

Операторы `?.`, `?:` и `!!` языка Kotlin

Проблема обнуляемых типов (которые могут среди прочего принимать значение `null`) в том, что многие свойства и методы их узких классов-аналогов (без поддержки `null`) становятся недоступными.

```
fun main() {  
    val a: String = "Hello"  
    val b: String? =  
        if ((0 ≤ .. ≤ 1).random() == 1)  
            "World"  
        else  
            null  
  
    val aL = a.length  
    val bL = b.length  
  
    val aI = a[2]  
    val bI = b[2]  
  
    val aU = a.uppercase()  
    val bU = b.uppercase()  
}
```

В примере переменной `b` может быть присвоено одно из двух значений: либо строка `"World"`, либо значение `null`. Однако независимо от значения на переменную, которая потенциально может содержать `null`, вызывать атрибуты обычно строки нельзя.

В теории в таких ситуациях следует

1. проверить, что значение не является `null`,
2. привести его к другому типу (с помощью оператора `as`), в данном случае non-nullable (необнуляемому),

3. только после этого вызывать свойства и методы этого типа.

```
fun main() {  
    val b: String? =  
        if ((0 ≤ .. ≤ 1).random() == 1)  
            "World"  
        else  
            null  
    val c: String  
  
    if (b != null) {  
        c = b as String  
        println(c.length)  
    }  
}
```

Однако умный компилятор Kotlin во многих случаях позволяет не делать такое приведение вручную. Если сравнение в заголовке `if` возвращает истину (значение переменной не равно `null`), то в области действия сравнения компилятор будет считать это значение обычным. Как бы сам временно приведет переменную к необнуляемому типу.

```
fun main() {  
    val b: String? = readlnOrNull()  
  
    if (b != null)  
        println(b.length)  
}
```

```
fun main() {  
    val b: String? = readlnOrNull()  
  
    if (b != null)  
        println(b.length)  
}
```

Smart cast to kotlin.String

```
val b: String?  
KotlinBaseCode
```

Обратите внимание, что среда IntelliJ IDEA в теле `if` подсвечивает nullable-переменную зеленым цветом. При наведении указателя мыши вы увидите сообщение об умном приведении – smart cast.

Если следует обработать значение `null`, потребуется ветка `else`.

```
fun main() {  
    val b: String? =  
        if ((0..1).random() == 1)  
            "World"  
        else  
            null  
  
    if (b != null) {  
        println(b.length)  
    }  
    else {  
        println(-1)  
    }  
}
```

В Kotlin умное приведение также работает в ветке `else`, если в условии при `if` значение проверяется на равенство `null`.

```
fun main() {  
    val b: String? =  
        if ((0 ≤ .. ≤ 1).random() == 1)  
            "World"  
        else  
            null  
  
    if (b == null) {  
        println(-1)  
    }  
}
```

```
}  
else {  
    println(b.length)  
}  
}
```

Таким образом проблема nullable-типов в Kotlin решается просто. Однако несколько громоздко. Поэтому в языке предусмотрены специальные операторы и функции, упрощающие обработку значения `null`. В этом уроке рассмотрим три оператора Kotlin – оператор безопасного вызова, оператор "элвис" и утверждение "это не null".

? . – оператор безопасного вызова

Оператор безопасного вызова (safe-call operator), обозначаемый вопросительным знаком с точкой, похож на проверку на `null` в варианте с `if` без `else`. Он проверят, что значение слева от него не равно `null`. И только если это так, вызывается атрибут объекта, стоящий справа от оператора.

Если же значение равно `null`, то ничего не происходит. Точнее, все выражение возвращает `null`.

Можно представить, что слева от `? .` стоит проверка условия на неравенство `null`, а справа – это тело `if`, которое выполняется, если условие вернуло истину.

```
fun main() {  
    val b: String? = "World"  
    val c: String? = null  
  
    val bL: Int? = b?.length  
    val cL: Int? = c?.length  
  
    println(bL) // 5  
    println(cL) // null  
}
```

Поскольку выражение с оператором безопасного вызова метода или свойства потенциально может вернуть `null`, то значение всего выражения с этим

оператором всегда будет принадлежать какому-либо nullable-типу.

