Диапазоны

Диапазон представляет набор значений или некоторый интервал. Для создания диапазона применяется оператор ..:

```
fun main() {
    val range = 1 ≤ .. ≤ 5 // диапазон [1, 2, 3, 4, 5]
}
```

Этот оператор принимает два значения - границы диапазона, и все элементы между этими значениями (включая их самих) составляют диапазон.

Диапазон необязательно должна представлять числовые данные. Например, это могут быть строки:

```
fun main() {
    val range = "a".."d"
}
```

Оператор .. позволяет создать диапазон по нарастающей, где каждый следующий элемент будет больше предыдущего. С помощью специальной функции **downTo** можно построить диапазон в обратном порядке:

```
fun main() {
    val range1 = 1 ≤ .. ≤ 5      // 1 2 3 4 5
    val range2 = 5 ≥ downTo ≥ 1    // 5 4 3 2 1
}
```

Еще одна специальная функция **step** позволяет задать шаг, на который будут изменяться последующие элементы:

```
fun main() {
    val range1 = 1 ≤ .. ≤ 10 step 2          // 1 3 5 7 9
    val range2 = 10 ≥ downTo ≥ 1 step 3      // 10 7 4 1
}
```

Еще одна функция **until** позволяет не включать верхнюю границу в диапазон:

```
fun main() {
    val range1 = 1 ≤ until < 9          // 1 2 3 4 5 6 7 8
    val range2 = 1 ≤ until < 9 step 2   // 1 3 5 7
}
```

С помощью специальных операторов можно проверить наличие или отсутствие элементов в диапазоне:

- **in**: возвращает true, если объект имеется в диапазоне
- **!in**: возвращает true, если объект отсутствует в диапазоне

```kotlin
fun main() {

    val range = 1 ≤ .. ≤ 5

    var isInRange = 5 in range
    println(isInRange)      // true

    isInRange = 86 in range
    println(isInRange)      // false

    var isNotInRange = 6 !in range
    println(isNotInRange)   // true

    isNotInRange = 3 !in range
    println(isNotInRange)   // false
}
```

Перебор диапазона

С помощью цикла for можно перебирать диапазон:

```kotlin
fun main() {
    val range1 = 5 ≥ downTo ≥ 1
    for (c in range1) {
        print(c)
    }   // 54321
    println()
    val range2 = 'a' ≤ .. ≤ 'd'
    for (c in range2) {
        print(c)
    }   // abcd
    println()
    for (c in 1 ≤ .. ≤ 9) {
        print(c)
    }     // 123456789
    println()
    for (c in 1 ≤ until < 9) {
        print(c)
    }    // 12345678
    println()
    for (c in 1 ≤ .. ≤ 9 step 2) {
        print(c)
    }  // 13579
}
```

Массивы

Для хранения набора значений в Kotlin, как и в других языках программирования, можно использовать **массивы**. При этом массив может хранить данные только одного того же типа. В Kotlin массивы представлены типом **Array**.

При определении массива после типа Array в угловых скобках необходимо указать, объекты какого типа могут храниться в массиве. Например, определим массив целых чисел:

```
fun main() {
    val numbers: Array<Int>
}
```

С помощью встроенной функции **arrayOf()** можно передать набор значений, которые будут составлять массив:

```
fun main() {
    val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
}
```

То есть в данном случае в массиве 5 чисел от 1 до 5.

С помощью индексов мы можем обратиться к определенному элементу в массиве. Индексация начинается с нуля, то есть первый элемент будет иметь индекс 0. Индекс указывается в квадратных скобках:

Main.kt

```
fun main() {
    val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
    val n = numbers[1]  // получаем второй элемент  n=2
    numbers[2] = 7      // переустанавливаем третий элемент
    println("numbers[2] = ${numbers[2]}") // numbers[2] = 7
}
```

Также инициализировать массив значениями можно следующим способом:

Main.kt

```
fun main() {
    val numbers = Array( size: 3, {5}) // [5, 5, 5]
}
```

Здесь применяется конструктор класса Array. В этот конструктор передаются два параметра. Первый параметр указывает, сколько элементов будет в массиве. В данном случае 3 элемента. Второй параметр представляет выражение, которое генерирует элементы массива. Оно заключается в фигурные скобки. В данном случае в фигурных скобках стоит число 5, то есть все элементы массива будут представлять число 5. Таким образом, массив будет состоять из трех пятерок.

Но выражение, которое создает элементы массива, может быть и более сложным. Например:

```
Main.kt
fun main() {
    var i = 1;
    val numbers = Array( size: 3) { i++ * 2 } // [2, 4, 6]
}
```

В данном случае элемент массива является результатом умножения переменной i на 2. При этом при каждом обращении к переменой i ее значение увеличивается на единицу.

