Дополнительные возможности ООП

Обработка исключений

Исключение представляет событие, которое возникает при выполнении программы и нарушает ее нормальной ход. Например, при передаче файла по сети может оборваться сетевое подключение, и в результате чего может быть сгенерировано исключение. Если исключение не обработано, то программа падает и прекращает свою работу. Поэтому при возникновении исключений их следует обрабатывать.

Для обработки исключений применяется конструкция try..catch..finally. В блок try помещаются те действия, которые потенциально могут вызвать исключение (например, передача файла по сети, открытие файла и т.д.). Блок catch перехватывает возникшее исключение и обрабатывает его. Блок finally выполняет некоторые завершающие действия.

```
try {
    // код, генерирующий исключение
}
catch (e: Exception) {
    // обработка исключения
}
finally {
    // постобработка
}
```

После оператора catch в скобках помещается параметр, который представляет тип исключения. Из этого параметра можно получить информацию о произошедшем исключении.

Блок finally является необязательным, его можно опустить. Блок catch также может отсутствовать, однако обязательно должен быть блок try и как минимум один из блоков: либо catch, либо finally. Также конструкция может содержать несколько блоков catch для обработки каждого типа исключения, которое может возникнуть.

Блок catch выполняется, если только возникло исключение. Блок finally выполняется в любом случае, даже если нет исключения.

Например, при делении на ноль Kotlin генерирует исключение:

```
fun main() {

   try{
       val x : Int = 0
       val z : Int = 0 / x
       println("z = $z")
   }
   catch(e: Exception){
       println("Exception")
```

```
        println(e.message)
    }
}
```

Действие, которое может вызвать исключение, то есть операция деления, помещается в блок try. В блоке catch перехватываем исключение. При этом каждое исключение имеет определенный тип. В данном случае используется общий тип исключений - класс Exception:

```
Exception
```

Если необходимы какие-то завершающие действия, то можно добавить блок finally (например, если при работе с файлом возникает исключение, то в блоке finally можно прописать закрытие файла):

```
try{
    val x : Int = 0
    val z : Int = 0 / x
    println("z = $z")
}
catch(e: Exception){
    println("Exception")
    println(e.message)
}
finally{
    println("Program has been finished")
}
```

В этом случае консольный вывод будет выглядеть следующим образом:

```
Exception
Program has been finished
```

Информация об исключении

Базовый класс исключений - класс Exception предоставляет ряд свойств, которые позволяют получить различную информацию об исключении:

- message: сообщение об исключении
- stackTrace: трассировка стека исключения - набор строк, где было сгенерировано исключение

Из функций класса Exception следует выделить функцию printStackTrace(), которая выводит ту информацию, которая обычно отображается при необработанном исключении.

Применение свойств:

```
fun main() {

    try{
        val x : Int = 0
        val z : Int = 0 / x
        println("z = $z")
    }
    catch(e: Exception){
        println(e.message)
        for(line in e.stackTrace) {
            println("at $line")
        }
    }
}
```

Консольный вывод программы:

```
/ by zero
at AppKt.main(app.kt:5)
at AppKt.main(app.kt)
```

Обработка нескольких исключений

Одна программа, один код может генерировать сразу несколько исключений. Для обработки каждого отдельного типа исключений можно определить отдельный блок catch. Например, при одном исключении мы хотим производить одни действия, при другом - другие.

```
try {
    val nums = arrayOf(1, 2, 3, 4)
    println(nums[6])
}
catch(e:ArrayIndexOutOfBoundsException){
    println("Out of bound of array")
}
catch (e: Exception){
    println(e.message)
}
```

В данном случае при доступе по недействительному индексу в массиве будет генерироваться исключение типа ArrayIndexOutOfBoundsException. С помощью блока catch(e:ArrayIndexOutOfBoundsException). Если в программе будут другие исключения, которые не представляют тип ArrayIndexOutOfBoundsException, то они будут обрабатываться вторым блоком catch, так как Exception - это общий тип, который подходит под все типы исключений. При этом стоит отметить, что в начале обрабатывается исключение более частного типа - ArrayIndexOutOfBoundsException, и только потом - более общего типа Exception.

Оператор throw

Возможно, в каких-то ситуациях мы вручную захотим генерировать исключение. Для генерации исключения применяется оператор throw, после которого указывается объект исключения

Например, в функции проверки возраста мы можем генерировать исключение, если возраст не укладывается в некоторый диапазон:

```
fun main() {

    val checkedAge1 = checkAge(5)
    val checkedAge2 = checkAge(-115)
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110) throw  Exception("Invalid value $age. Age must be
greater than 0 and less than 110")
    println("Age $age is valid")
    return age
}
```

После оператора throw указан объект исключения. Для определения объекта Exception применяется конструктор, который принимает в качестве параметра сообщение об исключении. В данном случае это сообщение о некорректности введенного значения.

И если при вызове функции checkAge() в нее будет передано число меньше 1 или больше 110, то будет сгенерировано исключение. Так, в данном случае консольный вывод будет следующим:

```
Age 5 is valid
Exception in thread "main" java.lang.Exception: Invalid value -115. Age must be
greater than 0 and less than 110
    at AppKt.checkAge(app.kt:7)
    at AppKt.main(app.kt:4)
    at AppKt.main(app.kt)
```

Но опять же поскольку генерируемое здесь исключение не обаботано, то программа при генерации исключения аварийно завершает работу. Чтобы этого не произошло, мы можем обработать генерируемое исключение:

```
fun main() {
    try {
        val checkedAge1 = checkAge(5)
        val checkedAge2 = checkAge(-115)
    }
    catch (e: Exception){
        println(e.message)
    }
}
```

```
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110) throw  Exception("Invalid value $age. Age must be
greater than 0 and less than 110")
    println("Age $age is valid")
    return age
}
```

Возвращение значения

Конструкция try может возвращать значение. Например:

```
fun main() {
    val checkedAge1 = try { checkAge(5) } catch (e: Exception) { null }
    val checkedAge2 = try { checkAge(-125) } catch (e: Exception) { null }
    println(checkedAge1)    // 5
    println(checkedAge2)    // null
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110) throw  Exception("Invalid value $age. Age must be
greater than 0 and less than 110")
    println("Age $age is valid")
    return age
}
```

В данном случае переменная checkedAge1 получает результат функцию checkAge(). Если же произойдет исключение, тогда переменная checkedAge1 получает то значение, которое указано в блоке catch, то есть в данном случае значение null.

