4. Списки

Введение

Начиная с этого раздела мы переходим к изучению составных типов данных, включающих в себя несколько элементов простых типов. Такие типы очень часто необходимы в программировании. Вспомним, например, задачу про поиск минимального корня биквадратного уравнения $ax^4 + bx^2 + c = 0$ из урока 2. В гораздо более распространённой формулировке она выглядела бы так: найти ВСЕ корни биквадратного уравнения.

Можно ли написать функцию, которая эту задачу решит? Конечно, да, но результатом подобной функции должен быть *список* найденных корней биквадратного уравнения. *Список* — это и есть один из очень распространённых составных типов со следующими свойствами:

- список может включать в себя произвольное количество *элементов* (от нуля до бесконечности);
- количество элементов в списке называется его размером;
- все элементы списка имеют один и тот же тип (в свою очередь, этот тип может быть простым — список вещественных чисел, или составным — список строк, или список списков целых чисел, или любые другие варианты);
- в остальном элементы списка независимы друг от друга.

Рассмотрим решение задачи о поиске корней биквадратного уравнения на Котлине:

```
fun biRoots(a: Double, b: Double, c: Double): List<Double> {
    if (a == 0.0) {
        if (b == 0.0) return listOf()
        val bc = -c / b
        if (bc < 0.0) return listOf()
        val root = sqrt(bc)
        return if (root == 0.0) listOf(root) else listOf(-root, root)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    val y1 = (-b + sqrt(d)) / (2 * a)
    val y2 = (-b - sqrt(d)) / (2 * a)
    // part1: List<Double>
    val part1 = if (y1 < 0) listOf() else if (y1 == 0.0) listOf(0.0) else {
        val x1 = sqrt(y1)
        listOf(-x1, x1)
    }
    // part2: List<Double>
    val part2 = if (y2 < 0) listOf() else if (y2 == 0.0) listOf(0.0) else {
        val x2 = sqrt(y2)
        listOf(-x2, x2)
```

```
    }
    return part1 + part2
}
```

Данное решение построено по алгоритму, приведённому в конце второго урока, с той лишь разницей, что здесь мы ищем все имеющиеся корни:

1. Первый **if** рассматривает тривиальный случай a = 0 и более простое уравнение $bx^2 = -c$. Оно либо не имеет корней (c / b > 0), либо имеет один корень 0 (c / b = 0), либо два корня (c / b < 0).
2. Затем мы делаем замену $y = x^2$ и считаем дискриминант $d = b^2 - 4ac$. Если он отрицателен, уравнение не имеет корней.
3. Если дискриминант равен 0, уравнение $ay^2 + by + c = 0$ имеет один корень. В зависимости от его знака, биквадратное уравнение либо не имеет корней, либо имеет один корень 0, либо имеет два корня.
4. В противном случае дискриминант положителен и уравнение $ay^2 + by + c = 0$ имеет два корня. Каждый из них, в зависимости от его знака, превращается в ноль, один или два корней биквадратного уравнения.

Посмотрите на тип результата функции `biRoots` — он указан как `List<Double>`. `List` в Котлине — это и есть список. В угловых скобках `<>` указывается так называемый *типовой аргумент* — тип элементов списка, то есть `List<Double>` вместе — это список вещественных чисел.

Для создания списков, удобно использовать функцию `listOf()`. Аргументы этой функции — это элементы создаваемого списка, их может быть произвольное количество (в том числе 0). В ряде случаев, когда биквадратное уравнение не имеет корней, функция `biRoots` возвращает пустой список результатов.

В последнем, самом сложном случае, когда уравнение $ay^2 + by + c = 0$ имеет два корня y_1 и y_2, мы формируем решения уравнений $x^2 = y_1$ и $x^2 = y_2$ в виде списков part1 и part2. Обе эти промежуточные переменные имеют тип `List<Double>` — в этом можно убедиться в IDE, поставив на них курсор ввода и нажав комбинацию клавиш `Ctrl+Q`. В последнем операторе **return** мы **складываем** два этих списка друг с другом: `return part1 + part2`, образуя таким образом третий список, содержащий в себе все элементы двух предыдущих.

