## Kotlin Collections

Collection is a very important part of the data structure, which makes the software development easy for engineers.

A collection usually contains a number of objects (this number may also be zero) of the same type. Objects in a collection are called elements or items.

For example, all the students in a department form a collection that can be used to calculate their average age.

The following collection types are relevant for Kotlin:

**List** is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list.

**For example** of a list is a **sentence:** it's a group of words, their order is important, and they can repeat.

**Set** is a collection of unique elements.

It reflects the mathematical abstraction of set: a group of objects without repetitions.

Generally, the order of set elements has no significance.

**For example**, an alphabet is a set of letters.

**Map** (or dictionary) is a set of key-value pairs.

*Keys are unique*, and each of them maps to exactly one value.

The values can be **duplicates**.

Maps are useful for storing logical connections between objects,

**For example**, an employee's ID and their position.

# Kotlin For Android
# [5.0 : Collections]

Kotlin has two types of collection

- one is **immutable** collection (which means **lists**, maps and sets that cannot be editable) and

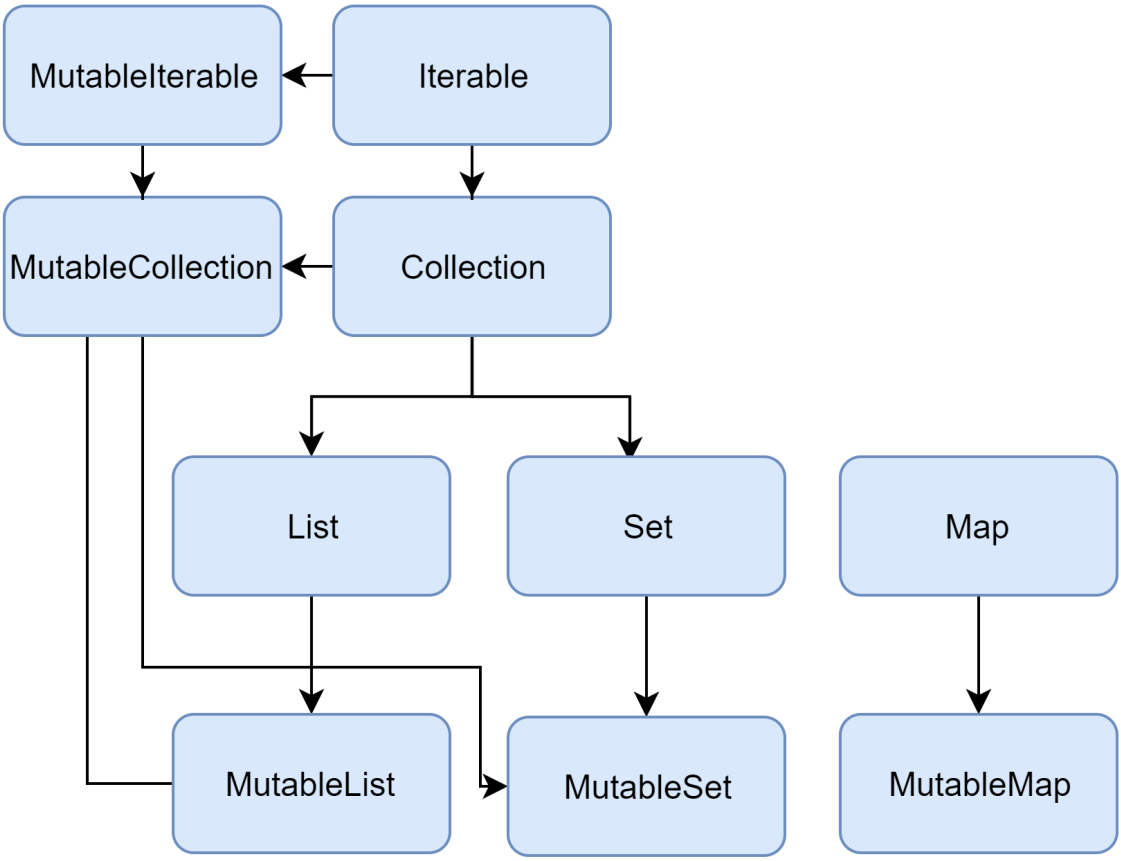
- **mutable** collection (this type of collection is editable).