При необрабходимости в блок catch можно добавить и другие выражения или возвратить другое значение:

```
fun main() {
    val checkedAge2 = try { checkAge(-125) } catch (e: Exception) {
println(e.message); 18 }
    println(checkedAge2)
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110) throw  Exception("Invalid value $age. Age must be
greater than 0 and less than 110")
    println("Age $age is valid")
    return age
}
```

В данном случае, если будет сгенерировано исключение, то конструкция try выведет исключение и возвратит число 18. Возвращаемое значение указывается после всех остальных инструкций в блоке catch.

Null и nullable-типы

Ключевое слово null представляет специальный литерал, который указывает, что переменная не имеет как такового значения. То есть у нее по сути отсутствует значение.

```
val n = null
println(n)  // null
```

Подобное значение может быть полезно в ряде ситуациях, когда необходимо использовать данные, но при этом точно неизвестно, а есть ли в реальности эти данные. Например, мы получаем данные по сети, данные могут прийти или не прийти. Либо может быть ситуация, когда нам надо явным образом указать, что данные не установлены.

Однако переменным стандартных типов, например, типа Int или String или любых других классов, мы не можем просто взять и присвоить значение null:

```
val n : Int = null   // ! Ошибка, переменная типа Int допускает только числа
```

Мы можем присвоить значение null только переменной, которая представляет тип Nullable. Чтобы превратить обычный тип в тип nullable, достаточно поставить после названия типа вопросительный знак:

```
// val n : Int = null  //! ошибка, Int не допускает значение null
val d : Int? = null // норм, Int? допускает значение null
```

При этом мы можем передавать переменным nullable-типов как значение null, так и конкретные значения, которые укладываются в диапазон значений данного типа:

```
var age : Int? = null
age = 34              // Int? допускает null и числа
var name : String? = null
name = "Tom"         // String? допускает null и строки
```

Nullable-типы могут представлять и создаваемые разработчиком классы:

```
fun main() {

    var bob: Person = Person("Bob")
    // bob = null // ! Ошибка - bob представляет тип Person и не допускает null
    var tom: Person? = Person("Tom")
    tom = null  // норм - tom представляет тип Person? и допускает null
}
class Person(val name: String)
```

В то же время надо понимать, что String? и Int? - это не то же самое, что и String и Int. Nullable типы имеют ряд ограничений:

- Значения nullable-типов нельзя присвоить напрямую переменным, которые не допускают значения null

```
var message : String? = "Hello"
val hello: String = message     // ! Ошибка - hello не допускает значение null
```

- У объектов nullable-типов нельзя вызвать напрямую те же функции и свойства, которые есть у обычных типов

```
var message : String? = "Hello"
// у типа String свойство length возвращает длину строки
println("Message length: ${message.length}")    // ! Ошибка
```

- Нельзя передавать значения nullable-типов в качестве аргумента в функцию, где требуется конкретное значение, которое не может представлять null

Оператор ?:

Одним из преимуществ Kotlin состоит в том, что его система типов позволяет определять проблемы, связанные с использованием null, во время компиляции, а не во время выполнения. Например, возьмем следующий код:

```
var name : String?  = "Tom"
val userName: String = name // ! Ошибка
```

Переменная name хранит строку "Tom". Переменная userName представляет тип String и тоже может хранить строки, но тем не менее напрямую в данном случае мы не можем передать значение из переменной name в userName. В данном случае для компилятора неизвестно, каким значением инициализирована переменная name. Ведь переменная name может содержать и значение null, которое недопустимо для типа String.

В этом случае мы можем использовать оператор ?:, который позволяет предоставить альтернативное значение, если присваиваемое значение равно null:

```
var name : String?  = "Tom"
val userName: String = name ?: "Undefined"  // если name = null, то присваивается
"Undefined"
```

```
var age: Int? = 23
val userAge: Int = age ?:0  // если age равно null, то присваивается число 0
```

Оператор ?: принимает два операнда. Если первый операнд не равен null, то возвращается значение первого операнда. Если первый операнд равен null, то возвращается значение второго операнда.

То есть это все равно, если бы мы написали:

```
var name : String?  = "Tom"
val userName: String
if(name!=null){

    userName = name
}
```

Но оператор ?: позволяет сократить подобную конструкцию.

Оператор ?.

Оператор ?. позволяет объединить проверку значения объекта на null и обратиться к функциям или свойствам этого объекта.

Например, у строк есть свойство length, которое возвращает длину строки в символах. У объекта String? мы просто так не можем обратиться к свойству length, так как если объект String? равен null, то и строки как таковой нет, и соответственно длину строки нельзя определить. И в этом случае мы можем применить оператор ?.:

```
var message : String? = "Hello"
val length: Int? = message?.length
```

Если переменная message вдруг равна null, то переменная length получит значение null. Если переменная name содержит строку, то возвращается длина этой строки. По сути выражение val length: Int? = message?.length эквивалентно следующему коду:

```
val length: Int?
if(message != null)
    length = message.length
else
    length = null
```

С помощью оператора ?. подобным образом можно обращаться к любым свойствам и функциям объекта.

Также в данном случае мы могли совместить оба выше рассмотренных оператора:

```
val message : String?  = "Hello"
val length: Int = message?.length ?:0
```

Теперь переменная length не допускает значения null. И если переменная name не определена, то length получает число 0.

Используя этот оператор, можно создавать цепочки проверок на null:

```
fun main() {

    var tom: Person? = Person("Tom")
    val tomName: String? = tom?.name?.uppercase()
    println(tomName)        // TOM

    var bob: Person? = null
    val bobName: String? = bob?.name?.uppercase()
    println(bobName)        // null

    var sam: Person? = Person(null)
    val samName: String? = sam?.name?.uppercase()
    println(samName)        // null

}
class Person(val name: String?)
```

Здесь класс Person в первичном конструкторе принимает значение типа String?, то есть это можт быть строка, а может быть null.