?: – оператор "элвис"

Оператор, обозначаемый вопросительным знаком с двоеточием (Elvis operator), подобен проверке на `null` в варианте if-else. Он возвращает значение слева от себя, если оно не `null`. И возвращает значение справа от себя, если то, что слева, – `null`.

Другими словами, после elvis-оператора находится значение по-умолчанию, которое возвращается только в том случае, если выражение до **?:** вернуло `null`.

Оператор **?:** используется для замены `null` каким-либо значением, принадлежащим обычно зауженному типу. В результате выражение с "элвисом" позволяет не увеличивать в программе количество nullable-переменных.

```
fun main() {  
    val b: String? = readLnOrNull()  
    val c: String = b ?: ""  
  
    println(c.length)  
}
```

Переменная `c` будет содержать либо строку, которую вернула функция `readLnOrNull()`, либо пустую строку, если `readLnOrNull()` вернет `null`. Избавимся от переменной `c`:

```
fun main() {  
    val b: String = readLnOrNull() ?: ""  
  
    println(b.length)  
}
```

Оператор **?:** часто используют совместно с оператором безопасного вызова. Если надо вызвать метод типа без поддержки `null` на nullable-переменную, требуется оператор безопасного вызова. Поскольку в случае `null` метод вызываться не будет, а `null` надо заменить на значение по-умолчанию, используется оператор "элвис".

```

fun main() {
    val b: String? =
        if ((0..1).random() == 1)
            "World"
        else
            null

    val c: Int = b?.length ?: -1

    println(c)
}

```

Последовательность действий в выражении `b?.length ?: -1` идет слева направо. Сначала выполняется подвыражение `b?.length`. Потом его результат подставляется как левый операнд оператора `?:`. Так если длина строки была удачно измерена, то она и запишется в переменную `c`. Если же оператор безопасного вызова вернул `null`, то в переменную запишется `-1`.

!! – утверждение "это не null"

Два восклицательных знака (not-null assertion operator), стоящих после nullable-переменной, преобразуют ее значение к типу без поддержки `null`. При этом перед преобразованием никак не проверяется, что значение действительно не содержит `null`.

