Коллекции и последовательности

Изменяемые и неизменяемые коллекции

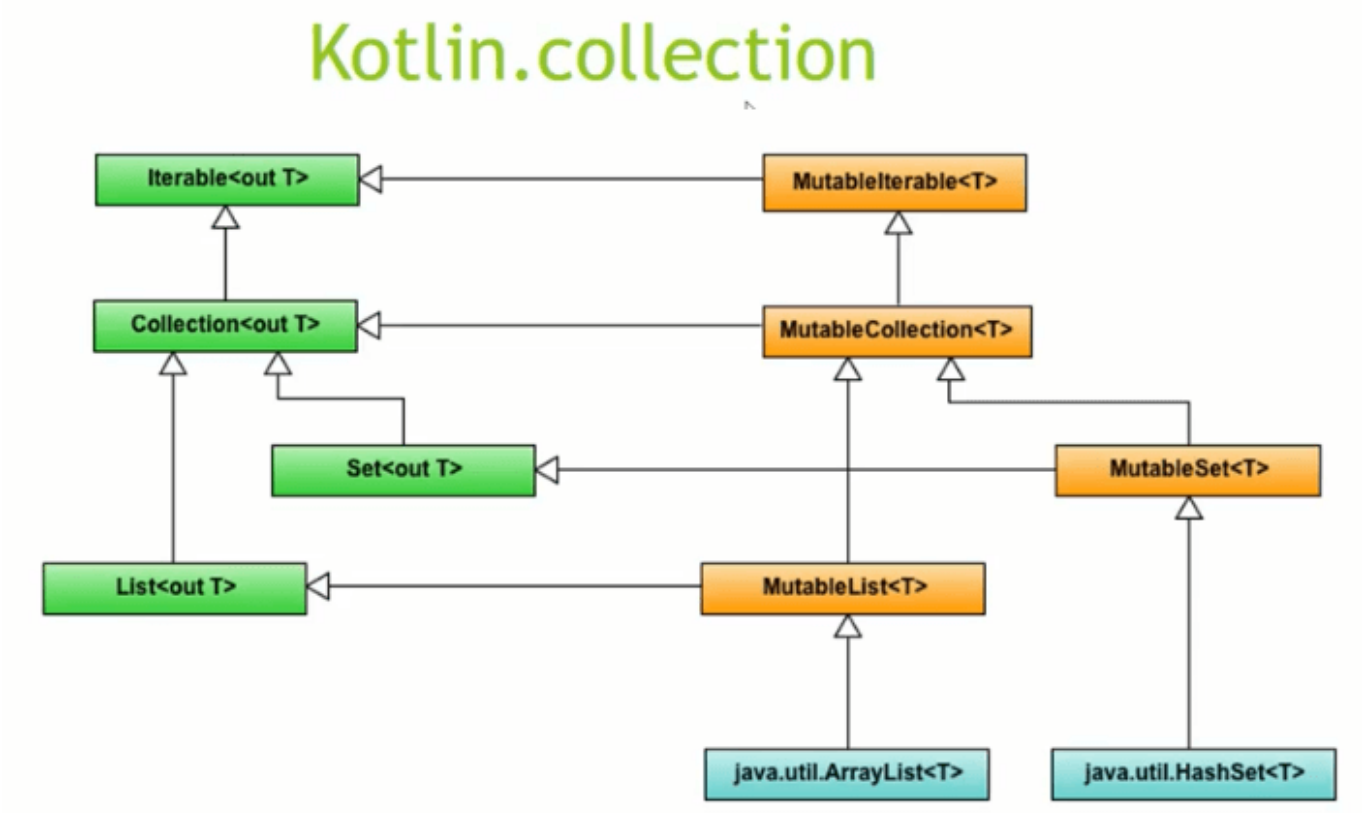
Коллекции представляют контейнеры, которые используются для хранения данных. В зависимости от типа коллекции различаются способы работы с данными.

Kotlin не имеет собственной библиотеки коллекций и полностью полагается на классы коллекций, которые предоставляет Java. В то же время эти коллекции в Kotlin расширяются дополнительными возможностями.

Так, в Kotlin коллекции разделяются на изменяемые (mutable) и неизменяемые (immutable) коллекции.

Mutable-коллекция может изменяться, в нее можно добавлять, в ней можно изменять, удалять элементы. Immutable-коллекция также поддерживает добавление, замену и удаление данных, однако в процессе подобных операций коллекция будет заново пересоздаваться.

Все коллекции в Kotlin располагаются в пакете kotlin.collections. Полный список интерфейсов и классов, которые представляют коллекции, можно найти здесь.



Неизменяемые коллекции

На вершине иерархии находится интерфейс Iterable, который определяет функцию итератор для перебора коллекции.

Основным интерфейсом, который позволяет работать с коллекциями, является kotlin.Collection. Данный интерфейс определяет функциональность для перебора элементов, проверки наличия элементов,

чтения данных. Однако он не предоставляет возможности по добавлению и удалению данных. Его основные компоненты:

- size: возвращает количество элементов в коллекции
- isEmpty(): возвращает true, если коллекция пустая
- contains(element): возвращает true, если коллекция содержит element
- containsAll(collection): возвращает true, если коллекция содержит элементы коллекции collection

Этот интерфейс расширяется другими интерфейсами, которые представляют неизменяемые коллекции - List, который представляет обычный список, и Set, который представляет неупорядоченную коллекцию элементов, не допускающую дублирования элементов.

Особняком стоит интерфейс Map. Он не расширяет Collection и представляет набор пар ключ-значение, где каждому ключу сопоставляет некоторое значение. Все ключи в коллекции являются уникальными.

Изменяемые коллекции

Все изменяемые коллекции реализуют интерфейс MutableIterable. Он представляет функцию итератора для перебора коллекции.

Для изменения данных в Kotlin также определен интерфейс kotlin.MutableCollection, который расширяет интерфейс kotlin.Collection и предоставляет методы для удаления и добавления элементов. В частности:

- add(element): добавляет элемент
- remove(element): удаляет элемент
- addAll(elements): добавляет набор элементов
- removeAll(elements): удаляет набор элементов
- clear(): удаляет все элементы из коллекции

Этот интерфейс расширяется интерфейсами MutableList, который представляет изменяемый список, и MutableSet, который предствляет изменяемую неупорядоченную коллекцию уникальных элементов.

И еще одна изменяемая коллекция представлена интерфейсом MutableMap - изменяемая карта, где каждый элемент представляет пару ключ-значение.

Операции с коллекциями

Кроме выше рассмотренных методов интерфейс Iterable также предоставляет ряд функций для выполнения различных операций над коллекциями. Рассмотрим основные операции:

- all(predicate: (T) -> Boolean): Boolean

возвращает true, если все элементы соответствуют предикату, который передается в функцию в качестве параметра

- any(): Boolean

возвращает true, если коллекция содержит хотя бы один элемент

Дополнительная версия возвращает true, если хотя бы один элемент соответствуют предикату, который передается в функцию в качестве параметра

- any(predicate: (T) -> Boolean): Boolean
- asSequence(): Sequence

создает из коллекции последовательность

- average(): Double

возвращает среднее значение для числовой коллекции типов Byte, Int, Short, Long, Float, Double

- chunked(size: Int): List<List>

расщепляет коллекцию на список, который состоит из объектов List, параметр size устанавливает максимальное количество элементов в каждом из списков

Дополнительная версия в качестве второго параметра получает функцию преобразования, которая преобразует каждый список в элемент новой коллекции

- chunked(size: Int,transform: (List) -> R): List
- contains(element: T): Boolean

возвращает true, если коллекция содержит элемент element

- count(): Int

возвращает количество элементов в коллекции

Дополнительная версия возвращает количество элементов, которые соответствуют предикату

- count(predicate: (T) -> Boolean): Int
- distinct(): List

возвращает новую коллекцию, которая содержит только уникальные элементы

- distinctBy(selector: (T) -> K): List

возвращает новую коллекцию, которая содержит только уникальные элементы с учетом функции селектора, которая передается в качестве параметра

- drop(n: Int): List

возвращает новую коллекцию, которая содержит все элементы за исключением первых n элементов

- dropWhile(predicate: (T) -> Boolean): List

возвращает новую коллекцию, которая содержит все элементы за исключением первых элементов, которые соответствуют предикату

- elementAt(index: Int): T

возвращает элемент по индексу index. Если индекс выходит за пределы коллекции, то генерируется исключение типа IndexOutOfBoundsException

- elementAtOrElse(index: Int, defaultValue: (Int) -> T): T

возвращает элемент по индексу index. Если индекс выходит за пределы коллекции, то возвращается значение, устанавливаемое функцией из параметра defaultValue

- elementAtOrNull(index: Int): T?

возвращает элемент по индексу index. Если индекс выходит за пределы коллекции, то возвращается null

- filter(predicate: (T) -> Boolean): List

возвращает новую коллекцию из элементов, которые соответствуют предикату

- filterNot(predicate: (T) -> Boolean): List

возвращает новую коллекцию из элементов, которые НЕ соответствуют предикату

- filterNotNull(): List

возвращает новую коллекцию из элементов, которые не равны null

- find(predicate: (T) -> Boolean): T?

возвращает первый элемент, который соответствует предикату. Если элемент не найден, то возвращается null

- findLast(predicate: (T) -> Boolean): T?

возвращает последний элемент, который соответствует предикату. Если элемент не найден, то возвращается null

- first(): T

возвращает первый элемент коллекции

Дополнительная версия возвращает первый элемент, которые соответствует предикату

- first(predicate: (T) -> Boolean): T

Если элемент не найден, то генерируется исключение типа NoSuchElementException

- firstOrNull(): T?

возвращает первый элемент коллекции

Дополнительная версия возвращает первый элемент, которые соответствует предикату

- firstOrNull(predicate: (T) -> Boolean): T?

Если элемент не найден, то возвращается null

- flatMap(transform: (T) -> List): List

преобразует коллекцию элементов типа T в коллекцию элементов типа R, используя функцию преобразования, которая передается в качестве параметра

- fold(initial: R, operation: (acc: R, T) -> R): R

Возвращает значение, которое является результатом действия функции operation над каждым элементом коллекции. Первый параметр функции operation - результат работы функции над предыдущим элементом коллекции (при первом вызове - значение из параметра initial), в второй параметр - текущий элемент коллекции.

- forEach(action: (T) -> Unit)

Выполняет для каждого элемента коллекции действие action.

- groupBy(keySelector: (T) -> K): Map<K, List>

Группирует элементы по ключу, который возвращается функцией keySelector. Результат функции карта Map, где ключ - собственно ключ элементов, а значение - список List из элементов, которые соответствуют этому ключу

Дополнительная версия принимает функцию преобразования элементов:

- groupBy(keySelector: (T) -> K, valueTransform: (T) -> V): Map<K, List>

indexOf(element: T): Int

Возвращает индекс первого вхождения элемента element. Если элемент не найден, возвращается -1

- indexOfFirst(predicate: (T) -> Boolean): Int

Возвращает индекс первого элемента, который соответствует предикату. Если элемент не найден, возвращается -1

- indexOfLast(predicate: (T) -> Boolean): Int

Возвращает индекс последнего элемента, который соответствует предикату. Если элемент не найден, возвращается -1

- intersect(other: Iterable): Set

Возвращает все элементы текущей коллекции, которые есть в коллекции other

- joinToString(): String

Генерирует из коллекции строку

- last(): T

возвращает последний элемент коллекции

Дополнительная версия возвращает последний элемент, которые соответствует предикату

- last(predicate: (T) -> Boolean): T

Если элемент не найден, то генерируется исключение типа NoSuchElementException

- lastOrNull(): T?