Допустим, мы хотим получить переданное через конструктор имя пользователя в верхнем регистре (заглавными буквами). Для перевода текста в верхний регистр у класса String есть функция uppercase(). Однако может сложиться ситуация, когда либо объект Person равен null, либо его свойство name (которое представляет тип String?) равно null. И в этом случае перед вызовом функции uppercase() нам надо проверять на null все эти объекты. А оператор ?. позволяет сократить код проверки:

```
val tomName: String? = tom?.name?.uppercase()
```

То есть если tom не равен null, то обращаемся к его свойству name. Далее если name не равен null, то обращаемся к ее функции uppercase(). Если какое-то звено в этой проверки возвратит null, переменная tomName тоже будет равна null.

Но здесь мы также можем избежать финального возвращения null и присвоить значение по умолчанию:

```
val tomName: String = tom?.name?.uppercase() ?: "Undefined"
```

Оператор !!

Оператор !! (not-null assertion operator) принимает один операнд. Если операнд равен null, то генерируется исключение. Если операнд не равен null, то возвращается его значение.

```
fun main() {
    try {
        val name : String?  = "Tom"
        val id: String = name!!
        println(id)
    } catch (e: Exception) { println(e.message)}
}
```

Поскольку данный оператор возвращает объект, который не представляет nullable-тип, то после применения оператора мы можем обратиться к методам и свойствам этого объекта:

```
val name : String?  = null
val length :Int = name!!.length
```

Преобразование типов

Нередко может возникать задача по преобразованию типов, например, чтобы использовать данные одного типа в констексте, где требуются данные другого типа. В этом случае Kotlin представляет ряд возможностей по преобразованию типов.

Встроенные методы преобразования типов Для преобразования данных одного типа в другой можно использовать встроенные следующие функции, которые есть у базовых типов (Int, Long, Double и т.д.) (Конкретный набор функций для разных базовых типов может отличаться.):

- toByte
- toShort
- toInt
- toLong
- toFloat
- toDouble
- toChar

Все эти функции преобразуют данные в тот тип, которые идет после префикса to: toByte.

```
val s: String = "12"
val d: Int = s.toInt()
println(d)
```

В данном случае строка s преобразуется в число d. Просто так передать строку переменной типа Int, мы не можем, несмотря на то, что вроде бы строка и содержит число 12:

```
val d: Int = "12"   // ! Ошибка
```

Однако надо учитывать, что значение не всегда может быть преобразовано к определенному типу. И в этом случае генерируется исключение. Соответственно в таких случаях желательно отлавливать исключение:

```
val s: String = "tom"
try {
    val d: Int = s.toInt()
    println(d)
}
catch(e: Exception){
    println(e.message)
}
```

Smart cast и оператор is

Оператор is позволяет проверить выражение на принадлежность определенному типу данных:

```
значение is тип_данных
```

Этот оператор возвращает true, если значение слева от оператора принадлежит типу, указанному справа от оператора. Если локальная переменная или свойство успешно пройдет проверку на принадлежность определенному типу, то далее нет нужды дополнительно приводить значение к этому типу. Данные преобразования еще называются smart casts или "умные преобразования".

Данный оператор можно применять как к базовым типам, но к собственным классам и интерфейсам, которые находятся в одной и той же иерархии наследования.

```
fun main() {

    val tom = Person("Tom")
    val bob = Employee("Bob", "JetBrains")

    checkEmployment(tom)    // Tom does not have a job
    checkEmployment(bob)    // Bob works in JetBrains
}

fun checkEmployment(person: Person){
    // println("${person.name} works in ${person.company}")    // Ошибка - y
```

```
Person нет свойства company
    if(person is Employee){
        println("${person.name} works in ${person.company}")
    }
    else{
        println("${person.name} does not have a job")
    }
}
open class Person(val name: String)
class Employee(name: String, val company: String): Person(name)
```

Здесь класс Employee наследуется от класса Person. В функции checkEmployment() получаем объект Person. С помощью оператора is проверяем, представляет ли он тип Employee (так как не каждый объект Person может представлять тип Employee). Если он представляет тип Employee, то выводим название его компании, если он не представляет тип Employee, то выводим, сообщение, что он безработный.

Причем даже если значение представляет тип Employee, то до применения оператора is оно тем не менее принадлежит типу Person. И только применение оператора is преобразует значение из типа Person в тип Employee.

Также можно применять другую форму оператора - !is. Она возвращает true, если значение НЕ представляет указанный тип данных:

```
fun main() {

    val tom = Person("Tom")
    val bob = Employee("Bob", "JetBrains")

    checkEmployment(tom)    // Tom does not have a job
    checkEmployment(bob)    // Bob works in JetBrains
}

fun checkEmployment(person: Person){
    // println("${person.name} works in ${person.company}")    // Ошибка - у
Person нет свойства company
    if(person !is Employee){
        println("${person.name} does not have a job")
    }
    else{
        println("${person.name} works in ${person.company}")
    }
}
open class Person(val name: String)
class Employee(name: String, val company: String): Person(name)
```

Однако, что, если свойство company имеет пустую строку, например, val bob = Employee("Bob", ""), то есть фактически компания не указана. А мы хотим выводить компанию, если это свойство имеет какое-

нибудь содержимое. В этом случае мы можем выполнить проверку на длину строку сразу же после применения оператора is:

```
fun checkEmployment(person: Person){
    if(person is Employee && person.company.length > 0){
        println("${person.name} works in ${person.company}")
    }
    else{
        println("${person.name} does not have a job")
    }
}
```

в выражении

```
person.company.length > 0){
```

компилятор уже видит, что person - это объект типа Employee, поэтому позволяет обращаться к его свойствам и функциям.

