## Практическая работа 4. Списки в Котлин

## Теоретический материал

## Зачем нужны списки

У сильной независимой женщины Наташи проживает сорок (!) котов. Аккуратная хозяйка решила записать имена котов в тетрадку.

### **Мои ненаглядные коты**

* Мурзик
* Рыжик
* Барсик
* И так далее...

Наташа немного знакома с программированием и решила вести учёт в электронном виде, написав собственную программу под Android.

Первоначальный вариант был следующим.

val cat1 = "Мурзик"

val cat2 = "Рыжик"

val cat3 = "Барсик"

println(cat1)

println(cat2)

println(cat3)

В какой-то момент, Наташе надоело писать такой код. Для каждого кота нужно заводить отдельную переменную, при этом сложно следить за порядком, если захочется поменять местами Рыжика и Барсика, сделав их котами под номерами 3 и 2.

К счастью, существуют коллекции. Список является распространённым вариантом коллекции. Создадим список котов через функцию **listOf()**.

val myLovelyCats = listOf( // Переменная Мои ненаглядные коты

"Мурзик", // первый кот в списке

"Рыжик",

"Барсик", // последний кот в списке

)

println(myLovelyCats)

>> [Мурзик, Рыжик, Барсик]

Все элементы списка находятся в том порядке, в котором их заносили.

**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 iнам 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, list2при вызове invertPositives склонен ускользать от внимания читателя, затрудняя понимание программы.

**Задания**

1. Создать список, состоящий из дисциплин с их кодами (МДК или ОП), которые у вас преподаются и вывести его на экран

* Вывести на экран только те дисциплины, у которых код начинается с МДК
* вывести количество дисциплин

1. Создать список, состоящий из фамилий студентов вашей группы с указанием их возраста.

* вывести фамилии студентов, который старше 18.
* вывести фамилии несовершеннолетних студентов в группе.
* вывести фамилии студентов, которые начинаются на «К»
* отсортировать значения в списке в алфавитном порядке и вывести их на экран

1. Создать изменяемый список, состоящий из фамилий преподавателей, которые у вас ведут занятия

* удалить в нем преподавателя с фамилией Иванов
* заменить в списке фамилию Петров на Сидоров