возвращает последний элемент коллекции

Дополнительная версия возвращает последний элемент, которые соответствует предикату

- lastOrNull(predicate: (T) -> Boolean): T?

Если элемент не найден, то возвращается null

- lastIndexOf(element: T): Int

Возвращает последний индекс элемента element. Если элемент не найден, возвращается -1

- map(transform: (T) -> R): List

Применяет к элементам коллекции функцию трансформации и возвращает новую коллекцию из новых элементов

- mapIndexed(transform: (index: Int, T) -> R): List

Применяет к элементам коллекции и их индексам функцию трансформации и возвращает новую коллекцию из новых элементов

- mapNotNull(transform: (T) -> R?): List

Применяет к элементам коллекции функцию трансформации и возвращает новую коллекцию из новых элементов, которые не равны null

- maxOf(selector: (T) -> Double): Double

Возвращает максимальное значение на основе селектора

- maxOfOrNull(selector: (T) -> Double): Double?

Возвращает максимальное значение на основе селектора. Если коллекцию пуста, возвращается null

- maxOrNull(): Double?

Возвращает максимальное значение. Если коллекцию пуста, возвращается null

- minOf(selector: (T) -> Double): Double

Возвращает минимальное значение на основе селектора

- minOfOrNull(selector: (T) -> Double): Double?

Возвращает минимальное значение на основе селектора. Если коллекцию пуста, возвращается null

- minOrNull(): Double?

Возвращает минимальное значение. Если коллекцию пуста, возвращается null

- minus(element: T): List

Возвращает новую коллекцию, которая содержит все элементы текущей за исключением элемента element.

Имеет разновидности, которую позволяют исключить из коллекции наборы элементов:

```
minus(elements: Array<T>): List<T>
minus(elements: Iterable<T>): List<T>
minus(elements: Sequence<T>): List<T>
```

- plus(element: T): List

Возвращает новую коллекцию, которая содержит все элементы текущей за исключением плюс элемент element.

Имеет разновидности, которую позволяют включить в коллекцию наборы элементов:

```
plus(elements: Array<T>): List<T>
plus(elements: Iterable<T>): List<T>
plus(elements: Sequence<T>): List<T>
```

- reduce(operation: (acc: S, T) -> S): S

Возвращает значение, которое является результатом действия функции operation над каждым элементом коллекции. Первый параметр функции operation - результат работы функции над предыдущим элементом коллекции, в второй параметр - текущий элемент коллекции.

- shuffled(): List

Условно перемешивает коллекцию

- sorted(): List

Сортирует коллекцию по возрастанию

- sortedBy(selector: (T) -> R?): List

Сортирует коллекцию по возрастанию на основе селектора

- sortedByDescending(selector: (T) -> R?): List

Сортирует коллекцию по убыванию на основе функции-селектора

- sortedDescending(): List

Сортирует коллекцию по убыванию

- sum(): Int

Возвращает сумму элементов коллекции.

- subtract(other: Iterable): Set

Возвращает набор элементов, которые есть в текущей коллекции и отсутствуют в коллекции other.

- sum(): Int

Возвращает сумму элементов коллекции

- sumOf(selector: (T) -> Int): Int

Возвращает сумму элементов коллекции на основе функции-селектора

- take(n: Int): List

Возвращает новую коллекцию, которая содержит n первых элементов текущей коллекции

- takeWhile(predicate: (T) -> Boolean): List

Возвращает новую коллекцию, которая содержит n первых элементов текущей коллекции, соответствующих функции-предикату

- toHashSet(): HashSet

Создает из коллекции объект HashSet

- toList(): List

Создает из коллекции объект List

- toMap(): Map<K, V>

Создает из коллекции объект Map

- toSet(): Set

Создает из коллекции объект Set

- union(other: Iterable): Set

Возвращает набор уникальных элементов, которые есть в текущей коллекции и коллекции other

List

List представляет последовательный список элементов. При этом List представляет неизменяемую (immutable) коллекцию, которая в основном только обеспечивает получение элементов по позиции.

Интерфейс List расширяет интерфейс Collection, поэтому перенимает его возможности.

Для создания объекта List применяется метод listOf():

```
var numbers = listOf(1, 2, 3, 4, 5)     // объект List<Int>
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")   // объект List<String>
```

Списки поддерживают перебор с помощью цикла for, кроме для списка по умолчанию задача реализация toString, которая выводит все элементы списка в удобочитаемом виде:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")

for(person in people) println(person)
println(people) // [Tom, Sam, Kate, Bob, Alice]
```

Методы списков

Кроме унаследованных методов класс List имеет ряд специфичных. Рассмотрим некоторые из них.

Для получения элемента по индексу можно применять метод get(index), который возвращает элемент по индексу

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
val first = people.get(0)
val second = people.get(1)
println(first)      // Tom
println(second)     // Sam
```

Вместо метода get для обращения по индексу можно использовать квадратные скобки []:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
val first = people[0]
val second = people[1]
println(first)      // Tom
println(second)     // Sam
```

Однако, если индекс выходит за границы списка, то при использовании метода get() и квадратных скобок генерируется исключение. Чтобы избежать подобной ситуации, можно применять метод getOrNull(), который возвращает null, если индекс находится вне границ списка:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
val first = people.getOrNull(0)
val tenth = people.getOrNull(10)
println(first)      // Tom
println(tenth)     // null
```

Либо в качестве альтернативы можно применять метод getOrElse():

```
getOrElse(index: Int, defaultValue: (Int) -> T): T
```

Первый параметр представляет индекс, а второй параметр - функция, которая получает запрошенный индекс и возвращает значение, которое возвращается, если индекс выходит за границы списка:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
val first = people.getOrElse(0){"Undefined"}
val seventh = people.getOrElse(7){"Invalid index $it"}
val tenth = people.getOrElse(10){"Undefined"}

println(first)      // Tom
println(seventh)    // Invalid index 7
println(tenth)     // Undefined
```

Получение части списка

Метод subList() возвращает часть списка и в качестве параметров принимает начальный и конечный индексы извлекаемых элементов:

```
subList(fromIndex: Int, toIndex: Int): List<E>
```

Например, получим подсписок с 1 по 4 индексы:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val subPeople = people.subList(1, 4)
println(subPeople)      // [Sam, Kate, Bob]
```

Изменяемые списки

Изменяемые списки представлены интерфейсом MutableList. Он расширяет интерфейс List и позволяют добавлять и удалять элементы. Данный интерфейс реализуется классом ArrayList.

Для создания изменяемых списков можно использовать ряд методов:

- arrayListOf(): создает объект ArrayList
- mutableListOf(): создает объект MutableList

Создание изменяемых списков:

```
var numbers : ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
var numbers2: MutableList<Int> = mutableListOf(5, 6, 7)
```

Если необходимо добавлять или удалять элементы, то надо использовать методы MutableList:

- add(index, element): добавлят элемент по индексу

- add(element): добавляет элемент
- addAll(collection): добавляет коллекцию элементов
- remove(element): удаляет элемент
- removeAt(index): удаляет элемент по индексу
- clear(): удаляет все элементы коллекции

```
fun main() {

    val numbers1 : ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
    numbers1.add(4)
    numbers1.clear()

    val numbers2: MutableList<Int> = mutableListOf(5, 6, 7)

    numbers2.add(12)
    numbers2.add(0, 23)
    numbers2.addAll(0, listOf(-3, -2, -1))
    numbers2.removeAt(0)
    numbers2.remove(5)

    for (n in numbers2){ println(n) }
}
```

Set

Интерфейс Set представляет неупорядоченный набор объектов, который хранит только уникальные объекты. Интерфейс Set представляет неизменяемый (immutable) набор. Set расширяет интерфейс Collection и соответственно все его методы.

Для создания неизменяемого (immutable) набора используется функция setOf().

```
val numbers = setOf(5, 6, 7)            // объект Set<Int>
val people = setOf("Tom", "Sam", "Bob") // объект Set<String>
```

Для перебора Set можно применять цикл for:

```
val people = setOf("Tom", "Sam", "Bob")

for(person in people) println(person)
println(people) // [Tom, Sam, Bob]
```

Set реализует метод toString таким образом, что возвращает в удобочитабельном виде все элементы в виде строки

Причем, поскольку Set представляет набор уникальных объектов, то даже если мы передадим через функцию setOf() повторяющиеся значения, то в наборе все равно будут только уникальные значения:

```
val numbers = setOf(5, 6, 7, 5, 6)
val people = setOf("Tom", "Sam", "Bob", "Tom")

println(numbers) // [5, 6, 7]
println(people) // [Tom, Sam, Bob]
```

Методы Set

Рассмотрим некоторые специфичные операции Set. Прежде всего это методы для операций с множествами:

- union: объединение множеств
- intersect: пересечение множеств (возвращает элементы, которые есть в обоих множествах)
- subtract: вычиьтание множеств (возвращает элементы, которые есть в первом множестве, но отсутствуют во втором)

Хотя эти операции могут применяться и к спискам List, но возвращают они всегда объект Set и более уместны для множеств Set.

```
val people = setOf("Tom", "Sam", "Bob", "Mike")
val employees = setOf("Tom", "Sam", "Kate", "Alice")

//  объединение множеств
val all = people.union(employees)
// пересечение множеств
val common = people.intersect(employees)
// вычитание множеств
val different = people.subtract(employees)

println(all)        // [Tom, Sam, Bob, Mike, Kate, Alice]
println(common)     // [Tom, Sam]
println(different)  // [Bob, Mike]
```

Причем данные методы можно применять как обычные операции:

```
//объединение множеств
val all = people union employees

//пересечение множеств
val common = people intersect employees
```

```
//вычитание множеств
val different = people subtract employees
```

Изменяемые коллекции

Изменяемые (mutable) наборы представлены интерфейсом MutableSet, который расширяет интерфейсы Set и MutableCollection, соответственно поддерживает методы по изменению коллекции.

Для создания изменяемых (mutable) наборов применяется функция mutableSetOf().

```
val numbers: MutableSet<Int> = mutableSetOf(35, 36, 37)
```

Интерфейс MutableSet реализуется следующими типами изменяемых наборов:

- LinkedHashSet: объединяет возможности хеш-таблицы и связанного списка. Создается с помощью функции linkedSetOf().
- HashSet: представляет хеш-таблицу. Создается с помощью функции hashSetOf().

```
val numbers1: HashSet<Int> = hashSetOf(5, 6, 7)
val numbers2: LinkedHashSet<Int> = linkedSetOf(25, 26, 27)
val numbers3: MutableSet<Int> = mutableSetOf(35, 36, 37)
```

Изменение набора с помощью MutableSet:

```
val numbers: MutableSet<Int> = mutableSetOf(35, 36, 37)
println(numbers.add(2))
println(numbers.addAll(setOf(4, 5, 6)))
println(numbers.remove(36))

for (n in numbers){ println(n) }    // 35 37 2 4 5 6
numbers.clear()
```

Map

Коллекция Map представляет коллекцию объектов, где каждый элемент имеет ключ и сопоставляемое с ним значение. При этом все ключи в коллекции являются уникальными. В отличие от List и Set интерфейс Map не расширяет интерфейс Collection.