It is very important to keep in mind the type of collection used in your application, as Kotlin system does not represent any specific difference in them.

```
fun main(args: Array<String>) {
      val numbers: MutableList<Int> = mutableListOf(1, 2, 3) //mutable List
      val readOnlyView: List<Int> = numbers // immutable list
      println("my mutable list--"+numbers)  // prints "[1, 2, 3]"
      numbers.add(4)
      println("my mutable list after addition --"+numbers) // prints "[1, 2, 3, 4]"
      println(readOnlyView)
      readOnlyView.clear()     // ⇒ does not compile
      // gives error
}
```

some useful methods such as first(), last(), filter(), etc.

```
fun main(args: Array<String>) {
      val items = listOf(1, 2, 3, 4)
      println("First Element of our list----"+items.first())
      println("Last Element of our list----"+items.last())
      println("Even Numbers of our List----"+items.
      filter { it % 2 == 0 })   // returns [2, 4]
      val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
      println(readWriteMap["foo"])  // prints "1"
      val strings = hashSetOf("a", "b", "c", "c")
      println("My Set Values are"+strings)
}
```


MutableIterable
Iterable
MutableCollection
Collection
List
Set
Map
MutableList
MutableSet
MutableMap

## Collection

**Collection<T>** is the root of the collection hierarchy.

This interface represents the common behavior of a read-only collection: retrieving size, checking item membership, and so on.

Collection inherits from the **Iterable<T>** interface that defines the operations for iterating elements.

You can use **Collection** as a parameter of a function that applies to different collection types.

For more specific cases, use the Collection's inheritors: **List** and **Set**.

```
fun printAll(strings: Collection<String>) {
        for(s in strings) print("$s ")
        println()
    }
fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

**MutableCollection** is a Collection with write operations, such as add and remove.

```
fun List<String>.getShortWordsTo(
          shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength}
     // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}
fun main() {
    val words = "A long time ago in a galaxy far far
                    away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
```

```
    println(shortWords)
}
```

**List**

List<T> stores elements in a specified order and provides indexed access to them.

Indices start from zero — the index of the first element — and go to lastIndex which is the (list.size – 1).

```
val numbers = listOf("one", "two", "three", "four")
     println("Number of elements: ${numbers.size}")
     println("Third element: ${numbers.get(2)}")
     println("Fourth element: ${numbers[3]}")
     println("Index of element \"two\" $
     {numbers.indexOf("two")}")
```

List elements (including nulls) **can duplicate**: *a list can contain any number of equal objects or occurrences of a single object*.

Two lists are considered equal if they have the same sizes and *structurally equal elements at the same positions*.

```
     val bob = Person("Bob", 31)
     val people1 = listOf<Person>(
                         Person("Adam", 20),
                         bob,
                         bob
                    )
     val people2 = listOf<Person>(
                         Person("Adam", 20),
                         Person("Bob", 31),
                         bob
                    )
          println(people1 == people2)
          bob.age = 32
          println(people1 == people2)
```

MutableList is a List with list-specific write operations,

for example, to add or remove an element at a specific position.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

As you see, in some aspects lists are very similar to **arrays**.

However, there is one important difference: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of List is ArrayList which you can think of as a resizable array.

**Set**

**Set<T>** stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null.

Two sets are equal if they have the same size, and for each element of a set there is an *equal element in the other set*.

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")
val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

**MutableSet is a Set with write operations from MutableCollection.**

The default implementation of Set – LinkedHashSet – preserves the order of elements insertion. Hence, the functions that rely on the order, such as first() or last(), return predictable results on such sets.

```
val numbers = setOf(1, 2, 3, 4)
      // LinkedHashSet is the default implementation
val numbersBackwards = setOf(4, 3, 2, 1)
println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

An alternative implementation – **HashSet** – says nothing about the elements order, so calling such functions on it returns unpredictable results.

However, HashSet requires less memory to store the same number of elements.

### Map

**Map<K, V>** is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well.

A Map stores **key-value pairs** (or entries); *keys are unique*, but different keys can be paired with equal values.

The Map interface **provides specific functions**, such as access to value by key, searching keys and values, and so on.

```
val numbersMap = mapOf("key1" to 1,
                       "key2" to 2,
                       "key3" to 3,
                       "key4" to 1)
println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap)
      println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values)
      println("The value 1 is in the map")
if (numbersMap.containsValue(1))
      println("The value 1 is in the map") // same as previous
```

Two maps containing the equal pairs are equal regardless of the pair order.

```
val numbersMap = mapOf("key1" to 1,
                       "key2" to 2,
                       "key3" to 3,
                       "key4" to 1)
val anotherMap = mapOf("key2" to 2,
                       "key1" to 1,
                       "key4" to 1,
                       "key3" to 3)
println("The maps are equal: ${numbersMap == anotherMap}")
```

**MutableMap** is a **Map** with map write operations,

for example, you can add a new key-value pair or update the value associated with the given key.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11
println(numbersMap)
```

The default implementation of **Map** – **LinkedHashMap** – preserves the order of elements insertion when iterating the map.

In turn, an alternative implementation – **HashMap** – says nothing about the elements order.

**Ranges**

Ranges is another unique characteristic of Kotlin.

It provides an operator that helps you iterate through a range.

Internally, it is implemented using rangeTo() and its operator form is (..).

