Лекция #5. Типы данных и операции в Kotlin, часть 2

Boolean и операторы сравнения в Kotlin

Мы уже познакомились с несколькими типами данных, такими как **Int**, **Double** и **String**. Рассмотрим тип данных, который используется с операторами сравнения.

При сравнении двух чисел для поиска наибольшего **1 > 2** ответом будет либо истина, либо ложь. В **Kotlin** для этого предусмотрен отдельный тип данных — **Boolean**. Тип был назван так в честь человека по имени Джордж Буль, который ввел целую математическую область, рассматривающую понятия истинного и ложного.

В **Kotlin** булев тип используется следующим образом:

```
1 val yes: Boolean = true
2 val no: Boolean = false
```

Булево значение может быть только истинным или ложным, для чего используются ключевые слова **true** и **false**. В коде выше, ключевые слова используются для указания состояния каждой константы.

Boolean, или логические операторы в Kotlin

Булев тип используется для сравнения значений. К примеру, когда нужно узнать, равны ли два значения, результат будет либо **true**, либо **false**.

В **Kotlin** для этого используется оператор равенства, который обозначается как **== :**

```
1 val doesOneEqualTwo = (1 == 2)
```

Kotlin знает, что типом константы **doesOneEqualTwo** является тип **Boolean**. Очевидно, что 1 не равен 2. Следовательно, константа **doesOneEqualTwo** будет **false**.

Аналогичным образом можно выяснить, не равны ли два значения через использование оператора **!=** :

```
1 | val doesOneNotEqualTwo = (1 != 2)
```

На этот раз сравнение истинно (**true**), потому что 1 не равна 2, и константа **doesOneNotEqualTwo** будет **true**.

Оператор префикс **!** переключает **true** на **false** и наоборот. Выше указанный код можно написать следующим образом:

```
1 | val alsoTrue = !(1 == 2)
```

Так как 1 не равна 2, (1 == 2) принимает значение **false**, и затем **!** переключает это на **true**.

Еще два оператора помогают узнать если одно значение больше (**>**) или меньше (**<**) другого значения.

```
1 | val isOneGreaterThanTwo = (1 > 2)
2 | val isOneLessThanTwo = (1 < 2)
```

Несложно понять, что константа **isOneGreaterThanTwo** будет **false**, а константа **isOneLessThanTwo** будет **true**.

Также существует оператор, который проверяет, меньше значение другого значения или равно ему: **<=**. Это комбинация **<** и **==**, поэтому будет возвращено значение true, если первое значение меньше второго или равно ему.

Аналогичный оператор, который позволяет проверить, больше ли значение или равно другому: **>=**.

Булева логика в Kotlin

В каждом из приведенных выше примеров проверяется только одно условие. Когда Джордж Буль ввел булевы значения, у него были более грандиозные планы. Он изобрел булеву логику, которая позволяет комбинировать несколько условий для формирования результата.

Один из способов комбинировать условия — использовать **AND** (**И**). Когда вы соединяете с **AND** (**И**) два булевых значения, результатом является другое булево значение. Если оба входных значения истинны, результат будет истинным. В противном случае результат будет ложным.

В **Kotlin** оператором для **AND** (**И**) является **&&:**

```
val and = true && true
```

В данном случае константа **and** будет равна **true**. Если бы какое-то из значений справа было **false**, тогда **and** тоже равнялась бы **false**.

Другим способом соединения условий является использование **OR** (**ИЛИ**). При использовании **OR** с двумя булевыми значениями результат будет **true**, если хотя бы одно из булевых значений истинно. Только если оба значения ложны, результат будет равен **false**.

В **Kotlin** для булева оператора **OR** используется **||**:

```
val or = true || false
```

В данном случае константа **or** будет равна **true**. Если бы оба значения справа были **false**, тогда константа or тоже была бы равна **false**. Если бы они оба были **true**, тогда значением константы **or** по-прежнему было бы **true**.

В **Kotlin** булева логика обычно используется, когда есть несколько условий. Возможно, требуется определить, если два условия истинны. В данном случае можно использовать **AND**. Если нужно выяснить, является ли истинными хотя бы одно из двух условий, тогда можно использовать **OR**.