Функцию `biRoots` можно несколько упростить, обратив внимание на то, что мы в ней **четыре** раза решаем одну и ту же задачу: поиск корней уравнения $x^2 = y$. Для программиста такая ситуация должна сразу превращаться в сигнал — **следует** написать для решения этой задачи отдельную, более простую функцию:

```
fun sqRoots(y: Double) =
        if (y < 0) listOf()
        else if (y == 0.0) listOf(0.0)
        else {
            val root = sqrt(y)
            // Результат!
```

```
            listOf(-root, root)
        }
```

Посмотрите внимательнее на оператор **if..else if..else**. Первые две его ветки формируют результат сразу же, используя `listOf()` и `listOf(0.0)`. А вот ветка **else** вначале создаёт промежуточную переменную `root` и уже потом формирует результат `listOf(-root, root)`. Запомните: результат ветки в таких случаях формирует **последний** её оператор.

Эту же функцию можно переписать с использованием оператора **when**:

```
fun sqRoots(y: Double) =
        when {
            y < 0 -> listOf()
            y == 0.0 -> listOf(0.0)
            else -> {
                val root = sqrt(y)
                // Результат!
                listOf(-root, root)
            }
        }
```

С использованием `sqRoots` функция `biRoots` примет следующий вид:

```
fun biRoots(a: Double, b: Double, c: Double): List<Double> {
    if (a == 0.0) {
        return if (b == 0.0) listOf()
        else sqRoots(-c / b)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    if (d == 0.0) return sqRoots(-b / (2 * a))
    val y1 = (-b + sqrt(d)) / (2 * a)
    val y2 = (-b - sqrt(d)) / (2 * a)
    return sqRoots(y1) + sqRoots(y2)
}
```

Из исходных 24 строчек осталось только 11, да и понимание текста функции стало существенно проще.

Напишем теперь тестовую функцию для проверки работы функции `biRoots`. Для этой цели последовательно решим с её помощью следующие уравнения:

- $0x^4 + 0x^2 + 1 = 0$ (корней нет)
- $0x^4 + 1x^2 + 2 = 0$ (корней нет)
- $0x^4 + 1x^2 - 4 = 0$ (корни -2, 2)
- $1x^4 - 2x^2 + 4 = 0$ (корней нет)

- $1x^4 - 2x^2 + 1 = 0$ (корни -1, 1)
- $1x^4 + 3x^2 + 2 = 0$ (корней нет)
- $1x^4 - 5x^2 + 4 = 0$ (корни -2, -1, 1, 2)

```
fun biRoots() {
    assertEquals(listOf<Double>(), biRoots(0.0, 0.0, 1.0))
    assertEquals(listOf<Double>(), biRoots(0.0, 1.0, 2.0))
    assertEquals(listOf(-2.0, 2.0), biRoots(0.0, 1.0, -4.0))
    assertEquals(listOf<Double>(), biRoots(1.0, -2.0, 4.0))
    assertEquals(listOf(-1.0, 1.0), biRoots(1.0, -2.0, 1.0))
    assertEquals(listOf<Double>(), biRoots(1.0, 3.0, 2.0))
    assertEquals(listOf(-2.0, -1.0, 1.0, 2.0), biRoots(1.0, -5.0, 4.0))
}
```

Обратите внимание, что здесь мы используем запись `listOf<Double>()` для создания пустого списка. Дело в том, что для вызовов вроде `listOf(-2.0, 2.0)` тип элементов создаваемого списка понятен из аргументов функции — это `List<Double>`. А вот вызов `listOf()` без аргументов не даёт никакой информации о типе элементов списка, в то же время, например, пустой список строк и пустой список целых чисел — с точки зрения Котлина не одно и то же.

Во многих случаях Котлин, тем не менее, может понять, о каком списке идёт речь. Например, функция `biRoots` имеет результат `List<Double>`, а значит, все списки, используемые в операторах **return**, должны иметь такой же тип. Случай с вызовом `assertEquals`, однако, не несёт достаточной информации, чтобы понять тип элементов, и мы вынуждены записать вызов функции более подробно — `listOf<Double>()`, указывая *типовой аргумент* `<Double>` **между** именем вызываемой функции и списком её аргументов в круглых скобках.

