

Управляющие операторы

Несколько классных упрощений

Содержание

1. If else в Kotlin, тернарный оператор
2. Instanceof в Kotlin и cast
3. Null-safety
4. Sealed class вместо абстрактного класса в Джава

1. If else в Kotlin, тернарный оператор

Да, как ни странно, но в Kotlin те же самые операторы if else что и в Джава, но есть пару нюансов. Посмотрим.

```
fun max(a: Int, b: Int): Int {  
    var result: Int  
    if (a > b) {  
        result = a  
    } else {  
        result = b  
    }  
    return result  
}
```

Да, все конструкции с 1 условием или 2 и 3 можно писать цепочкой иф елсов как в джаве. Но давайте посмотрим как теперь нам идея поможет переделать код.

```
fun max(a: Int, b: Int): Int {  
    val result: Int  
    result = if (a > b) {  
        a  
    } else {  
        b  
    }  
    return result  
}
```

Оказывается, можно сразу писать val и вынести присваивание за условие. Давайте еще раз упростим код.

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

Да, как видите идея не предложила сделать тернарный оператор, просто потому что его в том же виде как в джава просто нет. Нужно писать иф елс, но можно в 1 линию.

Давайте посмотрим на более сложную конструкцию.

```
fun checkNumber(x: Int) {  
    if (x > 0) {  
        print("positive")  
    } else if (x < 0) {  
        print("Negative")  
    } else {  
        print("zero")  
    }  
}
```

Помните switch в Java ? Мы бы не смогли в джава преобразовать это в нечто иное. Потому что свитч работает с int и string или же с enum. А теперь посмотрите на то, что предлагает нам идея для котлин кода.

```
fun checkNumber(x: Int) {  
    when {  
        x > 0 -> print("positive")  
        x < 0 -> print("Negative")  
        else -> print("zero")  
    }  
}
```

When. Да, это не то же самое что и свитч в джава. Потому что здесь можно указывать аргумент, а можно и не указывать. т.е. если бы смотрели на конкретные значения, то оно было бы так

```
fun convertGrade(grade: Int) {  
    when (grade) {  
        5 -> print("A")  
        4 -> print("B")  
        3 -> print("C")  
        2 -> print("D")  
        1 -> print("E")  
        else -> throw IllegalArgumentException("unknown grade")  
    }  
}
```

Давайте предположим, что у нас двойка и единица одно и то же. Можно сделать так:

```
fun convertGrade(grade:Int) {
    when (grade) {
        5 -> print("A")
        4 -> print("B")
        3 -> print("C")
        2, 1 -> print("D")
        else -> throw IllegalArgumentException("unknown grade")
    }
}
```

И да, здесь вы можете написать сразу `= when` вместо фигурных скобок метода. Ладно, а что если значений много? Можно сделать такое

```
fun checkNumber(x: Int) {
    when (x) {
        in 1..100 -> print("positive")
        in -100..-1 -> print("Negative")
        else -> print("out of range")
    }
}
```

Вместо того чтобы писать `x > 1 && x < 100` можно использовать `range`.

2. Instanceof в котлин и каст

Мы не закончили говорить про `when`, но для продолжения мы обсудим синтаксис проверки классов и каст. Если помните в джава все классы наследовались от `Object`, в котлин можно использовать другое – `Any`. Давайте посмотрим на это.

```
fun check(x:Any) = when(x) {
    is String -> print(x.isEmpty())
    is Int -> print (x + 1)
    else -> print("Unknown type")
}
```

Самое интересное то, что после проверки на инстанс через `is` мы можем обращаться к методам класса без каста. Кстати, как обстоят дела с кастом типов в котлин? Все просто:

```
fun check(x: Any) {
    print((x as String).isEmpty())
}
```

В джава мы делали так `((String) x).isEmpty()`, но в котлине попроще. Но мы до сих пор не говорили про то, что котлин `nullsafe` язык. Ну давайте изучим этот момент и вернемся к операторам `is` , `as`.