Поэтому, если в процессе выполнения программы окажется, что значение, которое пытается преобразовать оператор `!!`, все-таки `null`, останется только один выход – выбросить исключение `NullPointerException`. Если оно не обрабатывается кодом, программа аварийно завершится.

```

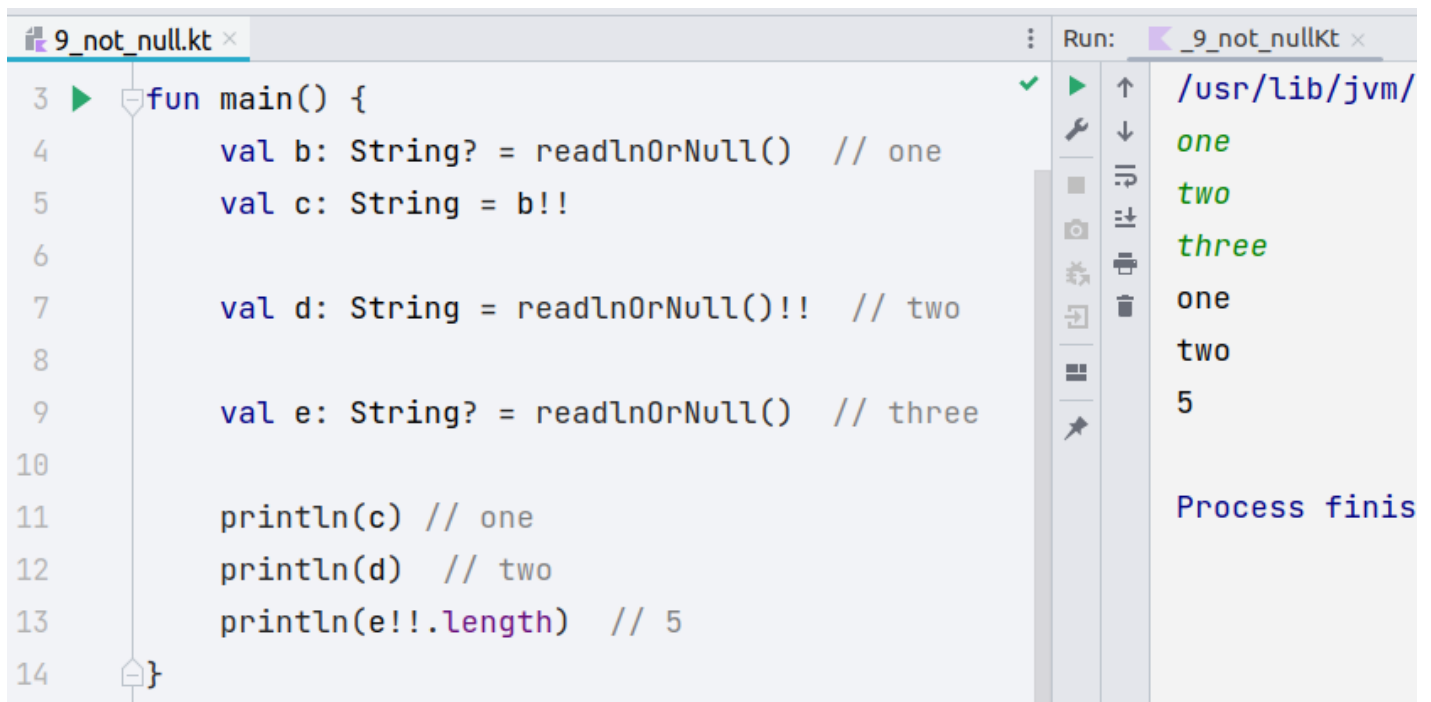
9_exc.kt
2
3 fun main() {
4     val b: String? =
5         if ((0 ≤ .. ≤ 1).random() == 1)
6             "World"
7         else
8             null
9
10    println(b!!.length)
11 }
12
13
Run: _9_excKt
/usr/lib/jvm/java-19-openjdk-amd64/bin/java -javaagent:/opt/idea-IC-223
Exception in thread "main" java.lang.NullPointerException Create breakpoint
    at _2_null_operators._9_excKt.main(9_exc.kt:10)
    at _2_null_operators._9_excKt.main(9_exc.kt)
Process finished with exit code 1

_9_excKt
/usr/lib/jvm/java-19-openjdk-amd64/bin/java -javaagent:/opt/idea-IC-223
5
Process finished with exit code 0

```

Поэтому, несмотря на удобство этого оператора, его следует использовать только там, где вы уверены, что `null` быть не может. В программе ниже показано использование утверждения `not-null` на разных этапах.

```
fun main() {  
    val b: String? = readlnOrNull()  
    val c: String = b!!  
  
    val d: String = readlnOrNull()!!  
  
    val e: String? = readlnOrNull()  
  
    println(c)  
    println(d)  
    println(e!!.length)  
}
```



```
9_not_null.kt x Run: _9_not_nullKt x  
3 fun main() {  
4     val b: String? = readlnOrNull() // one  
5     val c: String = b!!  
6  
7     val d: String = readlnOrNull()!! // two  
8  
9     val e: String? = readlnOrNull() // three  
10  
11     println(c) // one  
12     println(d) // two  
13     println(e!!.length) // 5  
14 }
```

/usr/lib/jvm/
one
two
three
one
two
5
Process finis

[PDF-версия курса с дополнительными уроками](#)