Map представляет неизменяемую коллекцию, для создания которой применяется метод mapOf().

```
val people = mapOf(1 to "Tom", 5 to "Sam", 8 to "Bob")
```

Функция mapOf принимает набор элементов, каждый из которых с помощью оператора to сопоставляет ключ со значением, например, 1 to "Tom" (с условным идентификатором пользователя сопоставляется его имя). В данном случае переменная people представляет объект Map<Int, String>, где первый тип - Int представляет тип ключей (идентификатор пользователя), а второй тип - String представляет тип значений.

Для перебора Map можно использовать цикл for, при этом каждый элемент будет представлять объект Map.Entry<K, V>. Через его свойство key можно получить ключ, а с помощью value - значение:

```
val people = mapOf(1 to "Tom", 5 to "Sam", 8 to "Bob")
for(person in people){
    println("${person.key} - ${person.value}")
}
println(people)     // {1=Tom, 5=Sam, 8=Bob}
```

Консольный вывод:

```
1 - Tom
5 - Sam
8 - Bob
{1=Tom, 5=Sam, 8=Bob}
```

Обращение к элементам Map

Для получения элементов по ключу может применяться метод get(), в который передается ключ элемента:

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
val blue = dictionary.get("blue")
println(blue)   // синий
```

В данном случае переменная dictionary представляет объект Map<String, String>, где ключи представляют строки, а значения - то же строки (условный перевод слова). Для получения значения по ключу "blue", применяется выражение dictionary.get("blue")

Также можно сократить получение элемента с помощью квадратных скобок:

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
val blue = dictionary["blue"]
println(blue)   // синий
```

Если в Map нет элемента с указанным ключом, то возвращается null:

```
val yellow = dictionary.get("yellow")
// либо так
// val yellow = dictionary["yellow"]
println(yellow)   // null
```

Такое поведение не всегда может быть предпочтительно. И в этом случае можно использовать пару других методов для получения элементов. Так, метод getOrDefault() позволяет задать значение по умолчанию, которое будет возвращаться, если по указанному ключу нет элементов:

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
val yellow = dictionary.getOrDefault("yellow", "Undefined")
println(yellow)   // Undefined
val blue = dictionary.getOrDefault("blue", "Undefined")
println(blue)   // синий
```

Еще один метод - getOrElse() в качестве второго параметра принимает функцию, которая задает значение на случай, если по указанному ключу нет элементов:

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
val yellow = dictionary.getOrElse("yellow"){"Not found"}
println(yellow)   // Not found
val blue = dictionary.getOrElse("blue"){"Not found"}
println(blue)   // синий
```

Кроме получения отдельных элементов Map позволяет получить отдельно ключи и значения с помощью свойств :

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
println(dictionary.values)  // [красный, синий, зеленый]
println(dictionary.keys)   // [red, blue, green]
```

Также с помощью методов containsKey() и containsValue() можно проверить наличие в Map определенного ключа и значения соответственно:

```
val dictionary = mapOf("red" to "красный", "blue" to "синий", "green" to
"зеленый")
println(dictionary.containsKey("blue"))     // true
println(dictionary.containsKey("yellow"))  // false
```

```
println(dictionary.containsValue("желтый"))    // false
println(dictionary.containsValue("зеленый"))  // true
```

MutableMap

Изменяемые коллекции представлены интерфейсом MutableMap, который расширяет интерфейс Map. Для создания объекта MutableMap применяется функция mutableMapOf().

```
val people = mutableMapOf(220 to "Tom", 225 to "Sam", 228 to "Bob") //
MutableMap<Int, String>
```

Интерфейс MutableMap реализуется рядом коллекций:

- HashMap: простейшая реализация интерфейса MutableMap, не гарантирует порядок элементов в коллекции. Создается функцией hashMapOf()
- LinkedHashMap: представляет комбинацию HashMap и связанного списка, создается функцией linkedMapOf()

```
val linkedMap = linkedMapOf(220 to "Tom", 225 to "Sam", 228 to "Bob")  // объект
типа LinkedHashMap<Int, String>
val hashMap = hashMapOf(220 to "Tom", 225 to "Sam", 228 to "Bob")   // объект типа
HashMap<Int, String>
```

Для изменения Map можно применять ряд методов:

- put(key: K, value: V): добавляет элемент с ключом key и значением value
- putAll(): добавляет набор объектов типа Pair<K, V>. В качестве такого набора могут выступать объекты Iterable, Sequence и Array
- set(key: K, value: V): устанавливает для элемента с ключом key значение value
- remove(key: K): V?: удаляет элемент с ключом key. Если удаление успешно произведено, то возвращается значение удаленного элемента, иначе возвращается null.

Дополнительная версия метода

```
remove(key: K, value: V): Boolean
```

удаляет элемент с ключом key, если он имеет значение value, и возвращает true, если элемент был успешно удален

Добавление данных:

```
val people = mutableMapOf(1 to "Tom", 2 to "Sam", 3 to "Bob")

// добавляем один элемент с ключом 229 и значением Mike
people.put(229, "Mike")
println(people)     // {1=Tom, 2=Sam, 3=Bob, 229=Mike}

// добавляем другую коллекцию
val employees = mapOf(301 to "Kate", 302 to "Bill")
people.putAll(employees)
println(people)     // {1=Tom, 2=Sam, 3=Bob, 229=Mike, 301=Kate, 302=Bill}
```

Изменение данных:

```
val people = mutableMapOf(1 to "Tom", 2 to "Sam", 3 to "Bob")

// изменяем элемент с ключом 1
people.set(1, "Tomas")
println(people) // {1=Tomas, 2=Sam, 3=Bob}
```

Вместо метода set для установки значения могут применяться квадратные скобки, в которые передается ключ элемента:

```
val people = mutableMapOf(1 to "Tom", 2 to "Sam", 3 to "Bob")

// изменяем элемент с ключом 1
people[1] = "Tomas"
println(people) // {1=Tomas, 2=Sam, 3=Bob}
// изменяем элемент с ключом 5
people[5] = "Adam"
println(people) // {1=Tomas, 2=Sam, 3=Bob, 5=Adam}
```

Причем если с указанным ключом нет элемента, то он добавляется, как в примере выше в случае с ключом 5.

Удаление данных:

```
val people = mutableMapOf(1 to "Tom", 2 to "Sam", 3 to "Bob")

// удаляем элемент с ключом 1
people.remove(1)
println(people) // {2=Sam, 3=Bob}

// удаляем элемент с ключом 3, если его значение - "Alice"
people.remove(3, "Alice")
```

```
println(people) // {2=Sam, 3=Bob}

// удаляем элемент с ключом 3, если его значение - "Bob"
people.remove(3, "Bob")
println(people) // {2=Sam}
```

Последовательности

Наряду с коллекциями Kotlin предоставляет еще один тип наборов элементов – последовательности (sequences). Последовательности предоставляют похожую функциональность, что и интерфейс Iterable, который реализуется типами коллекций. Ключевая разница состоит в том, как обрабатываются элементы последовательности при применении к ним набора операций.

Создание последовательности

Функция sequenceOf

Последовательности представляют интерфейс Sequence. Для создания объекта данного типа можно использовать встроенную функцию sequenceOf(). В качестве параметра она принимает набор элементов, которые будут входить в последовательность. Например:

```
val people = sequenceOf("Tom", "Sam", "Bob")    //тип Sequence<String>
println(people.joinToString())  // Tom, Sam, Bob
```

В данном случае определяется последовательность типа Sequence, которая содержит три элемента. Для вывода последовательности на консоль применяется ее преобразование в строку с помощью функции joinToString()

Метод asSequence коллекций

Также можно создать последовательность из объекта Iterable (например, из объектов типа List или Set), используя метод asSequence()

```
val employees = listOf("Tom", "Sam", "Bob") // объект List<String>
val people = employees.asSequence()         //тип Sequence<String>
println(people.joinToString())    // Tom, Sam, Bob
```

Функция generateSequence

Третий способ создания последовательности представляет функция generateSequence

```
var number = 0
val numbers = generateSequence{ number += 2; number}
```

```
println(numbers.take(5).joinToString())    // получаем первые 5 элементов
последовательности - 2, 4, 6, 8, 10
```

В качестве параметра функция generateSequence() принимает функцию, которая возвращает некоторое значение. Это значение затем станет элементом последовательности. Так в данном случае увеличиваем значение переменной number на 2 и возвращаем ее значение. То последовательность будет содержать числа с шагом в 2: 2, 4, 6, 8, 10, 12....

Стоит учитывать, что эта функция по умолчанию генерирует бесконечную последовательность. Поэтому в примере выше при выводе элементов на консоль с помощью метода take() получаем только первые пять элементов. Чтобы ограничить генерируемую последовательность, функция в generateSequence() должна возвращать null:

```
var number = 0
val numbers = generateSequence{ number += 2; if(number > 8) null else number}
println(numbers.joinToString())    // 2, 4, 6, 8
```

В данном случае последовательность содержит только числа от 2 до 8 включительно.

Функция generateSequence() имеет несколько вариаций. В частности, в качестве первого параметра мы можем передать первое значение по умолчанию, а второй параметр по прежнему представляет функцию, которая генерирует последовательность

```
val numbers = generateSequence(5){ if(it == 25) null else it + 5}
println(numbers.joinToString())    // 5, 10, 15, 20, 25
```

В данном случае в функцию первому параметру передается число 5, то есть это будет первый элемент последовательности. Функция создания последовательности через параметр it получает предыдущий результат функции - фактически предыдущий элемент последовательности (при первом вызове функции - это значение первого параметра). В данном случае создается конечная последовательность, где каждый последующий элемент больше предыдущего на 5. Если последний элемент равен 25, то возвращаем null, тем самым завершаем генерацию последовательности. Стоит отметить, что даже если функция возвратит null, то последовательность будет содержать один элемент: generateSequence(5){ null }

Функция sequence

Четвертый способ представляет применение функции sequence(). В этой функции можно генерировать элементы последовательности с помощью функций yield() и yieldAll():

```
val numbers = sequence {
    yield(1)
    yield(4)
    yield(7)
```

```
}
println(numbers.joinToString())    // 1, 4, 7
```

Функция yield() фактически возвращает во вне некоторое значение, которое ей передается через параметр. То есть при первом обращении к функции sequence сработает вызов yield(1), который возвратит значение 1. При втором обращении сработает вызов yield(4), который возвратит 4. И при третьем обращении сработает вызов yield(7). Таким образом, последовательность будет содержать 3 элемента: 1, 4, 7.

Можно создать и бесконечную последовательность:

```
val numbers = sequence {
    var start = 0
    while(true) yield(start++)
}
println(numbers.take(5).joinToString())    // 0, 1, 2, 3, 4
```

Здесь при каждом обращении к функции sequence возвращается одно из чисел начиная с 0.

Если надо создать последовательности на основе другой последовательности или коллекции, то удобнее источник данных передать в функцию yieldAll()