К примеру, рассмотрим следующий код:

```
val andTrue = 1 < 2 && 4 > 3
val andFalse = 1 < 2 && 3 > 4

val orTrue = 1 < 2 || 3 > 4
val orFalse = 1 == 2 || 3 == 4
```

В каждом случае проверяются два разных условия, которые соединяются с помощью **AND** или **OR**.

Также можно использовать булеву логику для объединения более двух сравнений. К примеру, можно создать сложное сравнение вроде следующего:

```
val andOr = (1 < 2 && 3 > 4) || 1 < 4
```

Скобки устраняют неоднозначность выражения. Сначала **Kotlin** оценивает выражение внутри круглых скобок, а затем оценивает все выражение, выполнив следующие этапы:

```
1. (1 < 2 && 3 > 4) || 1 < 4
2. (true && false) || true
3. false || true
4. true
```

Равенство строк в Kotlin

Зачастую требуется определить, равны ли две строки. К примеру, для детской игры по угадыванию названий животных по фото потребуется определить, ответил ли игрок правильно.

В **Kotlin** можно сравнить две строки через использование оператора равенства **==**. Все делается точно так же, как и в случае сравнения чисел. К примеру:

```
val guess = "dog"
val dogEqualsCat = guess == "cat"
```

Здесь у константы **dogEqualsCat** булев тип, ее значения равно **false**, потому что "**dog**" не равна "**cat**".

Как и в случае с числами, здесь можно не только узнать, являются ли строки равными, но и сравнить их. К примеру:

```
val order = "cat" < "dog"
```

Инструкция проверяет, идут ли буквы по алфавиту в одной строке раньше, чем в другой. В данном случае константа **order** равна **true**.

Выражение if else в Kotlin

Наиболее распространенным способом управления порядком выполнения программы является использование выражения **if**, которое указывает программе на выполнение определенного действия при определенном условии.

Рассмотрим следующий пример:

```
if (2 > 1) {
  println("Да, 2 больше чем 1.")
}
```

Это простое выражение **if**. Если условие истинно, то выражение выполнит код между фигурными скобками. Если условие ложно, то выражение не будет выполнять данный код.

Термин **выражение if** используется здесь вместо **оператора if**, поскольку, в отличие от многих других языков программирования, в **Kotlin** значение возвращается из выражения **if**. Возвращаемое значение является значением последнего выражения в блоке **if**.

От вас не требуется использовать возвращаемое значение или присваивать его переменной. Подробнее о возврате значения будет рассказано дальше. Вы можете расширить выражение **if**, чтобы указать действия для случая, когда условие ложное. Это называется условием **else**. К примеру:

```
val animal = "Лиса"

if (animal == "Кошка" || animal == "Собака") {
    println("Это домашнее животное.")
} else {
    println("Это дикое животное.")
}
```

Здесь, если константа **animal** была бы равна "**Кошка**" или "**Собака**", тогда выражение выполняет первый блок кода. Если константа **animal** не равна ни "**Кошка**", ни "**Собака**", тогда выражение выполняет блок внутри **else** от **if** выражения, выводя на консоль следующее:

```
Это дикое животное.
```

Также можно использовать выражение **if-else** в одну строку. Рассмотрим, как это поможет сделать код более кратким и читабельным.

Если нужно определить минимальное и максимальное значение из двух переменных, можно использовать выражение **if** следующим образом:

```
val a = 5
val b = 10

val min: Int
if (a < b) {
    min = a
} else {
    min = b
}

val max: Int
if (a > b) {
    max = a
} else {
    max = b
}
```

На данный момент вам должно быть понятно, как это работает, но здесь используется довольно много кода. Взглянем, как можно улучшить код через использование выражения **if-else**, которое возвращает значение.

Просто уберем скобки и поместим все в одну строку следующим образом:

```
val a = 5
val b = 10

val min = if (a < b) a else b
val max = if (a > b) a else b
```

В первом примере мы имеем условие **a < b**. Если это истинно, то результат, присваиваемый назад в константу **min**, будет значение из константы **a**. Если ложно, то результат будет значением из константы **b**. Значение в константе **min** будет **5**. Во втором примере, **max** присваивается значение **b**, которым является **10**.