3. Null-safety

В котлин продумали проблему с null и сделали так, что если вы пишете тип переменной String то это означает сразу же что он не может никак хранить null. Давайте посмотрим на это.

```
fun main(args: Array<String>) {  
    var x : String = "a"  
    x = null  
}
```

Хорошо, а что если нам приходит код из джава класса и там метод помечен как @Nullable, что будет в котлин коде? Специально чтобы подчеркнуть возможность хранения null в котлине придумали добавлять знак вопроса в конце. Смотрите.

```
fun main(args: Array<String>) {  
    var x : String? = "a"  
    x = null  
}
```

Было String, а стало String? как видите. Сразу видно что можно хранить null. Это очень удобно для проверок. Посмотрите теперь на следующее.

```
9  
10 fun main(args: Array<String>) {  
11     val x: String? = null  
12  
13     print(x.length)  
14  
15 }
```

Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?

В джава ты мог бы хранить null в том же String и спокойно обращаться к методам и полям объекта. Но получил бы ошибку если все же там null. Котлин же не терпит подобного и требует что-то сделать с этим. Как видите из сообщения – только безопасный вызов через точку можно использовать или же через 2 восклицательных знака. В чем их различие?

Для начала посмотрим что предлагает идея – жмем Alt+Enter. Первый вариант сделать как в джава

```
if (x != null) {  
    print(x.length)  
}
```

Простая проверка на то, что это не null и спокойно вызываем. А теперь посмотрим на второй вариант.

Мы использовали безопасный вызов – как видите написали ?. Но что же будет в консоли? Null. Но зато не завершение исключением. Кстати, называется NullPointerException – когда вы пытались использовать методы или поля объекта, который был null.

```
@JvmStatic
fun main(args: Array<String>) {
    val x: String? = null

    print(x?.length)
}
```

Main

h: Main x

/usr/lib/jvm/java-1.8.0-openjdk-amd64

null

Process finished with exit code 0

Чтобы понять это давайте выделим переменную изнутри вызова print - Ctrl+Alt+V

```
val length: Int? = x?.length
print(length)
```

Т.е. метод print может спокойно принять null как аргумент и вывести в консоли. т.е. магическое `x?.length` означает следующее : `val length = if (x == null) null else x.length`

Да, котлин помогает писать многие вещи короче. Что касается второго варианта – `print(x!!.length)` то это джава стиль – если будет `x` null то ты получаешь ошибку как и в джава. Очень не рекомендуется использовать `!!` если вы не уверены на 100 процентов что на этой линии переменная уже точно не null.

Вроде разобрались – знак вопроса указывает на возможность наличия null. Но вспомним зачем мы об этом начали говорить – проверка типов и каст. Если есть операторы `is` и `as` то должны быть `is?` и `as?` Что же они тогда делают? Посмотрите на следующий код.

```
fun check(x:Any?) = when (x) {
    is String -> print("it's string")
    is String? -> print("it's string, maybe null")
    else -> print("unknown")
}
```

Нет, если вы хотите проверить `instanceof` в котлин, то выбирайте класс или же класс со знаком вопроса. Как видите наша функция проверит что пришла строка или же строка или null. Кстати, null строковый и null числовой это все тот же null. Докажем это просто так.

```
@JvmStatic
fun main(args: Array<String>) {
    val x:Int? = null
    check(x)
}
```

Main > main()

h: Main x

/usr/lib/jvm/java-1.8.0-openjdk-amd64/b

it's string, maybe null

Process finished with exit code 0

Как видите у нас переменная которая может хранить или число или null, но нул он и в Африке нул, поэтому в консоли видим что это или null или строка. Мы бы могли написать иначе

```
fun check(x: Any?) = when {  
    x is String -> print("it's string")  
    x == null -> print("it's null")  
    else -> print("unknown")  
}
```

И теперь наш код выдаст нам все верно

```
@JvmStatic  
fun main(args: Array<String>) {  
    val x: Int? = null  
    check(x)  
}
```

Main x
/usr/lib/jvm/java-1.8.0-openjdk-amd64
it's null
Process finished with exit code 0