```
val personal = sequence {
    val data = listOf("Alice", "Kate", "Ann")
    yieldAll(data)
}
println(personal.joinToString())    // Alice, Kate, Ann
```

Перебор последовательности

Для перебора последовательности можно применять стандартный цикл for:

```
val people = sequenceOf("Tom", "Sam", "Bob")
for(person in people) println(person)
```

Операции с последовательностями Интерфейс Sequence предоставляет ряд функций, который позволяют производить различные операции с последовательностями. Все операции с последовательностями делятся на ряд типов.

Прежде всего операции различаются по наличию состояния. Одни операции могут хранить состояние (statefull-операции), например, операция distinct(). Подобное состояние обычно пропорционально количеству элементов в последовательности. А есть операции, которые не имеют состояния (stateless-операции), например, функции map() и filter(), или требуют очень небольшого константного состояния, как функции take() и drop(). Подобные операции обрабатывают каждый элемент независимо от других.

Другая классификация операций основывается на том, когда выполняется операции. В данном случае они бывают промежуточными (intermediate) - такие операции обычно возвращают другую последовательность. Они НЕ выполняются сразу при их вызове.

А есть терминальные или конечные операции (terminal). Такие операции подразумевают извлечение элементов последовательности, например, когда нам надо получить количество элементов с помощью функции count() или элемент по индексу. Такие операции выполняются сразу при их вызове. То же самое можно сказать о переборе с помощью цикла for, который извлекает элементы из последовательности и соответственно также запускает процесс выполнения всех операций последовательности.

Рассмотрим основные операции:

- all(predicate: (T) -> Boolean): Boolean

возвращает true, если все элементы соответствуют предикату, который передается в функцию в качестве параметра

Терминальная операция

- any(): Boolean

возвращает true, если последовательность содержит хотя бы один элемент

Дополнительная версия возвращает true, если хотя бы один элемент соответствуют предикату, который передается в функцию в качестве параметра

- any(predicate: (T) -> Boolean): Boolean

Терминальная операция

- average(): Double

возвращает среднее значение для числовой последовательности типов Byte, Int, Short, Long, Float, Double

- chunked(size: Int): Sequence<List>

расщепляет последовательность на последовательность из списков List, параметр size устанавливает максимальное количество элементов в каждом из списков

Дополнительная версия в качестве второго параметра получает функцию преобразования, которая преобразует каждый список в элемент новой последовательности

- chunked(size: Int,transform: (List) -> R): Sequence

Промежуточная операция

- contains(element: T): Boolean

возвращает true, если последовательность содержит элемент element

Терминальная операция

- count(): Int

возвращает количество элементов в последовательности

Дополнительная версия возвращает количество элементов, которые соответствуют предикату

- count(predicate: (T) -> Boolean): Int

Терминальная операция

- distinct(): Sequence

возвращает новую последовательность, которая содержит только уникальные элементы

Промежуточная операция

- distinctBy(selector: (T) -> K): Sequence

возвращает новую последовательность, которая содержит только уникальные элементы с учетом функции селектора, которая передается в качестве параметра

Промежуточная операция

- drop(n: Int): Sequence

возвращает новую последовательность, которая содержит все элементы за исключением первых n элементов

Промежуточная операция

- dropWhile(predicate: (T) -> Boolean): Sequence

возвращает новую последовательность, которая содержит все элементы за исключением первых элементов, которые соответствуют предикату

Промежуточная операция

- elementAt(index: Int): T

возвращает элемент по индексу index. Если индекс выходит за пределы последовательности, то генерируется исключение типа IndexOutOfBoundsException

Терминальная операция

- elementAtOrElse(index: Int, defaultValue: (Int) -> T): T

возвращает элемент по индексу index. Если индекс выходит за пределы последовательности, то возвращается значение, устанавливаемое функцией из параметра defaultValue

Терминальная операция

- elementAtOrNull(index: Int): T?

возвращает элемент по индексу index. Если индекс выходит за пределы последовательности, то возвращается null

Терминальная операция

- filter(predicate: (T) -> Boolean): Sequence

возвращает новую последовательность из элементов, которые соответствуют предикату

Промежуточная операция

- filterNot(predicate: (T) -> Boolean): Sequence

возвращает новую последовательность из элементов, которые НЕ соответствуют предикату

Промежуточная операция

- filterNotNull(): Sequence

возвращает новую последовательность из элементов, которые не равны null

Промежуточная операция

- find(predicate: (T) -> Boolean): T?

возвращает первый элемент, который соответствует предикату. Если элемент не найден, то возвращается null

Терминальная операция

- findLast(predicate: (T) -> Boolean): T?

возвращает последний элемент, который соответствует предикату. Если элемент не найден, то возвращается null

Терминальная операция

- first(): T

возвращает первый элемент последовательности

Дополнительная версия возвращает первый элемент, которые соответствует предикату

- first(predicate: (T) -> Boolean): T

Если элемент не найден, то генерируется исключение типа NoSuchElementException

Терминальная операция

- firstOrNull(): T?

возвращает первый элемент последовательности

Дополнительная версия возвращает первый элемент, которые соответствует предикату

- firstOrNull(predicate: (T) -> Boolean): T?

Если элемент не найден, то возвращается null

Терминальная операция

- flatMap(transform: (T) -> Sequence): Sequence

преобразует последовательность элементов типа T в последовательность элементов типа R, используя функцию преобразования, которая передается в качестве параметра

Промежуточная операция

- fold(initial: R, operation: (acc: R, T) -> R): R

Возвращает значение, которое является результатом действия функции operation над каждым элементом последовательности. Первый параметр функции operation - результат работы функции над предыдущим элементом последовательности (при первом вызове - значение из параметра initial), в второй параметр - текущий элемент последовательности.

Терминальная операция

- forEach(action: (T) -> Unit)

Выполняет для каждого элемента последовательности действие action.

Терминальная операция

- groupBy(keySelector: (T) -> K): Map<K, List>

Группирует элементы по ключу, который возвращается функцией keySelector. Результат функции карта Map, где ключ - собственно ключ элементов, а значение - список List из элементов, которые соответствуют этому ключу

Дополнительная версия принимает функцию преобразования элементов:

- groupBy(keySelector: (T) -> K, valueTransform: (T) -> V): Map<K, List>

Терминальная операция

- indexOf(element: T): Int

Возвращает индекс первого вхождения элемента element. Если элемент не найден, возвращается -1

Терминальная операция

- indexOfFirst(predicate: (T) -> Boolean): Int

Возвращает индекс первого элемента, который соответствует предикату. Если элемент не найден, возвращается -1

Терминальная операция

- indexOfLast(predicate: (T) -> Boolean): Int

Возвращает индекс последнего элемента, который соответствует предикату. Если элемент не найден, возвращается -1

Терминальная операция

- joinToString(): String

Генерирует из последовательности строку

Терминальная операция

- last(): T

возвращает последний элемент последовательности

Дополнительная версия возвращает последний элемент, которые соответствует предикату

- last(predicate: (T) -> Boolean): T

Если элемент не найден, то генерируется исключение типа NoSuchElementException

Терминальная операция

- lastOrNull(): T?

возвращает последний элемент последовательности

Дополнительная версия возвращает последний элемент, которые соответствует предикату

- lastOrNull(predicate: (T) -> Boolean): T?

Если элемент не найден, то возвращается null

Терминальная операция

- lastIndexOf(element: T): Int

Возвращает последний индекс элемента element. Если элемент не найден, возвращается -1

Терминальная операция

- map(transform: (T) -> R): Sequence

Применяет к элементам последовательности функцию трансформации и возвращает новую последовательность из новых элементов

Промежуточная операция

- mapIndexed(transform: (index: Int, T) -> R): Sequence

Применяет к элементам последовательности и их индексам функцию трансформации и возвращает новую последовательность из новых элементов

Промежуточная операция

- mapNotNull(transform: (T) -> R?): Sequence

Применяет к элементам последовательности функцию трансформации и возвращает новую последовательность из новых элементов, которые не равны null

Промежуточная операция

- maxOf(selector: (T) -> Double): Double

Возвращает максимальное значение на основе селектора

Терминальная операция

- maxOfOrNull(selector: (T) -> Double): Double?

Возвращает максимальное значение на основе селектора. Если последовательность пуста, возвращается null

Терминальная операция

- maxOrNull(): Double?

Возвращает максимальное значение. Если последовательность пуста, возвращается null

Терминальная операция

- minOf(selector: (T) -> Double): Double

Возвращает минимальное значение на основе селектора

Терминальная операция

- minOfOrNull(selector: (T) -> Double): Double?

Возвращает минимальное значение на основе селектора. Если последовательность пуста, возвращается null

Терминальная операция

- minOrNull(): Double?

Возвращает минимальное значение. Если последовательность пуста, возвращается null

Терминальная операция

- minus(element: T): Sequence

Возвращает новую последовательность, которая содержит все элементы текущей за исключением элемента element.

Имеет разновидности, которую позволяют исключить из последовательности наборы элементов:

```
minus(elements: Array<T>): Sequence<T>
minus(elements: Iterable<T>): Sequence<T>
minus(elements: Sequence<T>): Sequence<T>
```

Промежуточная операция

- plus(element: T): Sequence

Возвращает новую последовательность, которая содержит все элементы текущей за исключением плюс элемент element.

Имеет разновидности, которую позволяют включить в последовательность наборы элементов:

```
plus(elements: Array<T>): Sequence<T>
plus(elements: Iterable<T>): Sequence<T>
plus(elements: Sequence<T>): Sequence<T>
```

Промежуточная операция

- reduce(operation: (acc: S, T) -> S): S

Возвращает значение, которое является результатом действия функции operation над каждым элементом последовательности. Первый параметр функции operation - результат работы функции над предыдущим элементом последовательности, в второй параметр - текущий элемент последовательности.

Терминальная операция

- shuffled(): Sequence

Условно перемешивает последовательность

Промежуточная операция

- sorted(): Sequence

Сортирует последовательность по возрастанию

Промежуточная операция

- sortedBy(selector: (T) -> R?): Sequence

Сортирует последовательность по возрастанию на основе селектора

Промежуточная операция

- sortedByDescending(selector: (T) -> R?): Sequence

Сортирует последовательность по убыванию на основе функции-селектора

Промежуточная операция

- sortedDescending(): Sequence

Сортирует последовательность по убыванию

Промежуточная операция

- sum(): Int

Возвращает сумму элементов последовательности.

Терминальная операция

- sumOf(selector: (T) -> Int): Int

Возвращает сумму элементов последовательности на основе функции-селектора

Терминальная операция

- take(n: Int): Sequence

Возвращает новую последовательность, которая содержит n первых элементов текущей последовательности

Промежуточная операция

- takeWhile(predicate: (T) -> Boolean): Sequence

Возвращает новую последовательность, которая содержит n первых элементов текущей последовательности, соответствующих функции-предикату

Промежуточная операция

- toHashSet(): HashSet

Создает из последовательности объект HashSet

Терминальная операция

- toList(): List

Создает из последовательности объект List

Терминальная операция

- toMap(): Map<K, V>

Создает из последовательности объект Map

Терминальная операция

- toSet(): Set

Создает из последовательности объект Set

Терминальная операция

Отличие последовательности от коллекций Iterable

Отличие последовательности от коллекций Iterable И последовательности, и коллекции, которые реализуют интерфейс Iterable, по сути представляют набор элементов. Более того предоставляют похожий набор операций для обработки элементов. Но отличие состоит, как эти операции обрабатывают элементы при применении сразу нескольких операций.

Так, при применении набора операций к коллекции Iterable каждая отдельная операция возвращает промежуточный результат - промежуточную коллекцию. А при обработке последовательности весь набор операций выполняется только тогда, когда требуется конечный результат обработки.

Также меняется порядок применения операций. Коллекция применяет каждую операцию последовательно к каждому элементу. То есть сначала выполняет первую операцию для всех элементов, потом вторую операцию для элементов коллекции, полученных после первой операции. И так далее.

Последовательность применяет весь набор операций отдельно к каждому элементу. То есть сначала весь набор операций применяется к первому элементу, потом ко второму элементу и так далее. Таким образом, последовательность позволяет избежать создания помежуточных коллекций и в тоже время повышают производительность при выполнении набора операций особенно для большого набора данных. Однако при небольших наборах данных и малом количестве операций может быть эффективнее использовать коллекции Iterable.

Рассмотрим на примере. Сначала возьмем коллекции Iterable (в данном случае List):