Так намного проще. Это пример идиоматического кода, который означает, что код пишется ожидаемым образом для определенного языка программирования. Идиомы не только улучшают код, но и позволяют другим разработчикам, знакомым с языком, быстро понять чужую программу.

На заметку: Так как поиск большего или меньшего из двух чисел является очень распространенной операцией, стандартная библиотека **Kotlin** предоставляет для этой цели две функции: **max()** и **min()**.

Выражения **if** можно использовать более эффективно. Иногда нужно проверить одно условие, затем другое. Здесь вступает в игру **else-if**, встраивая другое выражение **if** в условие **else** предыдущего условия **if**.

Вы можете использовать **else-if** следующим образом:

```
fun main() {
    val hourOfDay = 12

    val timeOfDay = if (hourOfDay < 6) {
        "Раннее утро"
    } else if (hourOfDay < 12) {
        "Утро"
    } else if (hourOfDay < 17) {
        "После полудня"
    } else if (hourOfDay < 20) {
        "Вечер"
    } else if (hourOfDay < 24) {
        "Поздний вечер"
    } else {
        "НЕДЕЙСТВИТЕЛЬНЫЙ ЧАС!"
    }
    println(timeOfDay)
}
```

Данные вложения **if** проверяют несколько условий одно за другим, пока не будет найдено истинное условие. Выполняется только тело-**if**, связанный с первым истинным условием, независимо от того, истинны ли последующие условия **else-if**. Другими словами, важен порядок ваших условий.

В конце можно добавить вложение **else** для обработки случая, когда ни одно из условий не выполняется. Если условие **else** не нужно, его можно не использовать. В этом примере оно требуется, чтобы гарантировать, что константа **timeOfDay** обладает допустимым значением ко времени его вывода.

В данном примере выражение **if** принимает число, являющееся временем, и преобразует его в строку, представляющую часть дня, которой принадлежит время. При работе в 24-часовом формате условия проверяются в следующем порядке, по одному пункту за раз:

Сначала проверяется, если время меньше 6. Если так, значит, это раннее утро;

Если время не меньше 6, тогда выражение продолжается до первого условия **else-if**, где проверяется, является ли время меньше 12;

Когда условие оказывается ложным, выражение проверяет, является ли время меньше 17, затем меньше ли 20, затем меньше ли 24;

В конечном итоге, если время за пределами диапазона, выражение возвращает недействительное значение.

В коде выше константа **hourOfDay** равна 12. Следовательно, код выведет следующее:

```
1 После полудня
```

Обратите внимание, что хотя оба условия **hourOfDay < 20** и **hourOfDay < 24** истинны, выражение выполняет и возвращает только первый блок, чье условие истинно. В данном случае это блок с условием **hourOfDay < 17**.

Замыкание в Kotlin

Важным аспектом выражений **if** и булевых операторов является то, что происходит, когда несколько логических условий разделяются с **AND (&&)** или **OR (||).**

Рассмотрим следующий код:

```
if (1 > 2 && name == "Matt Galloway") {
  // ...
}
```

Так как **1 < 2** истинно, все выражение также будет истинным. Следовательно, еще раз проверка **name** не выполняется. Это пригодится позже, когда вы начнете работать с более сложными типами данных.

Инкапсуляция переменных в Kotlin

Вместе с выражениями **if** вводится новый концепт — **область видимости**, которая является способом инкапсуляции переменных через использование фигурных скобок.

Представьте, что вам нужно посчитать оплату, которую вам должен заплатить клиент. Условия сделки заключаются в том, что вы зарабатываете 25 долларов каждый час до 40 часов работы. После этого времени вы получаете по 50 долларов в час.

Используя **Kotlin**, оплату можно посчитать следующим образом:

```
var hoursWorked = 45

var price = 0
if (hoursWorked > 40) {
    val hoursOver40 = hoursWorked - 40
    price += hoursOver40 * 50
    hoursWorked -= hoursOver40
}
price += hoursWorked * 25

println(price)
```

Данный код принимает количество часов и проверяет, не превышает ли оно 40. Если да, то код вычисляет количество часов, превышающее 40, и умножает его на 50 долларов, а затем добавляет результат к цене. Затем код вычитает количество часов, превышающих 40, из отработанных часов. Остальные отработанные часы умножаются на 25 долларов и прибавляются к общей стоимости.