Запустим теперь написанную тестовую функцию. Мы получим проваленный тест из-за последней проверки:

```
org.opentest4j.AssertionFailedError: expected: <[-2.0, -1.0, 1.0, 2.0]> but was: <[-
2.0, 2.0, -1.0, 1.0]>
```

То есть мы ожидали список корней -2, -1, 1, 2, а получили вместо этого -2, 2, -1, 1. Дело в том, что списки в Котлине считаются равными, если совпадают их размеры, и соответствующие элементы списков равны. Списки, состоящие из одних и тех же элементов, но на разных местах, считаются разными.

В этом месте программист должен задуматься, а что, собственно, он хочет в точности от функции `biRoots`. Должны ли найденные корни быть упорядочены по возрастанию, или они могут присутствовать в списке в любом порядке? Если должны, то он должен исправить функцию `biRoots`, а если нет — то тестовую функцию, так как она требует от тестируемой функции больше, чем та по факту даёт.

В обоих случаях нам придётся отсортировать список найденных корней перед сравнением.

В Котлине это можно сделать, вызвав функцию `.sorted()`:

```
fun biRoots() {
    // ...                                                              v
    assertEquals(listOf(-2.0, -1.0, 1.0, 2.0), biRoots(1.0, -5.0, 4.0).sorted())
}
```

В уроке 3 мы уже встречались с функциями с *получателем* `.toInt()` и `.toDouble()`. Функция `.sorted()` также требует наличия получателя: вызов `list.sorted()` создаёт список того же размера, что и исходный, но его элементы будут упорядочены по возрастанию.

Распространённые операции над списками

Перечислим некоторые операции над списками, имеющиеся в библиотеке языка Котлин:

1. `listOf(⋯)` — создание нового списка.
2. `list1 + list2` — сложение двух списков, сумма списков содержит все элементы их обоих.
3. `list + element` — сложение списка и элемента, сумма содержит все элементы `list` и дополнительно `element`
4. `list.size` — получение размера списка (Int).
5. `list.isEmpty()`, `list.isNotEmpty()` — получение признаков пустоты и непустоты списка (Boolean).
6. `list[i]` — индексация, то есть получение *элемента* списка с целочисленным *индексом* (номером) `i`. По правилам Котлина, в списке из `n` элементов они имеют индексы, начинающиеся с нуля: 0, 1, 2, …, последний элемент списка имеет индекс `n - 1`. То есть, при использовании записи `list[i]` должно быть справедливо `i >= 0 && i < list.size`. В противном случае выполнение программы будет прервано с ошибкой (использование индекса за пределами границ списка).
7. `list.sublist(from, to)` — создание списка меньшего размера (подсписка), в который войдут элементы списка `list` с индексами `from`, `from + 1`, …, `to - 2`, `to - 1`. Элемент с индексом `to` не включается.
8. `element in list` — проверка принадлежности элемента `element` списку `list`.
9. `for (element in list) { ⋯ }` — цикл **for**, перебирающий все элементы списка `list`.
10. `list.first()` — получение первого элемента списка (если список пуст, выполнение программы будет прервано с ошибкой).
11. `list.last()` — получение последнего элемента списка (аналогично).
12. `list.indexOf(element)` — поиск индекса элемента `element` в списке `list`. Результат этой функции равен -1, если элемент в списке отсутствует. В противном случае, при обращении к списку `list` по вычисленному индексу мы получим `element`.
13. `list.min()`, `list.max()` — поиск минимального и максимального элемента в списке.

14. `list.sum()` — сумма элементов в списке.
15. `list.sorted()`, `list.sortedDescending()` — построение отсортированного списка (по возрастанию или по убыванию) из имеющегося.
16. `list1 == list2` — сравнение двух списков на равенство. Списки равны, если равны их размеры и соответствующие элементы.