Если необходимо определить различные действия в зависимости от типа объекта, то удобно использовать конструкцию when:

```
fun identifyPerson(person: Person){
    when(person){
        is Manager -> println("${person.name} is a manager")
        is Employee -&t; println("${person.name} is an employee")
        is Person -> println("${person.name} is just a person")
    }
}

open class Person(val name: String)
open class Employee(name: String, val company: String): Person(name)
class Manager(name: String, company: String): Employee(name, company)
```

Ограничения умных преобразований

Подобные smart-преобразования тем не менее имеют ограничения. Они могут применяться, только если компилятор может гарантировать, что переменная не изменила своего значения в промежутке между проверкой и использованием. Для smart-преобразований действуют следующие правила:

1. smart-преобразования применяются к локальным val-переменным (за исключением делегированных свойств)
2. smart-преобразования применяются к val-свойствам, за исключением свойств с модификатором open (то есть открытых к переопределению в производных классах) или свойств, для которых явным образом определен геттер

3. smart-преобразования применяются к локальным var-переменным (то есть к переменным, определенным в функциях), если переменная не изменяет своего значения в промежутке между проверкой и использованием и не используется в лямбда-выражении, которое изменяет ее, а также не является локальным делегированным свойством

4. к var-свойствам smart-преобразования не применяются

Рассмотрим некоторые случаи. Возьмем последнее правило про var-свойства:

```
fun main() {

    val tom = Person("Tom")

    if(tom.phone is SmartPhone){

        println("SmartPhone: ${tom.phone.name}, OS: ${tom.phone.os}")   // !
Ошибка
    }
    else{
        println("Phone: ${tom.phone.name}")
    }
}
open class Phone(val name: String)
class SmartPhone(name: String, val os: String) : Phone(name)

open class Person(val name: String){
    var phone: Phone = SmartPhone("Pixel 5", "Android")
}
```

Здесь класс Person хранит var-свойство класса Phone, которому присваивается объект класса SmartPhone. Соответственно в выражении:

```
if(tom.phone is SmartPhone){
```

очевидно, что свойство tom.phone представляет класс SmartPhone, однако поскольку это свойство определено с помощью var, то к нему не применяются smart-преобразования. То есть если в данном случае мы заменим var на val, то у нас проблем не возникнет.

Или второе правило - изменим класс Person, определив для свойства геттер:

```
open class Person(val name: String){
    val phone: Phone
        get()  = SmartPhone("Pixel 5", "Android")
}
```

В соответствии с вторым правилом опять же к такому свойству не применяются smart-преобразования, так как оно имеет геттер.

Явные преобразования и оператор as

С помощью оператора as мы можем приводить значения одного типа к другому типу:

```
значение as тип_данных
```

Слева от оператора указывается значение, а справа - тип данных, в которых надо преобразовать значение. Например, преобразуем значение типа String? в тип String:

```
fun main() {

    val hello: String? = "Hello Kotlin"
    val message: String = hello as String
    println(message)
}
```

Здесь переменная hello хранит строку и может быть удачно преобразована в значение типа String. Однако если переменная hello равна null:

```
val hello: String? = null
val message: String = hello as String
println(message)
```

В этом случае преобразование завершится неудачно - ведь значение null нельзя преобразовать в значение типа String. Поэтому будет сгенерировано исключение NullPointerException.

Чтобы избежать генерации исключения мы можем применять более безопасную версию оператора as?, которая в случае неудачи преобразования возвращает null.

```
val hello: String? = null
    val message: String? = hello as? String
    println(message)
```

В данном случае оператор as? возвратит null, так как строку нельзя преобразовать в число.

Также можно применять данный оператор к преобразованиям своих типов:

```
fun main() {

    val tom = Person("Tom")
```

```
    val bob = Employee("Bob", "JetBrains")
    checkCompany(tom)
    checkCompany(bob)
}
fun checkCompany(person: Person){
    val employee = person as? Employee
    if (employee!=null){
        println("${employee.name} works in ${employee.company}")
    }
    else{
        println("${person.name} is not an employee")
    }
}
open class Person(val name: String)
open class Employee(name: String, var company: String): Person(name)
```

Здесь функция checkCompany() принимает объект класса Person и пытается преобразовать его в объект типа Employee, который унаследован от Employee. Но если каждый объект Employee представляет также объект Person (каждый работник является человеком), то не каждый объект Person представляет объект Employee (не каждый человек является работником). И в этом случае чтобы получить значение типа Employee, применяется оператор as?. Если объект person представляет тип Employee, то возвращается объект этого типа, иначе возвращается null. И далее мы можем проверить на значение null и выполнить те или иные действия.

Консольный вывод программы:

```
Tom is not an employee
Bob works in JetBrains
```

Функции расширения

Функции расширения (extension function) позволяют добавить функционал к уже определенным типам. При этом типы могут быть определены где-то в другом месте, например, в стандартной библиотеке.

Функция расширения определяется следующим образом:

```
fun тип.имя_функции(параметры) : возвращаемый_тип{
    тело функции
}
```

По большому счету определение аналогично определению обычной функции за тем исключением, что после слова fun идет название типа, для которого определяется функция, и через точку название функции.

Определим пару функций расширения к стандартным типам Int и String:

```
fun main() {

    val hello: String = "hello world"
    println(hello.wordCount('l'))   // 3
    println(hello.wordCount('o'))   // 2
    println(4.square())                 // 16
    println(6.square())                 // 36
}

fun String.wordCount(c: Char) : Int{
    var count = 0
    for(n in this){
        if(n == c) count++
    }
    return count
}
fun Int.square(): Int{
    return this * this
}
```

Для типа Int определена функция возведения в квадрат. В каждой функции расширения через ключевое слово this мы можем ссылаться на текущий объект того типа, для которого создается функция. Например, в функции:

```
fun Int.square(): Int{
    return this * this
}
```

Через this обращаемся к тому объекту, для которого будет вызывться функция. И затем вы можем вызвать ее следующим образом:

```
4.square()  // 16
```

Для типа String определена функция wordCount, которая подсчитывает, сколько встречается определенный символ в строке.

Следует учитывать, что в функциях расширения мы можем обращаться к любым общедоступным свойствам и методам объекта, однако не можем обращаться к свойствам и методам с модификаторами private и protected.

Также следует учитывать, что функции расширения не переопределяют функции, которые уже определены в классе. Если функция расширения имеет ту же сигнатуру, что и уже имеющаяся функция класса, то компилятор просто будет игнорировать подобную функцию расширения.