Результат примера выглядит следующим образом: **1250**

Интересен код внутри выражения **if**. Здесь объявляется новая константа **hoursOver40** для хранения количества часов, превышающих **40**. Вы можете использовать ее внутри оператора **if**.

Но что произойдет при попытке использовать его в конце кода свыше?

```
...

println(price)
println(hoursOver40)
```

Будет получена ошибка: **Unresolved reference: 'hoursOver40'**

Ошибка сообщает, что вам нельзя использовать константу **hoursOver40** вне **области видимости**, в которой она была создана. Обратите внимание, данная константа была создана в теле **if** выражения.

В данном случае выражение **if** вводит новую область видимости, за пределами которой константа становится невидимой.

Каждая область видимости может использовать переменные и константы из родительской области видимости **main()**. В примере выше в области видимости внутри выражения **if** используются переменные **price** и **hoursWorked**, которые были созданы в родительской области видимости.

Цикл while в Kotlin

Циклы являются способом многократного выполнения кода в **Kotlin**.

Цикл **while** повторяет блок кода, пока выполняется условие. Цикл **while** в **Kotlin** создается следующим образом:

```
while (<CONDITION>) {
  <LOOP CODE>
}
```

Цикл проверяет условие для каждой итерации. Если условие истинно (**true**), цикл выполняется и переходит к другой итерации.

Если условие ложно (**false**), цикл останавливается. Как и выражения **if**, циклы **while** создает свою собственную область видимости.

Простейший цикл **while** имеет следующую форму:

```
while (true) {

}
```

Это вечный цикл **while** в **Kotlin**, так как условие всегда истинно (true). Бесконечный цикл не приведет к сбою программы, но из-за него может зависнуть компьютер.

Далее представлен более полезный пример цикла **while** в **Kotlin**:

```
var sum = 1

while (sum < 1000) {
    sum = sum + (sum + 1)
}
```

Данный код подсчитывает математическую последовательность до того момента, пока значение меньше 1000.

Цикл выполняется следующим образом:

- Перед итерацией 1: sum = 1, условие цикла = true
- После итерации 1: sum = 3, условие цикла = true
- После итерации 2: sum = 7, условие цикла = true
- После итерации 3: sum = 15, условие цикла = true
- После итерации 4: sum = 31, условие цикла = true
- После итерации 5: sum = 63, условие цикла = true
- После итерации 6: sum = 127, условие цикла = true
- После итерации 7: sum = 255, условие цикла = true
- После итерации 8: sum = 511, условие цикла = true
- После итерации 9: sum = 1023, условие цикла = false

После девятой итерации переменная **sum** равна 1023, следовательно, условие **sum < 1000** становится ложным (**false**). В данной точке — цикл останавливается.

Цикл do-while в Kotlin

Одним из вариантов цикла **while** является цикл **do-while**. Он отличается от цикла **while** тем, что условие проверяется в конце цикла, а не в начале. Это означает, что хотя бы 1 раз тело цикла будет выполнено.

```
do {
    <LOOP CODE>
} while (<CONDITION>)
```

Далее дан пример из прошлого раздела, только здесь используется цикл **do-while**:

```
sum = 1

do {
    sum = sum + (sum + 1)
} while (sum < 1000)
```

В данном примере вывод такой же, как и в предыдущем. Однако, так бывает не всегда. Рассмотрим следующий цикл **while**:

```
sum = 1

while (sum < 1) {
    sum = sum + (sum + 1)
}
```

Рассмотрим аналогичный цикл **do-while**, который использует такое же условие:

```
sum = 1

do {
  sum = sum + (sum + 1)
} while (sum < 1)
```

В случае обычного цикла **while** условие **sum < 1** является ложным с самого начала. Это означает, что тело цикла не будет выполнено. Значение **sum** будет равно 1, потому что цикл не будет выполнять никаких итераций. В случае цикла **do-while** сумма **sum** будет равна 3, потому что цикл **do-while** **ВСЕГДА** выполнится хотя бы один раз, даже если изначально условие ложное.