Мутирующие списки

Мутирующий список является разновидностью обычного, его тип определяется как `MutableList<ElementType>`. В дополнение к тем возможностям, которые есть у всех списков в Котлине, мутирующий список может изменяться по ходу выполнения программы или функции. Это означает, что мутирующий список позволяет:

1. Изменять своё содержимое операторами `list[i] = element`.
2. **Добавлять** элементы в конец списка, с увеличением размера на 1: `list.add(element)`.
3. **Удалять** элементы из списка, с уменьшением размера на 1 (если элемент был в списке): `list.remove(element)`.
4. **Удалять** элементы из списка по индексу, с уменьшением размера на 1: `list.removeAt(index)`.
5. **Вставлять** элементы в середину списка: `list.add(index, element)` — вставляет элемент `element` по индексу `index`, сдвигая все последующие элементы на 1, например `listOf(1, 2, 3).add(1, 7)` даст результат `[1, 7, 2, 3]`.

Для создания мутирующего списка можно использовать функцию `mutableListOf(…)`, аналогичную `listOf(…)`.

Рассмотрим пример. Пусть имеется исходный список целых чисел `list`. Требуется построить список, состоящий из его отрицательных элементов, порядок их в списке должен остаться прежним. Для этого требуется:

- создать пустой мутирующий список
- пройтись по всем элементам исходного списка и добавить их в мутирующий список, если они отрицательны
- вернуть заполненный мутирующий список

```
fun negativeList(list: List<Int>): List<Int> {
    val result = mutableListOf<Int>()
    for (element in list) {
        if (element < 0) {
            result.add(element)
        }
    }
    return result
}
```

Здесь промежуточная переменная `result` имеет тип `MutableList<Int>` (убедитесь в этом в IDE с помощью комбинации `Ctrl+Q`). Несмотря на это, мы можем использовать её в операторе **return** функции с результатом `List<Int>`. Происходит это потому, что тип `MutableList<Int>` является разновидностью типа `List<Int>`, то есть, любой мутирующий список является также и просто списком (обратное неверно — не любой список является мутирующим). На языке математики это означает, что ОДЗ (область допустимых значений) типа `MutableList<Int>` является **подмножеством** ОДЗ типа `List<Int>`.

В следующем примере функция принимает на вход уже **мутирующий** список целых чисел, и меняет в нём все положительные числа на противоположные по знаку:

```
fun invertPositives(list: MutableList<Int>) {
    for (i in 0 until list.size) {
        val element = list[i]
        if (element > 0) {
            list[i] = -element
        }
    }
}
```

Функция `invertPositives` не имеет результата. Это ещё один пример функции с побочным эффектом, которые уже встречались нам в первом уроке. Единственный смысл вызова данной функции — это изменение мутирующего списка, переданного ей как аргумента.

Обратите внимание на заголовок цикла **for**. Здесь мы вынуждены перебирать не элементы списка, а их индексы, причём запись `i in 0 until list.size` эквивалентна `i in 0..list.size - 1` (использование `until` несколько лучше, так как позволяет избежать лишнего вычитания единицы). Прямой перебор элементов списка в данном примере не проходит:

```
fun invertPositives(list: MutableList<Int>) {
    for (element in list) {
        if (element > 0) {
            element = -element // Val cannot be reassigned
        }
    }
}
```

Параметр цикла **for** является неизменяемым. Записать здесь `list[i] = -element` тоже не получится, так как индекс `i` нам неизвестен. Возможна, правда, вот такая, чуть более хитрая запись, перебирающая элементы и индексы одновременно:

```
fun invertPositives(list: MutableList<Int>) {
    for ((index, element) in list.withIndex()) {
        if (element > 0) {
            list[index] = -element
        }
    }
```

```
}
```

Использованная здесь функция `list.withIndex()` из исходного списка формирует другой список, содержащий *пары* (индекс, элемент), а цикл `for((index, element) in ⋯)` перебирает параллельно и элементы и их индексы. О том, что такое *пара* и как ей пользоваться в Котлине, мы подробнее поговорим позже.

В общем и целом, редко когда стоит пользоваться функциями, основной смысл которых заключается в изменении их параметров. Посмотрите, например, как выглядит тестовая функция для `invertPositives`:

```
fun invertPositives() {
    val list1 = mutableListOf(1, 2, 3)
    invertPositives(list1)
    assertEquals(listOf(-1, -2, -3), list1)
    val list2 = mutableListOf(-1, 2, 4, -5)
    invertPositives(list2)
    assertEquals(listOf(-1, -2, -4, -5), list2)
}
```

Если ранее у нас одна проверка всегда занимала одну строку, то в этом примере она занимает три строки из-за необходимости создания промежуточных переменных `list1` и `list2`. Кроме этого, факт изменения `list1`, `list2` при вызове `invertPositives` склонен ускользать от внимания читателя, затрудняя понимание программы.