Перегрузка операторов

Kotlin позволяет определить для типов ряд встроенных операторов. Для определения оператора для типа определяется функция с ключевым словом operator:

```
operator fun название_оператора([параметры_оператора]) : возвращаемый_тип{
    // действия функции оператора
}
```

После ключевого слова fun идет название оператора и далее скобки. Если оператор бинарный, то в скобках указывается параметр оператора. После скобок через двоеточие указывается возвращаемый тип.

Рассмотрим простейший пример. Допустим, у нас есть класс Counter, который представляет некоторый счетчик:

```
class Counter(var value: Int)
```

Свойство value собственно хранит значение счетчика.

И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства value, используя стандартные операции сравнения и сложения:

```
val counter1 = Counter(5)
val counter2 = Counter(7)

val result = counter1 > counter2;
val counter3: Counter = counter1 + counter2;
```

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. Эти операции могут использоваться для ряда встроенных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты классов, которые создаются непосредственно разработчиком, компилятор не знает. И для этого нам надо выполнить перегрузку нужных нам операторов:

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(7)

    val counter1IsGreater = counter1 > counter2
    val counter3: Counter = counter1 + counter2

    println(counter1IsGreater)  // false
    println(counter3.value)     // 12
}

class Counter(var value: Int){
```

```
    operator fun compareTo(counter: Counter) : Int{
        return this.value - counter.value
    }
    operator fun plus(counter: Counter): Counter {
        return Counter(this.value + counter.value)
    }
}
```

Переопределение операторов предполагает переопределение соответствующих этим операторам функций. Например, операция сравнения

```
counter1 > counter2
```

транслируется в функцию

```
counter1.compareTo(counter2) > 0
```

То есть, если левый операнд (counter1) операции больше чем правый операнд (counter2), то функция оператора должна возвращать число больше 0. И в данном случае мы можем просто вычесть из counter1.value значение counter2.value, чтобы определить, больше ли counter1 чем counter2:

```
operator fun compareTo(counter: Counter) : Int{
    return this.value - counter.value
}
```

Оператор сложения + транслируется в функцию plus(). Параметр этой функции представляет правый операнд операции. Левый операнд доступен через ключевое слово this:

```
operator fun plus(counter: Counter): Counter {
    return Counter(this.value + counter.value)
}
```

Возвращаемое значение операции сложения может быть любым, но в данном случае мы предполагаем, что это будет также объект Counter.

Операторы могут быть определены как в виде функций класса, так и в виде функций расширений. А это значит, что мы можем переопределить операторы даже для встроенных типов:

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(3)
```

```
    val counter3: Counter = counter1 + counter2

    val counter4: Counter = 33 + counter1

    println(counter3.value)     // 8
    println(counter4.value)     // 38
}

class Counter(val value: Int)

operator fun Counter.plus(counter: Counter): Counter {
    return Counter(this.value + counter.value)
}

operator fun Int.plus(counter: Counter): Counter {
    return Counter(this + counter.value)
}
```

Здесь для класса Counter определена операция сложения с помощью функции расширения. Но кроме того, здесь также определен оператор сложения и для встроенного типа Int - в данном случае в качестве правого операнда будет передаваться объект Counter и результатом операции также будет объект Counter:

```
operator fun Int.plus(counter: Counter): Counter {
    return Counter(this + counter.value)
}
```

Благодаря этому мы также сможем складывать объекты Int и Counter:

```
val counter4: Counter = 33 + counter1
```

Рассмотрим, какие операторы мы можем переопределить

Унарные операторы

Операции	Транслируются в
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

Например, переопределение операции унарного минуса

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = -counter1

    println(counter2.value)     // -5
}

class Counter(var value: Int){

    operator fun unaryMinus(): Counter{
        return Counter(-value)
    }
}
```

Инкремент/декремент

Операции	Транслируются в
++a / a++	a.inc()
--a / a--	a.dec()

Следует отметить, что эти операторы не должны именять текущий объект, к которому применяется операция инкремента/декремента, а должны возвращать новый объект этого типа. Например:

```
fun main(){
    var counter1 = Counter(5)
    var counter2 = counter1++

    println(counter1.value)     // 6
    println(counter2.value)     // 5

    var counter3 = ++counter1
    println(counter1.value)     // 7
    println(counter3.value)     // 7
}

class Counter(var value: Int){

    operator fun inc(): Counter{
        return Counter(value + 1)
    }
    operator fun dec(): Counter{
        return Counter(value - 1)
    }
}
```

При операции постфиксного инкремента (counter1++) компилятор сначала создает временную переменную, в которую сохраняет текущий объект. Затем текущий объект замещает значением,

полученным из функции inc(). В качестве результата операции возвращается значение временной переменной:

```
var counter1 = Counter(5)
var counter2 = counter1++   // counter2 получает старое значение объекта counter1
из временной переменной

println(counter1.value)     // 6
println(counter2.value)     // 5
```

При префиксном инкременте (++counter1) компилятор возвращает новое значение, полученное из функции inc():

```
println(counter1.value)     // 6

var counter3 = ++counter1
println(counter1.value)     // 7
println(counter3.value)     // 7
```

Те же правила касаются и префиксного/постфиксного декремента.

Бинарные арифметические операторы

Операции	Транслируются в
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Пример операции сложения:

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(25)
    val counter3: Counter = counter1 + counter2
    println(counter3.value)     // 30

    val counter4: Counter = counter1 + 4
    println(counter4.value)     // 9
}
```

```
class Counter(var value: Int){

    operator fun plus(counter: Counter): Counter{
        return Counter(value + counter.value)
    }
    operator fun plus(number: Int): Counter{
        return Counter(value + number)
    }
}
```

Здесь реализовано две версии оператора. Первая складывает объект Counter с другим объектом Counter. Вторая версия складывает объект Counter с целым числом.