Прерывание цикла с помощью break в Kotlin

Иногда требуется прервать цикл раньше времени. Для этого можно использовать оператор **break**, который сразу прерывает цикл и продолжает выполнение кода после него.

К примеру, рассмотрим следующий код:

```
sum = 1

while (true) {
    sum = sum + (sum + 1)
    if (sum >= 1000) {
        break
    }
}
```

Здесь условие цикла истинно (**true**), поэтому цикл будет повторятся бесконечно. Однако оператор **break** указывает на то, что цикл **while** завершится, когда сумма будет больше или равна **1000**.

Мы узнали, как написать один и тот же цикл по-разному, и тем самым продемонстрировали, что в программировании бывает много способов добиться одного и того же результата разными способами.

Вам следует выбирать наиболее удобные в зависимости от ситуации. Это подход, который вы усвоите со временем и практикой.

Интервалы в Kotlin

Перед разбором цикла **for** нам обязательно нужно изучить интервалы, которые представляют собой последовательность целых чисел. Взглянем на два вида интервалов.

Первым типом является закрытый интервал, который записывается следующим образом:

```
fun main() {
    val closedRange = 0..5
}
```

Две точки (**..**) указывают на то, что данный интервал от 0 до 5 (включительно) конечен, он имеет логическое начало и конец. Это числа (0, 1, 2, 3, 4, 5).

Вторым типом является полуоткрытый интервал, который записывается следующим образом:

```
fun main() {
    val halfOpenRange = 0 until 5
}
```

Здесь вместо двух точек используется ключевое слово **until**. Полуоткрытый интервал не включает последнее число, в данном случае 5. Это числа (0, 1, 2, 3, 4).

Открытые и полуоткрытые интервалы, созданные с помощью операторов **..** и **until**, всегда идут по возрастанию. Другими словами, второе число всегда больше или равно первому. Для создания убывающего интервала используется команда **downTo**, получается закрытый интервал:

```
fun main() {
    val decreasingRange = 5 downTo 0
}
```

Числа из интервала — (5, 4, 3, 2, 1, 0).

Интервалы используются для циклов **for** и для **when**-выражений.

Синтаксис цикла for в Kotlin

Цикл **for** создается следующим образом:

```
for (<CONSTANT> in <RANGE>) {
  <LOOP CODE>
}
```

Цикл начинается с ключевого слова **for**, за которым следует название константы цикла (скоро мы поговорим и о ней), по середине у нас слово **in** и сам интервал цикла. Далее представлен пример:

```
fun main() {
    val count = 10

    var sum = 0
    for (i in 1..count) {
        sum += i
    }

    print(sum)
}
```

В приведенном выше коде, цикл **for** проходит через интервал от **1** до **count**. На первой итерации **i** будет равняться первому элементу интервала: 1. Каждый раз, проходя по циклу циклу, переменная **i** будет увеличиваться до тех пор, пока не сравняется с **count**. Цикл выполнится в последний раз, а затем завершится.

На заметку: Если вы использовали полуоткрытый интервал **until**, последней итерацией переменной **i** будет **count - 1**.

Внутри цикла, значение из **i** добавляется к переменной **sum**. Он запускается 10 раз для вычисления последовательности 1 + 2 + 3 + 4 + 5 + ... до 10.

Далее представлены значения **i** и переменной **sum** для каждой итерации:

- Начало итерации 1: i = 1, sum = 0
- Начало итерации 2: i = 2, sum = 1
- Начало итерации 3: i = 3, sum = 3
- Начало итерации 4: i = 4, sum = 6
- Начало итерации 5: i = 5, sum = 10
- Начало итерации 6: i = 6, sum = 15
- Начало итерации 7: i = 7, sum = 21
- Начало итерации 8: i = 8, sum = 28
- Начало итерации 9: i = 9, sum = 36
- Начало итерации 10: i = 9, sum = 45
- После итерации 10: sum = 55

Говоря об области видимости, константа **i** видна только внутри области видимости цикла **for**, а это значит, что она недоступна за пределами цикла.