Функции высшего порядка над списками

Вернёмся ещё раз к задаче формирования списка из отрицательных чисел в исходном списке. На Котлине, данная задача допускает ещё и такое, очень короткое решение:

```
fun negativeList(list: List<Int>) = list.filter { it < 0 }
```

Это короткое решение, однако, является довольно ёмким в плане его содержания. Попробуем в нём разобраться.

`list.filter` — это один из примеров так называемой *функции высшего порядка*. Суть функции `filter` в том, что она фильтрует содержимое списка-получателя. Её результатом также является список, содержащий все элементы списка-получателя, удовлетворяющие определённому условию.

Как же она это делает и что такое вообще *функция высшего порядка*? Это тоже *функция*, которая, однако, принимает в качестве параметра **другую функцию**. Более подробная запись вызова `filter` выглядела бы так:

```
fun negativeList(list: List<Int>) = list.filter(fun(it: Int) = it < 0)
```

Функция-аргумент в данном случае должна иметь параметр `it` того же типа, что и элементы списка, и результат типа `Boolean`. В этой записи она отличается от обычной функции только отсутствием имени. Функция `filter` передаёт функции-аргументу каждый элемент списка. Если функция-аргумент вернула `true`, элемент помещается в список-результат, если `false` — он фильтруется.

Более короткая запись `list.filter({ it < 0 })` использует так называемую *лямбду* `{ it < 0 }` в качестве аргумента функции `filter`. Этот краткий синтаксис не включает в себя не только имени функции, но и ключевого слова **fun**, а также явного указания имён и типов параметров. При этом предполагается, что:

- параметр называется **it**; если параметру хочется дать другое имя, лямбда записывается как, например, `{ element -> element < 0 }`
- тип параметра — ровно тот, который требуется функции высшего порядка, для `filter` это тип элементов списка
- тип результата — опять-таки ровно тот, что требуется
- в фигурные скобки помещается блок, определяющий результат функции; в идеале он состоит из одного оператора, в данном случае это `it < 0`

Наконец, в том случае, если *лямбда* является **последним** аргументом функции, при вызове функции разрешается вынести её за круглые скобки: `list.filter() { it < 0 }`. Если других аргументов у функции нет, разрешается опустить в этой записи круглые скобки, получив запись из исходного примера: `list.filter { it < 0 }`

Функции высшего порядка с первого взгляда могут показаться очень сложными, но реально это довольно простая вещь, позволяющая свести запись алгоритмов к более компактной. Рассмотрим другую типичную задачу: из имеющегося массива целых чисел сформировать другой массив, содержащий квадраты чисел первого массива. Задача решается в одну строчку с помощью функции высшего порядка `map`:

```
fun squares(list: List<Int>) = list.map { it * it }
```

`list.map` предназначена для преобразования списка `list` в другой список такого же размера, при этом над каждым элементом списка `list` выполняется преобразование, указанное в функции-аргументе `map`. Тип параметра функции-аргумента совпадает с типом элементов списка `list`, а вот тип результата может быть произвольным. Например, преобразование `list.map { "$it" }` создаст из списка чисел вида [0, 1, 2] список строк ["0", "1", "2"].

Чуть более сложный пример: проверка числа на простоту.

```
fun isPrime(n: Int) = n >= 2 && (2..n/2).all { n % it != 0 }
```

Функция высшего порядка `all` в данном примере вызывается для получателя-интервала: `2..n/2`. Применима она и для списка, как и для любого другого объекта, элементы которого можно перебрать с помощью **for**. Функция `all` имеет логический результат и возвращает **true**, если функция-аргумент возвращает **true** для ВСЕХ элементов списка. Тип параметра

функции-аргумента совпадает с типом элементов списка, тип результата — `Boolean`. Аналогично можно было бы применить функцию высшего порядка `any`:

```
fun isNotPrime(n: Int) = n < 2 || (2..n/2).any { n % it == 0 }
```

Функция высшего порядка `any` возвращает **true**, если функция-аргумент возвращает **true** ХОТЯ БЫ для одного элемента списка.