Перебор массивов

Для перебора массивов можно применять цикл for:

```
Main.kt
fun main() {
    val numbers = arrayOf(1, 2, 3, 4, 5)
    for (number in numbers) {
        print("$number \t")
    }
}
```

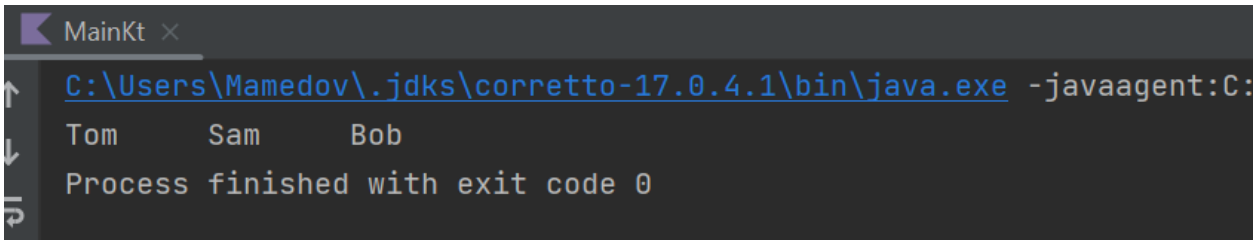
В данном случае переменная numbers представляет массив чисел. При переборе этого массива в цикле каждый его элемент оказывается в переменной number, значение которой, к примеру, можно вывести на консоль. Консольный вывод программы:

```
MainKt
C:\Users\Mamedov\.jdks\corretto-17.0.4.1\bin\j
1   2   3   4   5
Process finished with exit code 0
```

Подобным образом можно перебирать массивы и других типов:

```
Main.kt
fun main() {
    val people = arrayOf("Tom", "Sam", "Bob")
    for(person in people){
        print("$person \t")
    }
}
```

Консольный вывод программы:



Можно применять и другие типы циклов для перебора массива. Например, используем цикл while:

Здесь определена дополнительная переменная i, которая представляет индекс элемента массива. У массива есть специальное свойство indices, которое содержит набор всех индексов. А выражение i in people.indices возвращает true, если значение переменной i входит в набор индексов массива.

В самом цикле по индексу обащаемся к элементу массива: println(people[i]). И затем переходим к следующему индексу, увеличивая счетчик: i++.

То же самое мы могли написать с помощью цикла for:

Проверка наличия элемента в массиве

Как и в случае с последовательностью мы можем проверить наличие или отсутствие элементов в массиве с помощью операторов in и !in:

```kotlin
fun main() {
    val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
    println(4 in numbers)       // true
    println(2 !in numbers)      // false
}
```

Массивы для базовых типов

Для упрощения создания массива в Kotlin определены дополнительные типы BooleanArray, ByteArray, ShortArray, IntArray, LongArray, CharArray, FloatArray и DoubleArray, которые позволяют создавать массивы для определенных типов. Например, тип IntArray позволяет определить массив объектов Int, а DoubleArray - массив объектов Double:

```kotlin
    val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
    val doubles: DoubleArray = doubleArrayOf(2.4, 4.5, 1.2)
```

Для определения данных для этих массивов можно применять функции, которые начинаются на название типа в нижнем регистре, например, int, и затем идет ArrayOf.

Аналогично для инициализации подобных массивов также можно применять конструктор соответствующего класса:

```kotlin
    val numbers = IntArray( size: 3) { 5 }
    val doubles = DoubleArray( size: 3) { 1.5 }
```

Двухмерные массивы

Выше рассматривались одномерные массивы, которые можно представить в виде ряда или строки значений. Но кроме того, мы можем использовать многомерные массивы. К примеру, возьмем двухмерный массив - то есть такой массив, каждый элемент которого в свою очередь сам является массивом. Двухмерный массив еще можно представить в виде таблицы, где каждая строка — это отдельный массив, а ячейки строки — это элементы вложенного массива.

Определение двухмерных массивов менее интуитивно понятно и может вызывать сложности. Например, двухмерный массив чисел:

```kotlin
fun main() {
    val table: Array<Array<Int>> = Array( size: 3) { Array( size: 5) { 0 } }
}
```

В данном случае двухмерный массив будет иметь три элемента - три строки. Каждая строка будет иметь по пять элементов, каждый из которых равен 0.

Используя индексы, можно обращаться к подмассивам в подобном массиве, в том числе переустанавливать их значения

```kotlin
fun main() {
    val table = Array( size: 3) { Array( size: 3) { 0 } }
    table[0] = arrayOf(1, 2, 3)     // первая строка таблицы
    table[1] = arrayOf(4, 5, 6)     // вторая строка таблицы
    table[2] = arrayOf(7, 8, 9)     // третья строка таблицы
}
```

Для обращения к элементам подмассивов двухмерного массива необходимы два индекса. По первому индексу идет получение строки, а по второму индексу - столбца в рамках этой строки:

```kotlin
fun main() {
    val table = Array( size: 3) { Array( size: 3) { 0 } }
    table[0][1] = 6  // второй элемент первой строки
    val n = table[0][1]     // n = 6
}
```

Используя два цикла, можно перебирать двухмерные массивы:

```kotlin
fun main() {
    val table: Array<Array<Int>> = Array( size: 3) { Array( size: 3) { 0 } }
    table[0] = arrayOf(1, 2, 3)
    table[1] = arrayOf(4, 5, 6)
    table[2] = arrayOf(7, 8, 9)
    for (row in table) {
        for (cell in row) {
            print("$cell \t")
        }
        println()
    }
}
```

С помощью внешнего цикла for(row in table) пробегаемся по всем элементам двухмерного массива, то есть по строкам таблицы. Каждый из элементов двухмерного массива сам представляет массив, поэтому мы можем пробежаться по этому массиву и получить из него непосредственно те значения, которые в нем хранятся. В итоге на консоль будет выведено следующее:

```
C:\Users\Mamedov\.jdks\corretto-17.0.4.1\
1   2   3
4   5   6
7   8   9

Process finished with exit code 0
```