Операторы сравнения

Операции	Транслируются в
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

Для первых двух операторов (== и !=) необходимо переопределить функцию equals(value: Any?), которая должна иметь один параметр типа Any? и возвращать значение Boolean

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(5)
    val counter3 = Counter(7)

    println(counter1 == counter2)   // true
    println(counter1 == counter3)   // false
}
class Counter(var value: Int){

    override operator fun equals(counter: Any?): Boolean{
        if(counter is Counter) return this.value == counter.value
        return false
    }
}
```

В данном случае два объекта Counter равны, если равны значения их свойств value.

Для остальных операций сравнения реализуется функция compareTo(), которая должна возвращать число - значение типа Int. Если левый операнд больше правого, то возвращает число больше 0, если меньше, то возвращается число меньше 0. Если операнды равны, возвращается 0.

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(4)
    val counter3 = Counter(7)

    println(counter1 > counter2)   // true
    println(counter1 > counter3)   // false
    println(counter1 > 1)   // true
    println(counter1 > 56)   // false
}
class Counter(var value: Int){

    operator fun compareTo(counter: Counter): Int{
        return this.value - counter.value
    }
    operator fun compareTo(number: Int): Int{
        return this.value - number
    }
}
```

Операторы присвоения

Операции	Транслируются в
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

Все функции операторов присвоения должны возвращать значение типа Unit:

```
fun main(){
    val counter1 = Counter(5)
    val counter2 = Counter(4)
    val counter3 = Counter(7)

    counter1 += counter2
    println(counter1.value)   // 9

    counter3 += 3
    println(counter3.value)   // 10
}
```

```
class Counter(var value: Int){

    operator fun plusAssign(counter: Counter){
        this.value += counter.value
    }
    operator fun plusAssign(number: Int){
        this.value += number
    }
}
```

Оператор in

Операции	Транслируются в
a in b	b.contains(a)
a !in b	!b.contains(a)

Как правило, под операндом b подразумевается некоторая коллекция, которая в качестве элемента может иметь операнд a:

```
fun main(){
    val tom = Person("Tom")
    val mike = Person("Mike")
    val bob = "Bob"
    val alice = "Alice"
    val jetBrains = Company(arrayOf(Person("Tom"), Person("Bob"), Person("Sam")))

    val tomInJetBrains = tom in jetBrains
    println(tomInJetBrains) // true

    val mikeInJetBrains = mike in jetBrains
    println(mikeInJetBrains) // false

    val bobInJetBrains = bob in jetBrains
    println(bobInJetBrains) // true

    val aliceInJetBrains = alice in jetBrains
    println(aliceInJetBrains) // false
}
class Person(val name:String)
class Company(private val personal: Array<Person>){
    operator fun contains(person: Person): Boolean{
        for (employee in personal) {
            if(employee.name == person.name) return true
        }
        return false
    }
    operator fun contains(personName: String): Boolean{
        for (employee in personal) {
            if(employee.name == personName) return true
```

```
        }
        return false
    }
}
```

В данном случае класс компании Company хранит в свойстве personal штат сотрудников в виде массива объекта Person. И класс Company реализует две версии оператора in: одна версия для проверки наличия объекта Person в массиве, другая для проверки наличия объект Person, имя которого соответствует строке - условному имени сотрудника.

Операции	Транслируются в
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

Применим оператор доступа по индексу

```
fun main(){
    val jetBrains = Company(arrayOf(Person("Tom"), Person("Bob"), Person("Sam")))

    // получаем пользователей
    val firstPerson: Person? = jetBrains[0]
    println(firstPerson?.name) // Tom

    val fifthPerson: Person? = jetBrains[5]
    println(fifthPerson?.name) // null

    // устанавливаем пользователей
    jetBrains[0] = Person("Mike")
    println(jetBrains[0]?.name) // Mike
}
class Person(val name:String)
class Company(private val personal: Array<Person>){
    operator fun set(index: Int, person: Person){
        // если индекс есть в массиве personal
        if(index in personal.indices)
            personal[index] = person    // то переустанавливаем значение в массиве
    }
    operator fun get(index: Int): Person?{
        // если индекс есть в массиве personal
        if(index in personal.indices)
            return personal[index] // то возвращаем значение из массива
        return null // иначе возвращаем null
```

```
    }
}
```

В данном случае с помощью функций get/set опосредуется доступ к массиву personal в рамках объекта People. А благодаря операторам-индексаторам мы сможем использовать объект People как массив.

Операторы вызова

Операторы вызова в виде реализации функции invoke() применяются для выполнения объекта на манер функции:

Операции	Транслируются в
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

Применение:

```
fun main(){
    val message=Message()
    message("Hello Kotlin")     // Message text: Hello Kotlin
}
class Message(){
    operator fun invoke(text: String) {
        println("Message text: $text")
    }
}
```

Подобный оператор удобно использовать в качестве фабрики объекта:

```
fun main(){
    val message1=Message("Hello")
    val message2 = message1("World")
    val message3 = message2("!!!")
    println(message3.text)  // Hello World !!!
}
class Message(val text: String){
    operator fun invoke(addition: String) : Message {
        return Message("$text $addition")
    }
}
```

Делегированные свойства

Делегированные свойства позволяют делегировать получение или присвоение их значения во вне - другому классу. Это позволяет нам добавить некоторую дополнительную логику при операции со свойствами, например, логгирование, какую-то предобработку и т.д.

Формальный синтаксис делегированного свойства:

```
val/var имя_свойства: тип_данных by выражение
```

После типа данных свойства идет ключевое слово by, после которого указывается выражение. Выражение представляет класс, который условно называется делегатом. Делегаты свойств могут не применять никаких интерфейсов, однако они должны предоставлять функции getValue() и setValue(). А выполнение методов доступа get() и set(), которые есть у свойства, делегируется функциям getValue() и setValue() класса делегата.

Стоит отметить, что мы не можем объявлять делегированные свойства в первичном конструкторе.

Делегированные свойства для чтения

Для свойств только для чтения (то есть val-свойств), делегат должен предоставлять функцию getValue(), которая принимает следующие параметры:

- thisRef: должен представлять тот же тип, что и свойство, к которому применяется делегат. Это может быть и родительский тип.
- property: должен представлять тот же тип KProperty<*> или его родительский тип

При этом функция getValue() должна возвращать результат того же типа, что и тип свойства (либо его производного типа).

Рассмотрим на примере:

```
import kotlin.reflect.KProperty

fun main() {

    val tom = Person()
    println(tom.name)   // Tom

    val bob = Person()
    println(bob.name)   // Tom
}
class Person{
    val name: String by LoggerDelegate()
}
class LoggerDelegate {
    operator fun getValue(thisRef: Person, property: KProperty<*>): String {
        println("Запрошено свойство: ${property.name}")
        return "Tom"
```

```
    }
}
```

Здесь класс Person определяет свойство name, которое является делегированным - оно делегирует операцию получения значения функции getValue() класса LoggerDelegate.