Наконец, функция высшего порядка `fold` предназначена для "сворачивания" списка в один элемент или значение. Например:

```
fun multiplyAll(list: List<Int>) = list.fold(1.0) {
    previousResult, element -> previousResult * element
}
```

Функция `fold` работает следующим образом. Изначально она берёт свой первый аргумент (в данном примере 1.0) и сохраняет его как текущий результат. Далее перебираются все элементы списка получателя и для каждого из них применяется указанная лямбда, которая из текущего результата `previousResult` с предыдущего шага и очередного элемента `element` делает текущий результат этого шага (в данном примере предыдущий результат домножается на очередной элемент). По окончании элементов списка последний текущий результат становится окончательным. В данном примере результатом будет произведение всех элементов списка (или 1.0, если список пуст).

Строки

Строки `String` во многих отношениях подобны спискам, хотя формально и не являются ими. Список состоит из однотипных элементов, к которым можно обращаться по индексу и перебирать с помощью цикла **for**. Строки же точно так же состоят из символов `Char`, к которым также можно обращаться по индексу и которые также можно перебирать с помощью цикла **for**.

Напомним, что строковый *литерал* (явно указанная строка) в Котлине записывается в двойных кавычках. Переменную `name` произвольного типа можно преобразовать в строку, используя запись `"$name"` — строковый шаблон, или чуть более сложную запись `name.toString()` с тем же самым результатом.

Как мы видим, `$` внутри строкового литерала имеет специальный смысл — вместо `$name` в строку будет подставлено содержимое переменной `name`. Как быть, если мы хотим просто включить в строку символ доллара? В этом случае следует применить так называемое *экранирование*, добавив перед символом доллара символ `\`. Например: `"The price is 9.99 \$"`.

Экранирование может применяться и для добавления в строку различных специальных символов, не имеющих своего обозначения либо имеющих специальный смысл внутри строкового литерала. Например: `\n` — символ новой строки, `\t` — символ табуляции, `\\` — символ "обратная косая черта", `\"` — символ "двойная кавычка".

Строки в Котлине являются неизменяемыми, то есть изменить какой-либо символ в уже созданной строке нельзя, можно только создать новую. В этом смысле они аналогичны немутирующим спискам `List`.

Перечислим наиболее распространённые операции над строками:

1. `string1 + string2` — сложение или конкатенация строк, приписывание второй строки к первой.
2. `string + char` — сложение строки и символа (с тем же смыслом).
3. `string.length` — длина строки, то есть количество символов в ней.
4. `string.isEmpty()`, `string.isNotEmpty()` — получение признаков пустоты и непустоты строки (Boolean).
5. `string[i]` — индексация, то есть получение символа по целочисленному *индексу* (номеру) `i` в диапазоне от 0 до `string.length - 1`.
6. `string.substring(from, to)` — создание строки меньшего размера (подстроки), в который войдут символы строки `string` с индексами `from`, `from + 1`, ..., `to - 2`, `to - 1`. Символ с индексом `to` не включается.
7. `char in string` — проверка принадлежности символа `char` строке `string`.
8. `for (char in list) { ··· }` — цикл **for**, перебирающий все символы строки `string`.
9. `string.first()` — получение первого символа строки.
10. `string.last()` — получение последнего символа строки.
11. `string.indexOf(char, startFrom)` — найти индекс первого символа `char` в строке, начиная с индекса `startFrom`.
12. `string.lastIndexOf(char, startFrom)` — найти индекс первого символа `char` в строке, идя с конца и начиная с индекса `startFrom`.
13. `string.toLowerCase()` — преобразование строки в нижний регистр (то есть, замена прописных букв строчными).
14. `string.toUpperCase()` — преобразование строки в верхний регистр (замена строчных букв прописными).
15. `string.capitalize()` — замена ПЕРВОЙ буквы строки прописной.
16. `string.trim()` — удаление из строки пробельных символов в начале и конце: `" ab c "` преобразуется в `"ab c"`