```
fun main(args: Array<String>) {
     val i:Int  = 2
     for (j in 1..4)
          print(j) // prints "1234"
     if (i in 1..10) { // equivalent of 1 < = i && i < = 10
          println("we found your number --"+i)
     }
}
```

## Constructing from elements

The most common way to create a collection is with the standard library functions listOf<T>(), setOf<T>(), mutableListOf<T>(), mutableSetOf<T>(). If you provide a comma-separated list of collection elements as arguments, the compiler detects the element type automatically. When creating empty collections, specify the type explicitly.

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

The same is available for maps with the functions mapOf() and mutableMapOf(). The map's keys and values are passed as Pair objects (usually created with to infix function).

```
val numbersMap = mapOf("key1" to 1, "key2" to 2,
                       "key3" to 3, "key4" to 1)
```

Note that the to notation creates a short-living Pair object, so it's recommended that you use it only if performance isn't critical.

To avoid excessive memory usage, use alternative ways. For example, you can create a mutable map and populate it using the write operations.

The **apply()** function can help to keep the initialization fluent here.

```
val numbersMap = mutableMapOf<String, String>().apply
    { this["one"] = "1"; this["two"] = "2" }
```

## Empty collections

There are also functions for creating collections without any elements: **emptyList()**, **emptySet()**, and **emptyMap()**.

When creating empty collections, you should specify the type of elements that the collection will hold.

```
val empty = emptyList<String>()
```

## Initializer functions for lists

For lists, there is a constructor that takes the list size and the initializer function that defines the element value based on its index.

```
val doubled = List(3, { it * 2 })
// or MutableList if you want to change its content later
println(doubled)
```

### Concrete type constructors

To create a concrete type collection, such as an **ArrayList** or **LinkedList**, you can use the available constructors for these types.

Similar constructors are available for implementations of Set and Map.

```
val linkedList = LinkedList<String>(listOf("one", "two",
"three"))
val presizedSet = HashSet<Int>(32)
```

## Copying

To create a collection with the same elements as an existing collection, you can use copying operations.

Collection copying operations from the standard library create shallow copy collections with references to the same elements.

Thus, a change made to a collection element reflects in all its copies.

Collection copying functions, such as toList(), toMutableList(), toSet() and others, create a snapshot of a collection at a specific moment.

Their result is a new collection of the same elements.

If you add or remove elements from the original collection, this won't affect the copies.

Copies may be changed independently of the source as well.

```
val sourceList = mutableListOf(1, 2, 3)
val copyList = sourceList.toMutableList()
val readOnlyCopyList = sourceList.toList()
sourceList.add(4)
     println("Copy size: ${copyList.size}")
     //readOnlyCopyList.add(4)
     // compilation error
     println("Read-only copy size: ${copyList.size}")
```

These functions can also be used for converting collections to other types, for example, build a set from a list or vice versa.

```
     val sourceList = mutableListOf(1, 2, 3)
     val copySet = sourceList.toMutableSet()
     copySet.add(3)
     copySet.add(4)
     println(copySet)
```

Alternatively, you can create new references to the same collection instance.

New references are created when you initialize a collection variable with an existing collection.

So, when the collection instance is altered through a reference, the changes are reflected in all its references.

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList =  sourceList
referenceList.add(4)
println("Source size: ${sourceList.size}")
```

Collection initialization can be used for restricting mutability.

For example, if you create a List reference to a MutableList, the compiler will produce errors if you try to modify the collection through this reference.

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList: List<Int> = sourceList
//referenceList.add(4)
//compilation error
sourceList.add(4)
println(referenceList)
// shows the current state of sourceList
```

## Invoking functions on other collections

Collections can be created in result of various operations on other collections.

For example, filtering a list creates a new list of elements that match the filter:

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)
```

**Mapping** produces a list of a transformation results:

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

**Association** produces maps:

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

## Iterators

For traversing collection elements, the Kotlin standard library supports the commonly used mechanism of iterators – objects that provide access to the elements sequentially without exposing the underlying structure of the collection.

**Iterators** are useful when you need to process all the elements of a collection one-by-one,

for example, print values or make similar updates to them.

Iterators can be obtained for inheritors of the Iterable<T> interface, including Set and List, by calling the iterator() function.

Once you obtain an iterator, it points to the first element of a collection; calling the next() function returns this element and moves the iterator position to the following element if it exists.

Once the iterator passes through the last element, it can no longer be used for retrieving elements; neither can it be reset to any previous position.