Ладно, разобрались с is . Что же с as? и as тогда? Посмотрим

```
@JvmStatic  
fun main(args: Array<String>) {  
    val x: Int? = null  
    (x as String).length  
}
```

Main
Main x
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Exception in thread "main" kotlin.TypeCastException: null cannot be cast to non-null type
at Main.main(Main.kt:10)
Process finished with exit code 1

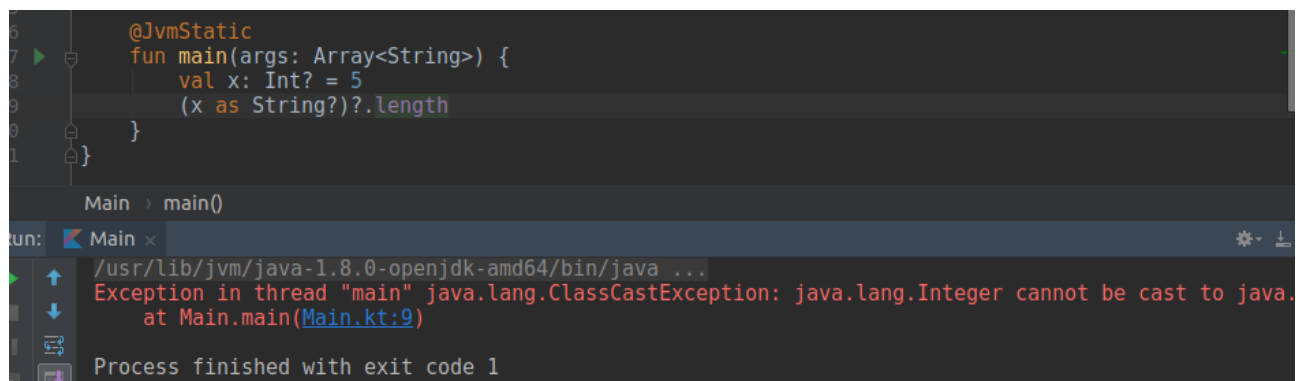
as нужно использовать если вы точно уверены что у вас пройдет каст, как в джава. А теперь сделаем то же самое, но безопасно.

```
(x as String?)?.length
```

Ну здесь понятно, x как строка или нул и так как итог будет нулем, то берем безопасный вызов длины. А теперь давайте перенесем один знак вопроса в оператор.

```
(x as? String)?.length
```

Может показаться что разницы нет, но до тех пор пока у вас x нул. Давайте поставим ему число.



```
@JvmStatic
fun main(args: Array<String>) {
    val x: Int? = 5
    (x as String)?.length
}
```

Main > main()

run: Main x

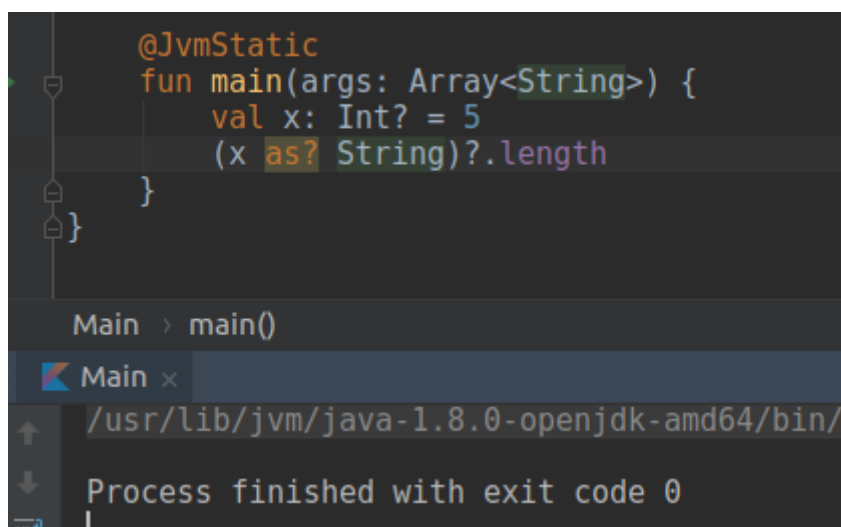
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...

Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

at Main.main(Main.kt:9)

Process finished with exit code 1

Мы говорим – я точно знаю что мой икс или нул или строка, если не нул, значит строка и тогда возьми у него длину. А теперь посмотрим второй вариант



```
@JvmStatic
fun main(args: Array<String>) {
    val x: Int? = 5
    (x as? String)?.length
}
```

Main > main()

Main x

/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/

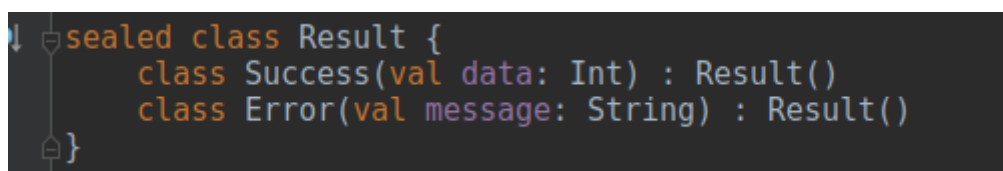
Process finished with exit code 0

Вот и кардинальное различие. Мы не знаем какого типа наш x, но если оно строка, то давайте возьмем длину у него. Иначе игнор. На джаве это будет так `if (x instanceof String){ if (x !=null.`

Надеюсь понятно. Теперь мы можем вернуться к нашей основной теме лекции.

4. Sealed class вместо абстрактного класса в Джава

В предыдущей лекции мы рассматривали какие бывают классы в котлин. Один из самых классных фиш котлина это sealed class. Что же это такое? А помните мы писали абстрактный класс, наследовались от него и потом вместо того чтобы проверять инстанс еще передавали в конструктор тип через enum? Ну вот, все это можно упростить в котлин.



```
sealed class Result {
    class Success(val data: Int) : Result()
    class Error(val message: String) : Result()
}
```

Итак, у нас есть абстрактный как бы класс Result и у него 2 наследника. Да, в котлин вместо

слова `extends` пишем : удобно, правда? И делаем разные данные каждому классу. Как видите все очень просто пишется. А использование еще проще. Сразу скажу это пример из реальной разработки. Мы в Андроид часто так делаем. Чуть немного сложнее, но это позже увидите.

```
interface Repository {  
    fun getData(): Result  
}
```

Пусть у нас будет интерфейс с методом, который вернет результат. Как же нам тогда понять какой результат? С помощью `when` и оператора `is`. Посмотрите

```
fun main(args: Array<String>) {  
    val repository = object : Repository {  
        override fun getData(): Result {  
            TODO()  
        }  
    }  
    when (val result = repository.getData()) {  
        is Result.Success -> print(result.data)  
        is Result.Error -> throw IllegalArgumentException(result.message)  
    }  
}
```

Если вы еще не написали код там где оно нужно можете написать `TODO()` и не нужно писать `return` что-то. Код не запустится конечно же, но вы можете позже добавить нужный код. И еще одно важное что умеет `when` передавать внутрь переменную и там же инициализировать. Помните как было с `try with resources` в джава? Так же и здесь. Похожий код мы будем видеть часто. `Sealed` классы очень удобные для использования. Мы можем сделать как в джава – добавить тип и требовать у каждого наследника. Но тогда нам нужно сделать поля одинаковыми или методы. А мы хотим чтобы у каждого наследника была своя структура данных и таким образом мы легко обращаемся к этим полям и методам. Потому что при проверке в `when` мы уже кастируем к нужному классу.

В следующей лекции мы поговорим про дженерики и как улучшить этот класс результата таким образом, чтобы его можно было использовать под все возможные структуры данных успеха и ошибки.

Про `data classes` придется рассказать в иной лекции, мы дошли до стандартного размера лекции – 8 страниц.

Для закрепления знаний найдите в старых задачах где можно применить новые знания и перепишите на котлин.