В качестве примера рассмотрим функцию, проверяющую, является ли строка палиндромом. В палиндроме первый символ должен быть равен последнему, второй предпоследнему и т.д. Пример палиндрома: "А роза упала на лапу Азора". Из этого примера видно, что одни и те же буквы в разном регистре следует считать равными с точки зрения данной задачи. Кроме этого, не следует принимать во внимание пробелы. Решение на Котлине может быть таким:

```
fun isPalindrome(str: String): Boolean {
    val lowerCase = str.toLowerCase().filter { it != ' ' }
    for (i in 0..lowerCase.length / 2) {
```

```
        if (lowerCase[i] != lowerCase[lowerCase.length - i - 1]) return false
    }
    return true
}
```

Обратите внимание, что мы с самого начала переписываем исходную строку `str` в промежуточную переменную `lowerCase`, преобразуя все буквы в нижний регистр и удаляя из строки все пробелы. Функция `filter` работает для строк точно так же, как и для списков — в строке оставляются только те символы, для которых функция-аргумент `{ it != ' '}` вернёт **true**. Затем мы перебираем символы в первой половине строки, сравнивая каждый из них с символом из второй половины. Символ с индексом 0 (первый) должен соответствовать символу с индексом `length - 1` (последнему) и так далее.

Преобразование из списка в строку

Очень часто используемой в Котлине является сложная функция преобразования списка в строку `joinToString()`. Её заголовок выглядит примерно так:

```
fun <T> List<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        limit: Int = -1,
        truncated: String = "...",
        transform: (T) -> String = { "$it" }
): String { ... }
```

Получателем данной функции может быть список с произвольным типом элементов: `List<T>`. Такая запись описывает так называемую настраиваемую функцию, о них мы будем говорить подробнее позже.

Все пять параметров этой функции имеют так называемые *значения по умолчанию*. Это значит, что при желании мы можем вызвать эту функцию вообще не указывая аргументов. Например, `listOf(1, 2, 3).joinToString()` даст нам следующий результат: `"1, 2, 3"` — выводя в строку все элементы списка через запятую. Возможна, однако, более тонкая настройка вывода:

- параметр `separator` задаёт разделитель между элементами
- параметр `prefix` задаёт строку, которая выводится перед самым первым элементом списка (префикс)
- аналогично, параметр `postfix` задаёт строку, которая выводится после самого последнего элемента списка (постфикс)
- параметр `limit` задаёт максимальное количество выводимых элементов. Значение -1 соответствует неограниченному количеству элементов, но, скажем, вызов `listOf(1, 2, 3).joinToString(", ", "", "", 1)` будет иметь результат `"1, …"` вместо результата по умолчанию `"1, 2, 3"`

- параметр `truncated` используется, если задан `limit`, и заменяет все элементы списка, которые не поместились в строке
- параметр `transform` задаёт способ преобразования каждого из элементов списка в строку — по умолчанию это `"$it"` для элемента списка `it`, может быть изменён с помощью лямбды (см. функции высшего порядка выше)

Рассмотрим простой пример: необходимо написать функцию, которая по заданному списку целых чисел вида `[3, 6, 5, 4, 9]` сформирует строку, содержащую пример их суммирования: `"3 + 6 + 5 + 4 + 9 = 26"`. На Котлине это записывается так:

```
fun buildSumExample(list: List<Int>) = list.joinToString(separator = " + ", postfix =
" = ${list.sum()}")
```

В данном случае нам требуется вызов функции `joinToString`, все параметры которой имеют некоторые значения по умолчанию, то есть не требуют обязательного указания при вызове. Нам требуется указать разделитель чисел `" + "` и в конце вывода добавить знак `=` с приписанной к нему суммой чисел из списка. Для этого нам необходимо указать значения параметров `separator` и `postfix`, при этом остальные параметры мы указывать не хотим. В этом случае используются так называемые *именованные аргументы*, например: `separator = " + "`. Эта запись указывает, что аргумент `" + "` соответствует параметру функции `separator`. Если бы имена `separator` и `postfix` не указывались, возникла бы путаница, поскольку неясно, какому именно из строковых параметров функции соответствует тот или иной аргумент вызова.