Поскольку свойство определено в классе Person, то первый параметр функции getValue() представляет тип Person. Благодаря этому мы можем выудить из этого параметра какую-то дополнительную информацию об объекте, если она необходима.

Поскольку свойство представляет тип String, то функция также возвращает значение типа String - это то значение, которое будет возвращаться самим свойством name. В данном случае возвращается строка "Tom". То есть при каждом обращении к свойству name объекта Person будет возвращаться строка "Tom".

Теперь немного видоизменим пример:

```
import kotlin.reflect.KProperty
fun main() {
    val tom = Person("Tom")
    println(tom.name)

    val bob = Person("Bob")
    println(bob.name)
}
class Person(_name: String){
    val name: String by LoggerDelegate(_name)
}
class LoggerDelegate(val personName: String) {
    operator fun getValue(thisRef: Person, property: KProperty<*>): String {
        println("Запрошено свойство ${property.name}")
        println("Устанавливаемое значение: $personName")
        return personName
    }
}
```

Теперь первичный конструктор Person принимает устанавливаемое значение для свойства name. Далее оно передается в конструктор классу LoggerDelegate, который использует его для логгирования на консоль. И в конце возвращает его в качестве значения свойства name.

Изменяемые свойства

Для изменяемых свойств (var-свойств) делегат должен также предоставить функцию setValue(), которая принимает следующие параметры:

- thisRef: должен представлять тот же тип, что и свойство, к которому применяется делегат. Это может быть и родительский тип.

- property: должен представлять тот же тип KProperty<*> или его родительский тип
- value: должен представлять тот же тип, что и свойство, или его родительский тип

Рассмотрим на примере:

```
import kotlin.reflect.KProperty

fun main() {

    val tom = Person("Tom", 37)
    println(tom.age)    //37
    tom.age = 38
    println(tom.age)    //38
    tom.age = -139
    println(tom.age)    //38

}
class Person(val name: String, _age: Int){
    var age: Int by LoggerDelegate(_age)
}
class LoggerDelegate(private var personAge: Int) {
    operator fun getValue(thisRef: Person, property: KProperty<*>): Int{
        return personAge
    }
    operator fun setValue(thisRef: Person, property: KProperty<*>, value: Int){
        println("Устанавливаемое значение: $value")
        if(value > 0 && value < 110) personAge = value
    }
}
```

Здесь класс Person определяет делегированное свойство age. Оно делегирует установку и получение значения классу LoggerDelegate и его функциям getValue() и setValue(). Само значение сохраняется в свойстве personAge класса LoggerDelegate. Функция getValue() просто возвращает значение это свойства.

Функция setValue() с помощью третьего параметра - value, которое представляет тот же тип, что и свойство - тип Int, получает устанавливаемое значение. И если оно соответствует некоторому диапазону, то передает в свойство personAge.

Консольный вывод программы:

```
37
Устанавливаемое значение: 38
38
Устанавливаемое значение: -139
38
```

Scope-функции

Scope-функции (можно перевести как "функции контекста" или "функции области видимости") позволяют выполнить для некоторого объекта некоторый код в виде лямбда-выражение. При вызове подобной функции, создается временная область видимости. В этой области видимости можно обращаться к объекту без использования его имени.

В Kotlin есть пять подобных функций: let, run, with, apply и also. Эти функции похожи по своему действию и различаются прежде всего по параметрам и возвращаемым результатам

let

Лямбда-выражение в функции let в качестве параметра it получает объект, для которого вызывается функция. Возвращаемый результат функции let представляет результат данного лямбда-выражения.

```
inline fun <T, R> T.let(block: (T) -> R): R
```

Распространенным сценарием, где применяется данная функция, представляет проверка на null:

```
fun main() {

    val sam = Person("Sam", "sam@gmail.com")
    sam.email?.let{ println("Email: $it") }     // Email: sam@gmail.com
    // аналог без функции let
    //if(sam.email!=null) println("Email:${sam.email}")

    val tom = Person("Tom", null)
    tom.email?.let{ println("Email: $it") } // функция let не будет выполняться

}
data class Person(val name: String, val email: String?)
```

Допустим, мы хотим вывести значение свойства Email объекта Person. Но это свойство может хранить значение null (например, если электронный адрес у пользователя не установлен). С помощью выражения sam.email?. проверяем значение свойства email на null. Если email не равно null, то для строки в свойстве email вызывается функция let, которая создает временную область видимости и передает в нее значение свойства email через параметр it. Если же свойство email равно null, то функция let не вызывается.

Если лямбда-выражение вызывает лишь одну функцию, в которую передается параметр it, то можно сократить вызов - указать после оператора :: название вызываемой функции:

```
fun main() {

    val sam = Person("Sam", "sam@gmail.com")
    sam.email?.let(::println)   // sam@gmail.com

    val tom = Person("Tom", "tom@gmail.com")
    tom.email?.let(::printEmail)    // Email: tom@gmail.com
```

```
}

fun printEmail(email: String){
    println("Email: $email")
}
data class Person(val name: String, var email: String?)
```

with

Лямбда-выражение в функции with в качестве параметра this получает объект, для которого вызывается функция. Возвращаемый результат функции with представляет результат данного лямбда-выражения.

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R
```

Функция with принимает объект, для которого надо выполнить блок кода, в качестве параметра. Далее в самом блоке кода мы можем обращаться к общедоступным свойствам и методам объекта без его имени.

Обычно функция with() применяется, когда надо выполнить над объектом набор операций как единое целое:

```
fun main() {

    val tom = Person("Tom", null)
    val emailOfTom = with(tom) {
        if(email==null)
            email = "${name.lowercase()}@gmail.com"
        email
    }
    println(emailOfTom) // tom@gmail.com
}
data class Person(val name: String, var email: String?)
```

В данном случае функция with получает объект tom, поверяет его свойство email - если оно равно null, то устанавливает его на основе его имени. В качестве результата функции возвращается значение свойства email.

run

Лямбда-выражение в функции run в качестве параметра this получает объект, для которого вызывается функция. Возвращаемый результат функции run представляет результат данного лямбда-выражения.