Иногда требуется пройтись по циклу только определенное количество раз, и константа цикла не нужна вообще. В данном случае **цикл repeat** используется следующим образом:

```
fun main() {
    var sum = 1
    var lastSum = 0

    repeat(10) {
        val temp = sum
        sum += lastSum
        lastSum = temp
    }

    print(lastSum)
}
```

В интервале можно указать шаг. К примеру, подсчитать сумму нечетных чисел:

```
fun main() {
    var sum = 0
    val count = 10

    for (i in 1..count step 2) {
        sum += i
    }

    print(sum)
}
```

Здесь внутри цикла **for** есть оператор **step**. Цикл будет выполняться только для значений, на которые выпадает данный шаг. В этом случае вместо того, чтобы проходить через каждое значение в интервале, цикл будет проходить через все остальные значения. Таким образом, **i** всегда будет нечетным, потому что начальное значение равно 1.

В цикле **for** также можно использовать обратный отсчет, используя **downTo**. В таком случае, если **count** имеет значение 10, цикл будет перебирать значения (10, 8, 6, 4, 2).

```
fun main() {
    var sum = 0
    val count = 10

    for (i in count downTo 1 step 2) {
        sum += i
    }

    print(sum)
}
```

Маркированные операторы continue и break

Иногда требуется пропустить итерацию цикла для отдельного случая, не прерывая его полностью. Это можно сделать с помощью оператора **continue**, который сразу завершает текущую итерацию цикла и запускает следующую. Оператор **continue** дает более высокий уровень контроля, позволяя решать, где и когда пропустить итерацию.

Возьмем пример сетки 8 на 8, где каждая ячейка содержит значение строки, умноженное на столбец. Это очень похоже на таблицу умножения, не так ли?

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

Предположим, нужно посчитать сумму всех клеток, кроме нечетных рядов, как показано ниже:

	0	1	2	3	4	5	6	7
0								
1	0	1	2	3	4	5	6	7
2								
3	0	3	6	9	12	15	18	21
4								
5	0	5	10	15	20	25	30	35
6								
7	0	7	14	21	28	35	42	49

Через цикл **for** это делается следующим образом:

```
fun main() {
    var sum = 0
    for (row in 0 until 8) {
        if (row % 2 == 0) {
            continue
        }

        for (column in 0 until 8) {
            sum += row * column
        }
    }

    print("Результат: $sum")
}
```

Строка четная если результат деления номера строки на 2 равен 0. В этом случае **continue** заставляет цикл **for** переходить к следующей строке.

Оператор **break**, который мы уже использовали при изучении цикла **while**, можно использовать с циклами **for** и его использование прерывает работу цикла полностью. Как и **break**, **continue** работает как с циклами **for**, так и с циклами **while**.

Во втором примере кода, будет вычислена сумма всех ячеек, за исключением тех, в которых столбец больше или равен строке.

Должны складываться следующие ячейки:

	0	1	2	3	4	5	6	7
0								
1	0							
2	0	2						
3	0	3	6					
4	0	4	8	12				
5	0	5	10	15	20			
6	0	6	12	18	24	30		
7	0	7	14	21	28	35	42	

С помощью цикла **for** это можно сделать следующим образом:

```
fun main() {
    var sum = 0
    rowLoop@ for (row in 0 until 8) {
        columnLoop@ for (column in 0 until 8) {
            if (row == column) {
                continue@rowLoop
            }
            sum += row * column
        }
    }

    print("Результат: $sum")
}
```

Предыдущий блок кода использует **маркер**, помечая два цикла как **rowLoop** и **columnLoop** соответственно. Когда строка равна столбцу внутри цикла **columnLoop**, внешний цикл **rowLoop** будет продолжен.

Можно использовать такие маркированные операторы с **break**, чтобы выйти из определенного цикла. Обычно **break** и **continue** применяются в цикле внутри цикла. Поэтому маркированные операторы нужно использовать, когда требуется манипулировать главным циклом в теле которого мы находимся.