```
fun main(){

    var people = listOf(
        Person("Tom", 37),
        Person("Sam", 25),
        Person("Alice", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
                   .filter{ println("Name filter: ${it}"); it.name.length == 3 }
    println("Result:")
    for(person in people) println(person)
}
data class Person(val name: String, val age: Int)
```

Здесь создается коллекция - список people (объект типа List), который содержит объекты Person. Далее к этому списку применяются две операции фидльтрации в виде метода filter(). Сначала получаем все объекты Person, у которых свойство age больше 30:

```
people.filter { println("Age filter: ${it}"); it.age > 30 }
```

Эта операция filter() возвратит промежуточную коллекцию, которая содержит все объекты Person с возрастом больше 30. Для наглядности здесь логгируются на консоль объекты, к которым применяется операция.

Затем выполняется вторая операция filter() - она возвращает из промежуточной коллекции те объекты Person, у которых длина свойства name равна 3.

```
filter{ println("Name filter: ${it}"); it.name.length == 3 }
```

Опять же для наглядности здесь логгируются на консоль объекты, к которым применяется операция.

Результатом будет вторая коллекция, которая будет присвоена переменной people и которую в конце с помощью цикла foreach выводится на консоль. В итоге мы получим следующий консольный вывод:

```
Age filter: Person(name=Tom, age=37)
Age filter: Person(name=Sam, age=25)
Age filter: Person(name=Alice, age=33)
Name filter: Person(name=Tom, age=37)
Name filter: Person(name=Alice, age=33)
Result:
Person(name=Tom, age=37)
```

Здесь мы видим, что первая операция filter пробегается по всем элементам в начальной коллекции и возвращает коллекцию с двумя элементами. Вторая операция filter пробегается по полученной коллекции из двух элементов и возвращает коллекцию из одного элемента.

Теперь вместо коллекций применим последовательности:

```
fun main(){

    var people = sequenceOf(
        Person("Tom", 37),
        Person("Sam", 25),
        Person("Alice", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
                   .filter{ println("Name filter: ${it}"); it.name.length == 3 }
    println("Result:")
    for(person in people) println(person)
}
data class Person(val name: String, val age: Int)
```

Здесь абсолютно такой же код, как и в предыдущем примере, только переменная people теперь представляет последовательность объектов Person. Однако консольный вывод будет совершенно иным:

```
Result:
Age filter: Person(name=Tom, age=37)
Name filter: Person(name=Tom, age=37)
Person(name=Tom, age=37)
Age filter: Person(name=Sam, age=25)
Age filter: Person(name=Alice, age=33)
Name filter: Person(name=Alice, age=33)
```

Во-первых, обратите внимание, что строка "Result:" выводится до выполнения всех операций. Потому что получение результата фактически происходит в цикле for при обращении к последовательности. До этого нет смысла выполнять операции, если элементы последовательности никак не используются.

Во-вторых, также обратите внимание на применение операций к элементам.

- Сначала обрабатывается первый элемент - Person(name=Tom, age=37). Поскольку он соответствует обоим фильтрам, то он в конечном счете выводится на консоль в цикле for.
- Далее обрабатывается второй элемент - Person(name=Sam, age=25), однако после применения первой операции filter его обработка завершается, поскольку он не соответствует условию первого фильтра
- В конце обрабатывается третий элемент - Person(name=Alice, age=33), к нему применяются две операции filter, но затем его обработка завершается, поскольку он не соответствует условию второго фильтра

Получение элементов последовательности происходит, когда идет непосредственное обращение к элементам последовательности. Например, мы можем изменить предыдущий пример следующим образом:

```
fun main(){

    var people = sequenceOf(
        Person("Tom", 37),
        Person("Sam", 25),
        Person("Alice", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
    println("Between Age filter and Name filter")
    people = people.filter{ println("Name filter: ${it}"); it.name.length == 3 }
    for(person in people) println(person)
}
data class Person(val name: String, val age: Int)
```

Здесь результат каждого фильтра присваивается переменной people. А между фильтрами идет вывод сообщения на консоль. Но все равно, поскольку непосредственное получение элементов последовательности происходит в цикле for, то именно в этой точке кода будут выполняться все операции с последовательностью, что можно увидеть из консольного вывода:

```
Between Age filter and Name filter
Age filter: Person(name=Tom, age=37)
Name filter: Person(name=Tom, age=37)
Person(name=Tom, age=37)
Age filter: Person(name=Sam, age=25)
Age filter: Person(name=Alice, age=33)
Name filter: Person(name=Alice, age=33)
```

Сокращение набора операций

Применение последовательностей может значительно сократить количество применяемых операций. Например:

```
fun main(){

    var people = listOf(
        Person("Tom", 37),
        Person("Sam", 25),
        Person("Alice", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
                   .take(1)
    for(person in people) println(person)

}
data class Person(val name: String, val age: Int)
```

Здесь опять же к списку people применяется фильтр по возрасту и затем с помощью вызова take(1) выбираем один объект в результирующую коллекцию. И в этом случае мы получим следующий консольный вывод:

```
Age filter: Person(name=Tom, age=37)
Age filter: Person(name=Sam, age=25)
Age filter: Person(name=Alice, age=33)
Person(name=Tom, age=37)
```

Здесь опять же мы видим, что вызов filter() применяется ко всем элементам, из которых формируется промежуточная коллекция, из которой в итоге выбирается 1 объект.

Изменим тип набора на последовательность:

```
fun main(){

    var people = sequenceOf(
        Person("Tom", 37),
        Person("Sam", 25),
        Person("Alice", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
                   .take(1)
    for(person in people) println(person)

}
data class Person(val name: String, val age: Int)
```

Консольный вывод программы:

```
Age filter: Person(name=Tom, age=37)
Person(name=Tom, age=37)
```

Сначала обрабатывается первый объект - Person(name=Tom, age=37) - к нему применяется вызов filter(). Поскольку этот объект соответствует фильтру, он переходит к применению вызова take(1). Этот вызов выбирает в результирующую коллекцию первый объект. Но поскольку результирующая коллекция должна содержать только 1 объект, то остальные элементы последовательности нет смысла обрабатывать. И на этом обработка последовательности закончилась. Таким образом, вместо 3 операций filter в данном случае мы получаем только 1. Соответственно на большем количестве данных и операций сокращение набора операций может быть более значительным.

Фильтрация

Фильтрация по условию Фильтрация является одной из распространенных операций. Для фильтрации по условию применяется функция filter(), которая в качестве параметра принимает условие-предикат в виде функции (T) -> Boolean.

```
filter(predicate: (T) -> Boolean): List<T>/Map<K, V>/Sequence<T>
```

Функция предиката принимает в качестве параметра элемент набора. Если элемент соответствует условию, то возвращается true, а данный элемент помещается в возвращаемый набор.

Для коллекций List и Set эта функция возвращает объект List, для Map - объект Map, для последовательностей Sequence - также объект Sequence:

```
fun main(){

    var people = sequenceOf("Tom", "Sam", "Mike", "Bob", "Alice")
    people = people.filter{it.length == 3}    // получаем значения с длиной в 3
символа
    println(people.joinToString())  // Tom, Sam, Bob

    var employees = listOf(
        Person("Tom", 37),
        Person("Bob", 41),
        Person("Sam", 25)
    )
    employees = employees.filter{it.age > 30} // получаем всех Person, у которых
age > 30
    println(employees.joinToString())  // Person(name=Tom, age=37),
Person(name=Bob, age=41)
}
data class Person(val name: String, val age: Int)
```

Если надо получить элементы, которые, наоборот, НЕ соответствует условию, то можно применить функцию filterNot(), которая работает аналогично:

```
var people = sequenceOf("Tom", "Sam", "Mike", "Bob", "Alice")
people = people.filterNot{it.length == 3}    // получаем значения с длиной, не равной 3 символам
println(people.joinToString())  // Mike, Alice
```

Фильтрация по индексу

Еще одна функция - filterIndexed() также получает индекс текущего элемента:

```
filterIndexed(predicate: (index: Int, T) -> Boolean): List<T>
```

Например, получим из коллекции строк элементы с четными индексами и длиной в 3 символа:

```
val people = listOf("Tom", "Mike", "Sam", "Bob", "Alice")
// получаем значения с длиной в 3 символа на четных индексах
val filtered = people.filterIndexed{ index, s -> (index % 2 == 0) && (s.length == 3)}
println(filtered)  // [Tom, Sam]
```

Фильтрация по типу

Если коллекция/последовательность содержит элементы разных типов, то с помощью функции filterIsInstance() можно извлечь элементы определенного типа. Например:

```
fun main(){

    val people = listOf(
        Person("Tom"), Employee("Bob"),
        Person("Sam"), Employee("Mike")
    )
    // получаем только элементы типа Employee
    val employees = people.filterIsInstance<Employee>();
    println(employees)  // [Bob, Mike]
}
open class Person(val name: String){
    override fun toString(): String  = name
}
class Employee(name: String): Person(name)
```

В данном случае получаем из коллекции people только те объекты, которые представляют тип Employee. Чтобы указать тип получаемых объектов, при вызове функция типизируется соответствующим типом.

Фильтрация по null

Функция filterNotNull() позволяет выфильтровать все значение, которые равны null:

```
filterNotNull(): List<T>/Sequence<T>
```

Например:

```
fun main(){

    val people = listOf(Person("Tom"), null, Person("Sam"), null)
    println(people)     // [Tom, null, Sam, null]
    val filtered = people.filterNotNull()
    println(filtered)  // [Tom, Sam]
}
open class Person(val name: String){
    override fun toString(): String  = name
}
```

Проверка элементов

Проверка соответствия элементов условию Отдельный вид операций позволяет проверить наличие элементов.

Функция all проверяет, все ли элементы коллекции/последовательности соответствуют условию предиката:

```
all(predicate: (T) -> Boolean): Boolean
```

Функция предиката получает каждый элемент и возвращает true, если элемент соответствует условию.

Функция any проверяет, соответствует хотя бы один элемент коллекции/последовательности условию предиката:

```
any(predicate: (T) -> Boolean): Boolean
```

Еще одна функция - none возвращает true, если ни один из элементов НЕ соответствует условию предиката:

```
none(predicate: (T) -> Boolean): Boolean
```

Проверка элементов на соответствие условию:

```
val people = listOf("Tom", "Kate", "Sam", "Alice", "Bob")
// all
println(people.all{it.length == 3})     // false
println(people.all{it.length != 10})     // true

// none
println(people.none{it.length == 3})     // false
println(people.none{it.length == 2})     // true

// any
println(people.any{it.length == 3})     // true
println(people.any{it.length == 10})    // false
```

Также функции any() и none() имеют версии без параметров. В этом случае функция any() возвращает true, если коллекция/последовательность содержит хотя бы один элемент, а функция none() возвращает true, если коллекция/последовательность пуста:

```
val people = listOf("Tom", "Kate", "Sam", "Alice", "Bob")
val empty: List<String> = listOf()
// any
println(people.any())     // true
println(empty.any())     // false

// none
println(people.none())     // false
println(empty.none())     // true
```

Функция contains() возвращает true, если в коллекции/последовательности есть определенный элемент:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
println(people.contains("Sam"))     // true
println(people.contains("Bill"))     // false
```

Еще одна функция - containsAll() возвращает true, если коллекция содержит все элементы другой коллекции:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
println(people.containsAll(listOf("Tom", "Sam")) ) // true
println(people.containsAll(listOf("Tom", "Bill")) ) // false
```

Трансформации

Трансформации

Для трансформации одной коллекции/последовательности применяется функция map()

```
map(transform: (T) -> R): List<R>/Sequence<R>
```

В качестве параметра она принимает функцию пребразования. Функция пребразования получает текущий элемент коллекции/последовательности и возвращает результат преобразования. Причем типа входных и выходных данных могут совпадать, а могут и отличаться. Например:

```
fun main(){

    val people = listOf(Person("Tom"), Person("Sam"), Person("Bob"))

    val names = people.map { it.name } // возвращаем имя каждого пользователя
    println(names)  // [Tom, Sam, Bob]
}
class Person(val name: String)
```

Здесь из списка List получаем список List, который содержит имена пользователей.

Другой пример - трансформируем последовательность чисел в последовательность квадратов этих чисел:

```
val numbers = listOf(1, 2, 3, 4, 5)
val squares = numbers.map { it * it }
println(squares)    // [1, 4, 9, 16, 25]
```

Еще одна функция - mapIndexed() также передает в функцию преобразования индекс текущего элемента:

```
mapIndexed(transform: (index: Int, T) -> R): List<R>
```

Применение функции:

```
fun main(){

    val people = listOf(Person("Tom"), Person("Sam"), Person("Bob"))

    val names = people.mapIndexed{ index, p-> "${index+1}.${p.name}"}
    println(names)  // [1.Tom, 2.Sam, 3.Bob]
}
class Person(val name: String)
```

Если необходимо отсеять значения null, которые могут возникать при преобразовании, то можно применять аналоги выше упомянутых функций - mapNotNull() и mapIndexedNotNull():

```
fun main(){

    val people = listOf(
        Person("Tom"), Person("Sam"),
        Person("Bob"), Person("Alice")
    )
    // элементы длиной имени не равной 3 преобразуем в null
    val names1 = people.mapNotNull{ if(it.name.length!=3) null else it.name }

    // элементы на четных позициях преобразуем в null
    val names2 = people.mapIndexedNotNull{ index, p -> if(index%2==0) null else
p.name }

    println(names1)  // [Tom, Sam, Bob]
    println(names2)  // [Sam, Alice]
}
class Person(val name: String)
```

flatten

Функция flatten() позволяет преобразовать коллекцию/последовательность, которая содержит вложенные коллекции/последовательности:

```
flatten(): List<T>/Sequence<T>
```

Эта функция помещает элементы всех вложенных коллекций в одну:

```
val personal = listOf(listOf("Tom", "Bob"), listOf("Sam", "Mike", "Kate"),
listOf("Tom", "Bill"))
val people = personal.flatten()
println(people)    // [Tom, Bob, Sam, Mike, Kate, Tom, Bill]
```

Группировка

Для группировки элементов коллекции/последовательности применяется функция groupBy():

```
groupBy(keySelector: (T) -> K): Map<K, List<T>>
groupBy(keySelector: (T) -> K,valueTransform: (T) -> V): Map<K, List<V>>
```

Обе версии в качестве параметра принимают функцию, которая определяет критерий группировки. Вторая версия в дополнение принимает функцию преобразования. Результатом функции является объект Map, который хранит набор групп. Ключами являются критерии группировки - ключи групп, а значениями - списки List, которые соответствуют этим критериям группировки и представляют группы.

Например, сгруппирует сотрудников по их компаниям:

```
fun main(){

    val employees = listOf(
        Employee("Tom", "Microsoft"),
        Employee("Bob", "JetBrains"),
        Employee("Sam", "Google"),
        Employee("Alice", "Microsoft"),
        Employee("Kate", "Google")
    )
    val companies = employees.groupBy { it.company }    // объект Map<String,
List<Employee>>

    println(companies) // {Microsoft=[Tom, Alice], JetBrains=[Bob], Google=[Sam,
Kate]}

    // перебор групп
    for (company in companies){
        println(company.key) // название компании
        // перебор списка сотрудников
        for (employee in company.value){
            println(employee.name)
        }
        println() // для отделения групп
    }
}
class Employee(val name: String, val company: String){
    override fun toString(): String = name
}
```

Здесь критерием группировки является свойство company объекта Employee. Соответственно ключом группы в результирующем объекте Map будет значение этого свойства, а значением - список Employee. Затем можно получить все значения из каждой группы стандартным перебором объекта Map. Консольный вывод программы:

```
{Microsoft=[Tom, Alice], JetBrains=[Bob], Google=[Sam, Kate]}
Microsoft
Tom
Alice

JetBrains
Bob

Google
Sam
Kate
```

Теперь применим другую версию функции groupBy с использованием трансформации:

```
fun main(){

    val employees = listOf(
        Employee("Tom", "Microsoft"),
        Employee("Bob", "JetBrains"),
        Employee("Sam", "Google"),
        Employee("Alice", "Microsoft"),
        Employee("Kate", "Google")
    )
    val companies = employees.groupBy({it.company}) { it.name }  // объект
Map<String, List<String>>

    println(companies) // {Microsoft=[Tom, Alice], JetBrains=[Bob], Google=[Sam,
Kate]}
    // перебор групп
    for (company in companies){
        println(company.key) // название компании
        // перебор списка сотрудников
        for (employee in company.value){
            println(employee)
        }
        println() // для отделения групп
    }
}
class Employee(val name: String, val company: String){
    override fun toString(): String = name
}
```

В данном случае в качестве критерия группировки опять выступает свойство company объектов Employee ({ it.company }), однако сама группа будет представлять список строк - объект List, поскольку функция преобразования вытаскивает значение свойства name объекта Employee: { it.name }

Сортировка

Для сортировки коллекции/последовательности применяются функции sorted() (сортировка по возрастанию) и sortedDescending() (сортировка по убыванию).

Что в данном случае значит сортировка по возрастанию или убыванию? По умолчанию процесс сортировки опирается на реализацию интерфейса Comparable, которая определяет, какой объект будет больше, а какой меньше. Так, для встроенных базовых типов действует следующая логика:

Числа сравниваются как в математике исходя из их значения

Символы (Char) и строки (String) сравниваются исходя из лексикографического порядка, то есть "Tom" больше, чем "Alice", потому что первый символ - T располагается в алфавите после символа A.

Сортировка чисел и строк:

```
val people = listOf("Tom", "Mike", "Bob", "Sam", "Alice")
val numbers = listOf(3, 5, 2, -4, -6, 9, 1)

// сортировка по возрастанию
val sortedPeople = people.sorted()
val sortedNumbers = numbers.sorted()
println(sortedPeople)   // [Alice, Bob, Mike, Sam, Tom]
println(sortedNumbers)  // [-6, -4, 1, 2, 3, 5, 9]

// сортировка по убыванию
println(people.sortedDescending())  // [Tom, Sam, Mike, Bob, Alice]
println(numbers.sortedDescending()) // [9, 5, 3, 2, 1, -4, -6]
```

Реализация интерфейса Comparable

Если мы определяем свои типы и хотим, чтобы их объекты также можно было отсортировать, то в этом случае следует реализовать интерфейс Comparable:

```
ublic interface Comparable<in T> {
    public operator fun compareTo(other: T): Int
}
```

Его функция compareTo() должна определять логику сравнения. В качестве параметра она принимает объект, который сравнивается с текущим.

В качестве результата возвращается число. Если текущий объект равен объекту other, то функция должна возвратить 0. Если текущий объект больше объекта other, то возвращается положительное число, если меньше - то отрицательное.

Посмотрим на примере реализацию интерфейса:

```
fun main(){
```

```
    val people = listOf(
        Person("Tom", 37),
        Person("Bob",41),
        Person("Sam", 25)
    )
    // сортировка по возрастанию
    val sortedPeople = people.sorted()

    // сортировка по  возрастанию
    println(sortedPeople)   // [Bob (41), Sam (25), Tom (37)]

    // сортировка по убыванию
    println(people.sortedDescending())  // [Tom (37), Sam (25), Bob (41)]
}
class Person(val name: String, val age: Int): Comparable<Person> {
    override fun compareTo(other: Person): Int = name.compareTo(other.name)
    override fun toString(): String = "$name ($age)"
}
```

В данном случае класс Person реализует интерфейс Comparable, однако в самом методе compareTo фактически мы полагаемся на реализацию этого интерфейса для строк. То есть мы фактически возвращаем результат сравнения двух строк - имен пользователей:

```
name.compareTo(other.name)
```

В итоге объекты Person будут сортироваться в соответствии с расположением первых букв их имен в лексикографического порядке - стандартная сортировка для строк.

Теперь изменим принцип сортировки и сравним два объекта по их возрасту - значению свойства age:

```
fun main(){

    val people = listOf(
        Person("Tom", 37),
        Person("Bob",41),
        Person("Sam", 25)
    )
    // сортировка по возрастанию
    val sortedPeople = people.sorted()

    // сортировка по  возрастанию
    println(sortedPeople)   // [Sam (25), Tom (37), Bob (41)]

    // сортировка по убыванию
    println(people.sortedDescending())  // [Bob (41), Tom (37), Sam (25)]
}
class Person(val name: String, val age: Int): Comparable<Person> {
    override fun compareTo(other: Person): Int = age - other.age
```

```
    override fun toString(): String = "$name ($age)"
}
```

Тепрь объект Person считается "больше", если у него больше значение свойства age.

sortedWith и интерфейс Comparator

Интерфейс Comparable позволяет легко определить логику сортировки, однако бывает, что нам недоступен код классов, объекты которых мы хотим отсортировать. Либо мы хотим задать для уже существующих типов, которые уже реализуют Comparable, другой способ сортировки. В этом случае мы можем использовать интерфейс Comparator (грубо говоря компаратор):

```
public expect fun interface Comparator<T> {
    public fun compare(a: T, b: T): Int
}
```

Его функция compare() принимает два параметра - два сравниваемых объекта и также возвращает целое число. Если первый параметр больше второго, то возвращается положительное число, если меньше - то отрицательное. Если объекты равно, возвращается 0.

Kotlin имеет встроенную функцию sortedWith(), которая в качестве параметра принимает компаратор и на его основе сортирует коллекцию/последовательность:

```
sortedWith(comparator: Comparator<in T>): List<T>
```

Пример реализации:

```
fun main(){

    val people = listOf(
        Person("Tom", 37),
        Person("Bob",41),
        Person("Sam", 25)
    )
    val personComparator = Comparator{ p1: Person, p2: Person -> p1.age - p2.age }

    val sortedPeople = people.sortedWith(personComparator)
    println(sortedPeople)   // [Sam (25), Tom (37), Bob (41)]
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

В данном случае компаратор определен в виде переменной personComparator. В реализации интерфейса как и в предыдущем примере сравниваем пользователей на основе возраста: если

значение свойства age больше, то и объект Person условно считается "больше".

