left` и `right`, для отображения экземпляров `Vector2D`, которые будут по левую и по правую сторону от оператора `+`. Функция возвращает новый экземпляр `Vector2D`, `x` и `y` которого инициализированы суммой свойств `x` и `y` из двух экземпляров `Vector2D`, которые были добавлены друг другу.

Функция определена глобальна, в отличие от метода структуры `Vector2D`, так что он может использоваться как инфиксный оператор между существующими экземплярами `Vector2D`:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector является экземпляром
Vector2D, который имеет значения (5.0, 5.0)
```

Этот пример складывает два вектора вместе (3.0, 1.0) и (2.0, 4.0) для создания вектора (5.0, 5.0), который нарисован ниже:



Префиксные и постфиксные операторы

Пример, отображенный выше, демонстрирует пользовательскую реализацию бинарного инфиксного оператора. Классы и структуры так же могут обеспечивать реализацию стандартных *унарных операторов*. Унарные операторы работают с одним операндом. Они бывают *префиксными*, если они предшествуют их операнду (например, `-a`) или *постфиксными*, если они следуют за операндом (например `i++`).

Вы реализуете префиксный или постфиксный унарный оператор при помощи модификаторов `prefix` или `postfix` перед ключевым словом `func`, когда объявляете операторную функцию:

```

prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -
vector.y)
}

```

Пример выше реализует унарный оператор (`-a`) для экземпляров `Vector2D`. Унарный оперного минуса является префиксным оператором, таким образом эта функция должна быть модифицирована при помощи `prefix` модификатора.

Для простых числовых значений оператор унарного минуса конвертирует положительные числа в их негативный эквивалент и наоборот. Соответствующая реализация для экземпляров `Vector2D` проводит операции и на `x`, и на `y` свойствах:

```

let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative - экземпляр Vector2D со значениями
(-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive - экземпляр Vector2D со
значениями (3.0, 4.0)

```

Составные операторы присваивания

Составные операторы присваивания комбинируют оператор присваивания (=) с другим оператором. Например, оператор сложения-присваивания (+=) комбинирует в себе оператор добавления и оператор присваивания. Вы обозначаете левый входной параметр составного оператора как `inout`, потому что именно эта величина и будет изменена напрямую изнутри самой операторной функции.

Пример ниже реализует операторную функцию добавления-присваивания для экземпляров `Vector2D`:

```
func += (inout left: Vector2D, right:
Vector2D) {
    left = left + right
}
```

Так как оператор сложения был определен ранее, то вам не нужно реализовывать процесс сложения здесь. Вместо этого оператор сложения-присваивания использует существующую операторную функцию сложения и использует ее для установки нового значения левому значению, как сумму левого и правого значений:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original теперь имеет значения (4.0, 6.0)
```


Вы можете комбинировать присваивание с `prefix` или `postfix` модификаторами, как это сделано в этой реализации префиксного оператора инкремента (`++a`) для экземпляров `Vector2D`:

```
prefix func ++ (inout vector: Vector2D) ->
Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

Операторная функция префиксного инкремента использует оператор сложения-присваивания, который был определен ранее. Он добавляет `Vector2D` с параметрами `x`, `y` равными `1.0` к `Vector2D`, на который он вызван, после чего возвращает результат:

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement имеет значения (4.0, 5.0)
// afterIncrement так же имеет значения (4.0, 5.0)
```

Заметка

Не такой возможности перегрузить оператор присваивания (`=`). Только составные операторы могут быть перегружены. Тернарный оператор (`a ? b : c`) так же не может быть перегружен.

Операторы эквивалентности

Пользовательские классы и структуры не получают дефолтной реализации эквивалентных операторов, известных как “равен чему-то” оператор (`==`) или “не равен чему-то” (`!=`). Swift не может угадать, что в конкретном случае может означать “равен” для вашего собственного пользовательского типа, потому что значение слова “равен” зависит от роли, которую играет конкретный тип в вашем коде.