```
inline fun <T, R> T.run(block: T.() -> R): R
```

Функция run похожа на функцию with за тем исключением, что run реализована как функция расширения. Функция run также принимает объект, для которого надо выполнить блок кода, в качестве параметра. Далее в самом блоке кода мы можем обращаться к общедоступным свойствам и методам объекта без его имени.

```
fun main() {

    val tom = Person("Tom", null)
    val emailOfTom = tom.run {
        if(email==null)
            email = "${name.lowercase()}@gmail.com"
        email
    }
    println(emailOfTom) // tom@gmail.com
}
data class Person(val name: String, var email: String?)
```

В данном случае функция run выполняет действия, аналогичные примеру с функцией with.

Реализация run как функции расширения упрощает проверку на null:

```
fun main() {

    val tom = Person("Tom", null)
    val validationResult = tom.email?.run {"valid"} ?: "undefined"
    println(validationResult) // undefined
}
data class Person(val name: String, var email: String?)
```

Также есть другая разновидность функции run(), которая просто позволяет выполнить некоторое лямбда-выражение:

```
fun main() {

    val randomText = run{ "hello world"}
    println(randomText)  // hello world

    run{ println("run function")}    // run function
}
```

apply

Лямбда-выражение в функции apply в качестве параметра this получает объект, для которого вызывается функция. Возвращаемым результатом функции фактически является передаваемый в функцию объект для которого выполняется функция.

```
inline fun T.apply(block: T.() -> Unit): T
```

Например:

```
fun main() {

    val tom = Person("Tom", null)
    tom.apply {
        if(email==null)
            email = "${name.lowercase()}@gmail.com"
    }
    println(tom.email) // tom@gmail.com
}
data class Person(val name: String, var email: String?)
```

В данном случае, как и в примерах с функциями with и run, проверяем значение свойства email, и если оно равно null, устанавливаем его, используя свойство name.

Распространенным сценарием применения функции apply() является построение объекта в виде реализации вариации паттерна "Строитель":

```
fun main() {

    val bob = Employee()
    bob.name("Bob")
    bob.age(26)
    bob.company("JetBrains")
    println("${bob.name} (${bob.age}) - ${bob.company}") // Bob (26) - JetBrains
}

data class Employee(var name: String = "", var age: Int = 0, var company: String =
"") {
    fun age(_age: Int): Employee = apply { age = _age }
    fun name(_name: String): Employee = apply { name = _name }
    fun company(_company: String): Employee = apply { company = _company }
}
```

В данном случае класс Employee содержит три метода, которые устанавливают каждое из свойств класса. Причем каждый подобный метод вызывает функцию apply(), которое передает значение соответствующему свойству и возвращает текущий объект Employee.

also

Лямбда-выражение в функции also в качестве параметра it получает объект, для которого вызывается функция. Возвращаемым результатом функции фактически является передаваемый в функцию объект для которого выполняется функция.

```
inline fun T.also(block: (T) -> Unit): T
```

Эта функция аналогична функции apply за тем исключением, что внутри also объект, над которым выполняется блок кода, доступен через параметр it:

```
fun main() {

    val tom = Person("Tom", null)
    tom.also {
        if(it.email==null)
            it.email = "${it.name.lowercase()}@gmail.com"
    }
    println(tom.email) // tom@gmail.com
}
data class Person(val name: String, var email: String?)
```

Инфиксная нотация

Инфиксная нотация представляет помещение оператора или функции перед операндами или аргументами. Для определения инфиксной функции вначале ее определения указывается ключевое слово infix:

```
infix fun название_функции(параметр: тип_параметра): тип_возвращаемого_значения{
    // действия функции
}
```

Инфиксная функция должна принимать только один параметр. При этом параметр не должен иметь значение по умолчанию и не должен представлять неопределенный набор значений.

Есть два способа определения инфиксной функции: либо внутри класса, либо как функции расширения.

Определим вначале внутри класса:

```
fun main() {

    val acc = Account(1000)
    acc put 150
    // равноценно вызову
    acc.put(150)
    acc.printSum()  // 1300
}
class Account(var sum: Int) {

    infix fun put(amount: Int){
        sum = sum + amount
```

```
    }
    fun printSum() = println(sum)
}
```

Здесь определен класс Account - класс банковского счета, который через конструктор принимает начальную сумму на счете. С помощью инфиксной функции put() определяем добавление на счет суммы, переданной через параметр функции.

Вызов функции выглядит следующим образом:

```
acc put 150
```

Первый параметр (здесь переменная acc) представляет объект, который вызывает функцию. А второй параметр - данные, которые непосредственно будут передаваться инфиксной функции через ее параметр. То есть данный вызов фактически аналогичен вызову:

```
acc.put(150)
```

Также инфиксная функция может определяться как функция расширения. Например, перепишем выше использованную функцию put() в виде функции расширения:

```
fun main() {

    val acc = Account(1000)
    acc put 150
    acc.put(150)
    acc.printSum()  // 1300
}
infix fun Account.put(amount: Int){
    this.sum = this.sum + amount
}
class Account(var sum: Int) {
    fun printSum() = println(sum)
}
```

Стоит отметить, что функция расширения в отличие от функции внутри класса имеет доступ только тем свойствам, которые являются публичными.

Однако использование функций расширений позволяет добавить инфиксные функции к уже существующим типам. Например, определим инфиксную функцию для подсчета частоты символа в строке:

```
fun main() {
```

```
    val hello = "hello world"
    val lCount = hello wordCount 'l'
    val oCount = hello wordCount 'o'
    println(lCount)   // 3
    println(oCount)   // 2
}

infix fun String.wordCount(c: Char) : Int{
    var count = 0
    for(n in this){
        if(n == c) count++
    }
    return count
}
```

Здесь функция wordCount проходит по всем символам строки и подсчитывает, сколько раз встречается символ, передаваемый через параметр функции. Результат возвращается функцией. Затем мы можем применить инфиксную нотацию:

```
val lCount = hello wordCount 'l'
```

Поскольку функция возвращает результат типа Int, то мы можем получить этот результат в переменную.