В принципе в данном случае мы можем сократить вызов функции сортировки:

```
val sortedPeople = people.sortedWith{ p1: Person, p2: Person -> p1.age - p2.age }
```

или так:

```
val sortedPeople = people.sortedWith(compareBy{ it.age })
```

Благодаря компаратору можно задать свою логику сортировки к уже имеющимся типам. Например, отсортируем строки по длине:

```
fun main(){

    val people = listOf("Tom", "Alice", "Kate", "Sam", "Bob")
    // отсортируем по длине строки
    val sorted  = people.sortedWith(compareBy{ it.length })
    println(sorted)   // [Tom, Sam, Bob, Alice]
}
```

Сортировка на основе критерия

Еще две функцию позволяют отсортировать объекты по определенному критерию:

```
sortedBy(crossinline selector: (T) -> R?): List<T> / Sequence<T>
sortedByDescending(crossinline selector: (T) -> R?): List<T> / Sequence<T>
```

Функция sortedBy() сортирует по возрастанию, а sortedByDescending() - по убыванию. В качестве параметра они принимают функцию, которая получает элемент и возвращает значение, применяемое для сортировки.

```
fun main(){

    val people = listOf( Person("Tom", 37), Person("Bob",41), Person("Sam", 25))

    // сортировка по свойству name по возрастанию
    val sortedByName = people.sortedBy { it.name }
    println(sortedByName)   // [Bob (41), Sam (25), Tom (37)]

    // сортировка по свойству age по возрастанию
    val sortedByAge = people.sortedBy { it.age }
    println(sortedByAge)   // [Sam (25), Tom (37), Bob (41)]
```

```
    // сортировка по свойству age по убыванию
    val sortedByAgeDesc = people.sortedByDescending { it.age }
    println(sortedByAgeDesc)   // [Bob (41), Tom (37), Sam (25)]
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

reverse и shaffle

Стоит отметить еще две функции, которые не сортируют, а просто меняют порядок элементов. Функция reversed() изменяет порядок элементов на обратный, а функция shuffle() перемешивает элементы случайным образом:

```
val numbers = listOf(1, 2, 3, 4, 5, 6)
val reversed = numbers.reversed()
println(reversed)   // [6, 5, 4, 3, 2, 1]

val shuffled = numbers.shuffled()
println(shuffled)   // [4, 6, 3, 5, 1, 2]
```

Агрегатные операции

Коллекции/последовательности поддерживают ряд агрегатных операций, которые возвращают некоторое значение.

Минимальное и максимальное значение Функции minOrNull() и maxOrNull() возвращают соответственно минимальное и максимальное значение (если коллекция/последовательность пута, то возвращается null). Причем для работы этих функций элементы коллекции/последовательности должны реализовать интерфейс Comparable:

```
val numbers = listOf(4, 6, 3, 5, 1, 2)
val people = listOf("Alice", "Tom", "Sam", "Kate", "Bob")

println(numbers.minOrNull())   // 1
println(numbers.maxOrNull())   // 6

println(people.minOrNull())   // Alice
println(people.maxOrNull())   // Tom
```

Функции minByOrNull() и maxByOrNull() в качестве параметра принимают функцию селектора, которая позволяет определить критерий сравнения объектов:

```
fun main(){

    val people = listOf( Person("Tom", 37), Person("Bob",41), Person("Sam", 25))
    // минимальный возраст
    val personWithMinAge = people.minByOrNull { it.age }
    println(personWithMinAge)     // Sam (25)
    // максимальный возраст
    val personWithMaxAge = people.maxByOrNull { it.age }
    println(personWithMaxAge)     // Bob (41)
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

Выше функции находили элемент с наименьшим и наибольшим значением свойства age. Но что, если мы хотим получить не весь объект, а сами значения минимального и максимального возраста? В этом случае мы можем использовать функции minOfOrNull() и maxOfOrNull(), которые также принимают функцию селектора, но возвращает само значение, на основе которого происходит сравнение:

```
fun main(){

    val people = listOf( Person("Tom", 37), Person("Bob",41), Person("Sam", 25))
    // минимальный возраст
    val minAge = people.minOfOrNull { it.age }
    println(minAge)     // 25
    // максимальный возраст
    val maxAge = people.maxOfOrNull { it.age }
    println(maxAge)     // 41
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

Еще пара функций - minWithOrNull() и maxWithOrNull() в качестве параметра принимают компаратор - реализацию интерфейса Comparator:

```
val colors = listOf("red", "green", "blue", "yellow")
val minColor = colors.minWithOrNull(compareBy {it.length})
val maxColor = colors.maxWithOrNull(compareBy {it.length})
println(minColor)   // red
println(maxColor)   // yellow
```

В качестве критерия сравнения здесь применяется свойство length строк, то есть строки сравниваются по длине.

И еще одна пара функций - minOfWithOrNull() и maxOfWithOrNull() принимают реализацию интерфейса Comparator (первый параметр) и селектор критерия для сравнения (второй параметр):

```
maxOfWithOrNull(comparator: Comparator<in R>, selector: (T) -> R): R?
minOfWithOrNull(comparator: Comparator<in R>, selector: (T) -> R): R?
```

Применим данные функции:

```
fun main(){

    val people = listOf(
        Person("Tom", 37), Person("Kate",29),
        Person("Sam", 25), Person("Alice", 33)
    )
    // самое короткое имя
    val minName = people.minOfWithOrNull(compareBy{it.length}) { it.name }
    println(minName)     // Tom
    // самое длинное имя
    val maxName = people.maxOfWithOrNull(compareBy{it.length}) { it.name }
    println(maxName)     // Alice
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

В качестве критерия сравнения здесь применяется свойство name объектов Person, так как функция селектора определена следующим образом: { it.name } (здесь it - это текущий объект Person)

Поскольку в качестве критерия сравнения применяется свойство name, то в функцию компаратора передается значение этого свойства. И собственно оно используется для сравнения объектов Person: compareBy{it.length} (здесь it - это значение свойства name)

Среднее значение

Для получения среднего значения применяется функция average():

```
val numbers = listOf(4, 6, 3, 5, 1, 2)
val avg = numbers.average()
println(avg)    // 3.5
```

Сумма значений

Для получения суммы числовых значений применяется функция sum():

```
val numbers = listOf(4, 6, 3, 5, 1, 2)
val sum = numbers.sum()
println(sum)    // 21
```

Количество элементов

Для получения количества элементов в коллекции/последовательности применяется функция count(). Она имеет два варианта:

```
count(): Int
count(predicate: (T) -> Boolean): Int
```

Первая версия возвращает количество всех элементов. Вторая версия возвращает количество элементов, которые соответствуют условию предиката, передаваемого в функцию в качестве параметра. Пример применения функций:

```
val people = listOf("Tom", "Sam", "Bob", "Kate", "Alice")
// совокупное количество
val count1 = people.count()
println(count1)      // 5

// количество строк, у которых длина равна 3
val count2 = people.count{it.length == 3}
println(count2)      // 3
```

Сведение элементов

Функция reduce() сводит все значения потока к одному значению:

```
reduce(operation: (acc: S, T) -> S): S
```

reduce принимает функцию, которая имеет два параметра. Первый параметр при первом запуске представляет первый элемент, а при последующих запусках - результат функции над предыдущими элементами. А второй параметр функции - следующий элемент.

Например, у нас есть список чисел, найдем сумму всех чисел:

```
val numbers = listOf(1, 2, 3, 4, 5)
val reducedValue = numbers.reduce{ a, b -> a + b }
println(reducedValue)   // 15
```

Здесь при первом запуске в функции в reduce параметр a равен 1, а параметр b равен 2. При втором запуске параметр a содержит результат предыдущего выполнения функции, то если число 1 + 2 = 3, а параметр b равен 3 - следующее число в потоке.

Или другой пример со строками:

```
val people = listOf("Tom", "Bob", "Kate", "Sam", "Alice")
val reducedValue = people.reduce{ a, b -> "$a $b" }
println(reducedValue)   // Tom Bob Kate Sam Alice
```

Здесь reduce соединяет все строки в одну.

Функция fold также сводит все элементы потока в один. Но в отличие от reduce в качестве первого параметра принимает начальное значение:

```
fold(initial: R, operation: (acc: R, T) -> R): R
```

Например:

```
val people = listOf("Tom", "Bob", "Kate", "Sam", "Alice")
val foldedValue = people.fold("People:", { a, b -> "$a $b" })
println(foldedValue)   // People: Tom Bob Kate Sam Alice
```

В данном случае начальным значением является строка "People:", к которой затем добавляются остальные элементы списка.

Сложение, вычитание и объединение коллекций

С помощью функции plus можно складывать коллекции/последовательности с другими элементами или коллекциями/последовательностями. Данная функция принимает как одиночный элемент, так и коллекцию/последовательность:

```
fun main(){
    val people = listOf("Tom", "Bob", "Sam")
    // добавляем один объект
    val result1 = people.plus("Alice")
    println(result1)  // [Tom, Bob, Sam, Alice]

    val employees = listOf("Mike", "Kate")
    // добавляем коллекцию объектов
    val result2 = people.plus(employees)
    println(result2)  // [Tom, Bob, Sam, Mike, Kate]
}
```

Обратите внимание, что начальная коллекция, у которой вызывается функция minus() (в примере выше коллекция people), не изменяется, результатом объединения является новая коллекция.

Вместо функции plus() можно использовать оператор +

```
fun main(){
    val people = listOf("Tom", "Bob", "Sam")
    val result1 = people + "Alice"
    println(result1)  // [Tom, Bob, Sam, Alice]

    val employees = listOf("Mike", "Kate")
    val result2 = people + employees
    println(result2)  // [Tom, Bob, Sam, Mike, Kate]
}
```

Аналогичным образом можно использовать функцию minus() для вычитания либо одиночного объекта, либо коллекции/последовательности:

```
fun main(){
    val people = listOf("Tom", "Bob", "Sam", "Kate")
    // вычитаем один объект
    val result1 = people.minus("Bob")
    println(result1)  // [Tom, Sam, "Kate"]

    val employees = listOf("Mike", "Kate")
    // вычитаем коллекцию
    val result2 = people.minus(employees)
    println(result2)  // [Tom, Bob, Sam]
}
```

Начальная коллекция, у которой вызывается функция minus(), также не изменяется, результатом вычитания является новая коллекция.