Массивы

Массив **Array** — ещё один тип, предназначенный для хранения и модификации некоторого количества однотипных элементов. С точки зрения возможностей, массив похож на мутирующий список **MutableList**; главным их отличием является отсутствие возможности изменять свой размер — для массивов отсутствуют функции `add` и `remove`.

Для обращения к элементу массива служит оператор индексации: `array[i]`, причём есть возможность как читать содержимое массива, так и изменять его. Для создания массива, удобно использовать функцию `arrayOf()`, аналогичную `listOf()` для списков.

Почти все возможности, имеющиеся для списков, имеются и для массивов тоже. Исключением являются функции для создания подсписков `sublist`. Также, массивы не следует сравнивать на равенство с помощью `array1 == array2`, поскольку во многих случаях такое сравнение даёт неверный результат (подробности про это — в разделе 4.5). Массив можно преобразовать к обычному списку с помощью `array.toList()` либо к мутирующему списку с помощью `array.toMutableList()`. Список, в свою очередь, можно преобразовать к массиву с помощью `list.toTypedArray()`.

В целом, при написании программ на Котлине почти нет случаев, когда массивы использовать необходимо. Одним из немногих примеров является главная функция, параметр которой имеет тип `Array<String>` — через него в программу передаются аргументы командной строки.

Параметры переменной длины

В некоторых случаях бывает удобно при вызове функции не передавать ей аргумент типа **List** или **Array**, а просто перечислить элементы этого списка при вызове. Например, чтобы сформировать список квадратов чисел от 1 до 3 с помощью рассмотренной выше функции `squares`, нам пришлось бы вызвать данную функцию как `squares(listOf(1, 2, 3))`. Вызов выглядел бы проще без прямого указания `listOf`: просто как `squares(1, 2, 3)`. Для поддержки этой возможности программисты придумали *параметры переменной длины*, в Котлине они называются **vararg**.

```
fun squares(vararg array: Int) = squares(array.toList())
```

При вызове подобной функции вместо параметра `array` может быть подставлено любое (в том числе ноль) количество аргументов указанного типа, в данном случае — **Int**.

Я не случайно назвал параметр именно `array`: функция может использовать такой параметр, как будто это массив соответствующего типа — в данном случае **Array<Int>**. В теле функции мы используем вызов `array.toList()`, чтобы свести задачу к уже решённой — функция `squares`, принимающая аргумент типа **List<Int>**, у нас уже имеется. Если необходимо подставить вместо **vararg**-параметра уже имеющийся массив, следует использовать *оператор раскрытия* `*`:

```
fun squares(array: Array<Int>) = squares(*array)
```

Здесь мы вызываем уже написанную функцию `squares`, принимающую параметр переменной длины.

В библиотеке Котлина имеется довольно много функций, имеющих **vararg**-параметр. Два классических примера были рассмотрены в этом разделе — это функции `listOf(…)` и `mutableListOf(…)`.

Упражнения

Откройте файл `src/lesson4/task1/List.kt` в проекте `KotlinAsFirst`. Выберите любую из задач в нём. Придумайте её решение и запишите его в теле соответствующей функции.

Откройте файл `test/lesson4/task1/Tests.kt`, найдите в нём тестовую функцию — её название должно совпадать с названием написанной вами функции. Запустите тестирование, в случае обнаружения ошибок исправьте их и добейтесь прохождения теста. Подумайте, все ли необходимые проверки включены в состав тестовой функции, добавьте в неё недостающие проверки.

Решите ещё хотя бы одну задачу из урока 4 на ваш выбор, попробуйте применить в процессе решения известные вам функции высшего порядка. Убедитесь в том, что можете решать такие задачи уверенно и без посторонней помощи. Попробуйте в решении хотя бы одной задачи применить функции высшего порядка.

По возможности решите одну из задач, помеченных как "Сложная" или "Очень Сложная". Если вам потребуется преобразование списка в строку, примените `list.joinToString()`. Имейте в виду, что последняя задача, функция **russian**, ДЕЙСТВИТЕЛЬНО очень сложная, и для её решения может потребоваться значительное время.

Переходите к следующему разделу.