Чтобы использовать операторы эквивалентности для проверки эквивалентности вашего собственного пользовательского типа, предоставьте реализацию для этих операторов тем же самым способом, что и для инфиксных операторов:

```
func == (left: Vector2D, right: Vector2D) ->
Bool {
    return (left.x == right.x) && (left.y ==
right.y)
}
func != (left: Vector2D, right: Vector2D) ->
Bool {
    return !(left == right)
}
```

Пример выше реализует оператор “равен чему-то” (`==`) для проверки эквивалентности значений двух экземпляров `Vector2D`. В контексте `Vector2D` имеет смысл считать, что “равно чему-то” означает, что “оба экземпляра имеют одни и те же значения `x` и `y`”, таким образом это является той логикой, которая используется при реализации оператора. Пример так же реализует оператор “не равен чему-то” (`!=`), который просто возвращает обратный результат оператора “равен чему-то”.

Теперь вы можете использовать эти операторы для проверки того, эквивалентны ли экземпляры `Vector2D` друг другу или нет:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("Эти два вектора эквиваленты.")
}
// выводит "Эти два вектора эквиваленты."
```

Пользовательские

операторы

Вы можете объявить и реализовать ваши собственные пользовательские операторы в дополнение к стандартным операторам Swift.

Новые операторы объявляются на глобальном уровне при помощи ключевого слова `operator` и отмечаются модификатором `prefix`, `infix`, `postfix`:

```
prefix operator +++ { }
```

Пример выше определяет новый префиксный оператор `+++`. Этот оператор не имеет никакого значения в Swift, таким образом мы даем ему собственное назначение, которое описано чуть ниже, которое имеет специфический контекст работы с экземплярами `Vector2D`. Для целей этого примера, оператор `+++` рассматривается как новый “префиксный двойной инкрементный” оператор. Он удваивает значения `x` и `y` у экземпляра `Vector2D`, путем добавления вектора самому себе при помощи оператора сложения-присваивания, который мы определили ранее:

```
prefix func +++ (inout vector: Vector2D) ->
Vector2D {
    vector += vector
    return vector
}
```

Эта реализация `+++` очень похожа на реализацию `++` для `Vector2D`, за исключением того факта, что вектор добавляется сам к себе, а не `Vector2D(1.0, 1.0)`:

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled теперь имеет значения (2.0,
8.0)
// afterDoubling так же имеет значения (2.0,
8.0)
```

Приоритет и ассоциативность для пользовательских инфиксных операторов

Пользовательские `infix` операторы так же могут указывать на приоритет и на ассоциативность. Посмотрите главу [“Приоритет и ассоциативность”](#) для объяснения того, как эти две характеристики влияют на взаимодействие инфиксных операторов между собой.

Возможные значения

для `associativity` бывают `left`, `right` или `none`.

Левоассоциативные операторы группируются слева, если записаны рядом с левоассоциативным оператором того же приоритета. Аналогично дело обстоит и с правоассоциативными операторами того же приоритета. Неассоциативные операторы не могут быть записаны рядом с другими операторами того же приоритета.

Дефолтным значением `associativity` является `none`, если не указано другое. Значение приоритета равняется `100`, если не указано другое.

Следующий пример определяет новый инфиксный оператор `+-` левой ассоциативности и с приоритетом `140`:

```
infix operator +- { associativity left
precedence 140 }
func +- (left: Vector2D, right: Vector2D) ->
Vector2D {
    return Vector2D(x: left.x + right.x, y:
left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +-
secondVector
// plusMinusVector является экземпляром
Vector2D со значениями (4.0, -2.0)
```

Этот оператор складывает значения *x* двух векторов и вычитает значение *y* второго вектора из значения *y* первого вектора. Так как этот оператор является эссенцией оператор “сложения”, то приоритет оставим `140`, а ассоциативность `left`, как и у дефолтных инфиксных операторов `+` и `-`.

Заметка

Вы не указываете приоритета, когда определяете префиксный и постфиксные операторы. Однако, если вы воздействуете на операнд сразу двумя операторами (префиксным и постфиксным), то первым будет применен постфиксный оператор.