Вместо функции minus() можно использовать оператор -

```
val people = listOf("Tom", "Bob", "Sam", "Kate")
val result1 = people - "Bob"
println(result1)  // [Tom, Sam, "Kate"]

val employees = listOf("Mike", "Kate")
val result2 = people - employees
println(result2)  // [Tom, Bob, Sam]
```

Объединение

Для объединения двух разных коллекций/последовательностей в одну применяется функция zip():

```
zip(other: Iterable<R>/Sequence<R>): List<Pair<T, R>>/Sequence<Pair<T, R>>
```

В качестве параметра она принимает другую коллекцию/последовательность и возвращает набор из объектов типа Pair

Например, объединим два списка в один:

```
val english = listOf("red", "blue", "green")
val russian = listOf("красный", "синий", "зеленый")
val words = english.zip(russian)

for(word in words)
    println("${word.first}: ${word.second}")
```

Здесь каждому элементу из списка english сопоставляется элемент на соответствующей позиции из списка russian. Результатом функции будет список из объектов Pair<String, String>, где свойство first хранит значение из текущей коллекции, а свойство second - значение из коллекции, переданной в качестве параметра:

```
red: красный
blue: синий
green: зеленый
```

Если в одной из коллекций меньше элементов, чем в другой, то сопоставляется столько элементов, сколько в наименьшей коллекции.

Функция zip() имеет еще одну версию, которая в качестве второго параметра получает функцию преобразования, применяемую к элементам обоих коллекций/последовательностей:

```
zip(other: Iterable<R>,transform: (a: T, b: R) -> V): List<V>
zip(other: Sequence<R>,transform: (a: T, b: R) -> V): Sequence<V>
```

Пример использования:

```
val english = listOf("red", "blue", "green")
val russian = listOf("красный", "синий", "зеленый")
val words = english.zip(russian){en, ru -> "$en - $ru"}

for(word in words) println(word)
```

Консольный вывод:

```
red - красный
blue - синий
green - зеленый
```

Также в Kotlin есть обратная функция - unzip, которая позволяет обратно получить две коллекции

```
unzip(): Pair<List<T>, List<R>>
```

Пример использования:

```
val dictionary = listOf("red", "blue", "green")
        .zip(listOf("красный", "синий", "зеленый"))
val words = dictionary.unzip()
println(words.first)    // [red, blue, green]
println(words.second)   // [красный, синий, зеленый]
```

Получение части элементов

slice

Функция slice() возвращает часть коллекции, элементы которой располагаются на определенных индексах. Индексы передаеются в функцию либо в виде диапазона IntRange, либо коллекции значений Iterable. Результатом функции является список List, который содержит элементы по указанным индексам:

```
val people = listOf("Tom", "Bob", "Sam", "Kate", "Alice", "Mike")

println(people.slice(1..3))     // [Bob, Sam, Kate]
println(people.slice(0..5 step 2))  // [Tom, Sam, Alice]
println(people.slice(listOf(1, 3, 5, 1)))   // [Bob, Kate, Mike, Bob]
```

Стоит отметить, что для последовательностей эта функция не доступна.

Take и takeWhile

Функция take() извлекает из начала коллекции/последовательности определенное количество элементов

```
take(n: Int): List<T>/Sequence<T>
```

Например, выберем 3 первых элементов:

```
val people = listOf("Tom", "Bob", "Sam", "Kate", "Alice", "Mike")
println(people.take(3))     // ["Tom", "Bob", "Sam"]
```

Для коллекций также доступна функция takeLast(), которая извлекает определенное количество элементов из конца коллекции

```
val people = listOf("Tom", "Bob", "Sam", "Kate", "Alice", "Mike")
// получаем три последних элемента
println(people.takeLast(3))     // ["Kate", "Alice", "Mike"]
```

Функция takeWhile() отбирает элементы с начала коллекции/последовательности, которые соответвуют условию предиката:

```
takeWhile(predicate: (T) -> Boolean): List<T>       // для коллекций
takeWhile(predicate: (T) -> Boolean): Sequence<T>   // для последовательностей
```

функция предиката получает в качестве параметра текущий элемент и возвращает true, если элемент соответствует условию:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.takeWhile{it.length == 3}
println(part)     // ["Tom", "Sam"]
```

В данном случае условие предиката возвращает true, если длина строки равна 3: {it.length == 3}, поэтому функция takeWhile отбирает строки, пока они соответствуют этому условию. Так, в данном случае в результирующем списке окажется первые два элемента: ["Tom", "Sam"]. Когда функция takeWhile() встретит хоть один элемент, который не соответствует условию, она завершит работу. Хотя в списке people есть еще элементы с длиной в три символа (элемент "Bob"). Поэтому если в списке первый элемент уже не соответствует условию предиката, то функция возвратит пустой список:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.takeWhile{it.length == 4}     // выбираем строки с длиной в 4
символа
println(part)     // [] - пустой список
```

Для коллекций также доступа функция takeLastWhile(), которая работает аналогичным образом, только выбирает элементы с конца списка

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.takeLastWhile{it.length !=3} // длина НЕ равна 3
println(part)     // ["Alice", "Mike"]
```

drop и dropWhile

Функция drop() позволяет пропустить определенное количество элементов в коллекции/последовательности:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.drop(3) // пропускаем первые 3 элемента
println(part)     // [Bob, Alice, Mike]
```

Для коллекций также доступна функция dropLast(), которая проупскает определенное количество элементов из конца коллекции

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.dropLast(2) // пропускаем последние 2 элемента
println(part)     // [Tom, Sam, Kate, Bob]
```

Функция dropWhile() пропускает элементы с начала коллекции/последовательности, которые соответвуют условию предиката:

```
dropWhile(predicate: (T) -> Boolean): List<T>       // для коллекций
dropWhile(predicate: (T) -> Boolean): Sequence<T>   // для последовательностей
```

функция предиката получает в качестве параметра текущий элемент и возвращает true, если элемент соответствует условию:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.dropWhile{it.length == 3}
println(part)     // [Kate, Bob, Alice, Mike]
```

В данном случае условие предиката возвращает true, если длина строки равна 3: {it.length == 3}, поэтому функция dropWhile() пропускает строки, пока они соответствуют этому условию. Так, в данном случае в результирующем списке окажется первые два элемента: [Kate, Bob, Alice, Mike]. Когда функция dropWhile() встретит хоть один элемент, который не соответствует условию, она завершит работу. Хотя в списке people есть еще элементы с длиной в три символа (элемент "Bob"). Поэтому если в списке первый элемент уже не соответствует условию предиката, то функция возвратит список со всеми элементами.

Для коллекций также доступа функция dropLastWhile(), которая работает аналогичным образом, только пропускает элементы с конца списка

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val part = people.dropLastWhile{it.length !=3} // длина НЕ равна 3
println(part)     // ["Tom", "Sam", "Kate", "Bob"]
```

Разделение на части

Функция chuncked() позволяет разбить коллекцию/последовательность на списки определенной длины:

```
chunked(size: Int): List<List<T>>       // для коллекций
chunked(size: Int): Sequence<List<T>>  // для последовательностей
```

В качестве параметра в функцию передается размер списков, в которые будут помещаться элементы:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val parts = people.chunked(3)
println(parts)     // [[Tom, Sam, Kate], [Bob, Alice, Mike]]
```

В данном случае разбиваем коллекцию на списки по 3 элемента, соответственно результирующая коллекция будет содержать два списка по три элемента.

Дополнительная версия функции в качестве второго параметра принимает функцию преобразования элементов:

```
chunked(size: Int, transform: (List<T>) -> R): List<R>      // для коллекций
chunked(size: Int,transform: (List<T>) -> R): Sequence<R> // для
последовательностей
```

То есть сначала коллекция/последовательность разбивается на несколько списков длиной size, затем каждый список передается в функцию преобразования, где на его основе можно сгенерировать новое значение. Например:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice", "Mike")
val parts = people.chunked(3){it.first()}
println(parts)     // [Tom, Bob]
```

В данном случае коллекция разделяется на два списка длиной по 3 элемента. В функции преобразования из каждого списка выбирается и возвращается первый элемент каждого списка. Соответственно результатом функции будет список из двух первых элементов.

Получение отдельных элементов

Получение элемента по индексу Для получения элемента по индексу применяется функция elementAt(), в которую передается индекс элемента:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
// получаем элемент по индексу 1
println(people.elementAt(1))     // Sam
```

Однако если переданный индекс выходит за границы коллекции/последовательности, то приложение сгенерирует ошибку. В этом случае мы можем использовать функцию elementAtOrNull(), которая возвращает null, если индекс вне границ:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
println(people.elementAtOrNull(1))     // Sam
println(people.elementAtOrNull(6))     // null
```

Также в этом случае можно использовать функцию elementAtOrElse():

```
elementAtOrElse(index: Int, defaultValue: (Int) -> T): T
```

Первый параметр функции - индекс элемента, а второй - функция, которая получает индекс и возвращает значение по умолчанию, если индекс находится вне границ коллекции/последовательности.

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")

println(people.elementAtOrElse(1){"Undefined"})     // Sam
println(people.elementAtOrElse(6){"Undefined"})     // Undefined
println(people.elementAtOrElse(8){"Index $it out of bounds"})  // Index 8 out of
bounds
```

Получение по условию

Если надо получить первый или последний элементы коллекции/последовательности, то можно использовать соответственно функции first() и last():

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
println(people.first())     // Tom
println(people.last())     // Alice
```

В качестве параметра эти функции могут принимать функцию предиката

```
first(predicate: (T) -> Boolean): T
last(predicate: (T) -> Boolean): T
```

Эти функции возвращают первый и последний элементы, которые соответствуют условию предиката:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
// первый элемент с длиной в 4 символа
println(people.first{it.length==4})     // Kate
// последний элемент с длиной в 3 символа
println(people.last{it.length==3})     // Bob
```

Однако если коллеккция пуста, или в коллекции/последовательности нет элементов, которые соответствуют условию, то программа генерирует ошибку. В этом случае можно применять функции firstOrNull() и lastOrNull(), которые возвращают null, если коллекция пуста или в ней нет элементов, соответствующих условию:

```
firstOrNull(): T?
firstOrNull(predicate: (T) -> Boolean): T?

lastOrNull(): T?
lastOrNull(predicate: (T) -> Boolean): T?
```

Пример применения:

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")

println(people.firstOrNull{it.length==3})     // Tom
println(people.firstOrNull{it.length==33})     // null

println(people.lastOrNull{it.length==3})     // Bob
println(people.lastOrNull{it.length==33})     // null
```

Выбор случайного элемента Для извлечения случайного элемента применяется функция random()

```
val people = listOf("Tom", "Sam", "Kate", "Bob", "Alice")
println(people.random())
```

Однако если коллекция/последовательность пуста, то эта функция генерирует ошибку. В этом случае можно использовать функцию randomOrNull(), которая в этом случае возвращает null:

```
val people = listOf<String>()
println(people.randomOrNull())     // null
```