To iterate through the collection again, create a new iterator.

```
val numbers = listOf("one", "two", "three", "four")
val numbersIterator = numbers.iterator()
while (numbersIterator.hasNext()) {
    println(numbersIterator.next())
}
```

Another way to go through an Iterable collection is the well-known for loop.

When using for on a collection, you obtain the iterator implicitly.

So, the following code is equivalent to the example above:

```
val numbers = listOf("one", "two", "three", "four")
for (item in numbers) {
    println(item)
}
```

Finally, there is a useful **forEach()** function that lets you automatically iterate a collection and execute the given code for each element.
So, the same example would look like this:

```
val numbers = listOf("one", "two", "three", "four")
numbers.forEach {
    println(it)
}
```

## List iterators

- For lists, there is a special iterator implementation: **ListIterator**.
- It supports iterating lists in both directions: forwards and backwards.
- Backward iteration is implemented by the functions hasPrevious() and previous().
- Additionally, the **ListIterator** provides information about the element indices with the functions nextIndex() and previousIndex().

```
val numbers = listOf("one", "two", "three", "four")
val listIterator = numbers.listIterator()
while (listIterator.hasNext()) listIterator.next()
println("Iterating backwards:")
while (listIterator.hasPrevious()) {
    print("Index: ${listIterator.previousIndex()}")
    println(", value: ${listIterator.previous()}")
}
```

Having the ability to iterate in both directions, means the ListIterator can still be used after it reaches the last element.

## Mutable iterators

For iterating mutable collections, there is MutableIterator that extends Iterator with the element removal function remove().
So, you can remove elements from a collection while iterating it.

```
val numbers = mutableListOf("one", "two", "three",
"four")
val mutableIterator = numbers.iterator()
mutableIterator.next()
mutableIterator.remove()
println("After removal: $numbers")
```

In addition to removing elements, the **MutableListIterator** can also insert and replace elements while iterating the list.

```
val numbers = mutableListOf("one", "four", "four")
val mutableListIterator = numbers.listIterator()

mutableListIterator.next()
mutableListIterator.add("two")
mutableListIterator.next()
mutableListIterator.set("three")
println(numbers)
```

## Ranges and Progressions

Kotlin lets you easily create ranges of values using the rangeTo() function from the kotlin.ranges package and its operator form ... Usually, rangeTo() is complemented by in or !in functions.

```
if(i in 1..4) {  // equivalent of 1 <= i && i <= 4
    print(i)
}
```

Integral type ranges (IntRange, LongRange, CharRange) have an extra feature: they can be iterated over.
These ranges are also progressions of the corresponding integral types.
Such ranges are generally used for iteration in the for loops.

```
for (i in 1..4) print(i)
```

To iterate numbers in reverse order, use the downTo function instead of ..

```
for (i in 4 downTo 1) print(i)
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1).
This is done via the step function.

```
for (i in 1..8 step 2) print(i)
println()
for (i in 8 downTo 1 step 2) print(i)
```

To iterate a number range which does not include its end element, use the until function:

```
for (i in 1 until 10) {
     // i in [1, 10), 10 is excluded
    print(i)
}
```

## Range

A range defines a closed interval in the mathematical sense: it is defined by its **two** endpoint values which are both included in the range.
Ranges are defined for comparable types: having an order, you can define whether an arbitrary instance is in the range between two given instances.

The main operation on ranges is contains, which is usually used in the form of in and !in operators.

To create a range for your class, call the **rangeTo()** function on the range start value and provide the end value as an argument. rangeTo() is often called in its operator form ..

```
val versionRange = Version(1, 11)..Version(1, 30)
println(Version(0, 9) in versionRange)
println(Version(1, 20) in versionRange)
```

## Progression

As shown in the examples above, the ranges of integral types, such as **Int**, **Long**, and **Char**, can be treated as arithmetic progressions of them.
In Kotlin, these *progressions* are defined by special types: *IntProgression*, *LongProgression*, and *CharProgression*.

**Progressions** have **three** essential properties:

- the first element,
- the last element, and
- a non-zero step.

The first element is first, subsequent elements are the previous element plus a step.
The last element is always hit by iteration unless the progression is empty.
Iteration over a progression with a positive step is equivalent to an indexed for loop in Java/JavaScript.

```
for (int i = first; i <= last; i += step) {
```

```
    // ...
}
```

When you create a **progression** implicitly by iterating a range, this progression's first and last elements are the range's endpoints, and the step is 1.

```
for (i in 1..10) print(i)
```

To define a custom progression step, use the step function on a range.

```
for (i in 1..8 step 2) print(i)
```

The last element of the progression is calculated to find the maximum value not greater than the end value for a positive step or the minimum value not less than the end value for a negative step such that `(last - first) % step == 0`.

To create a progression for iterating in reverse order, use downTo instead of `..` when defining the range for it.

```
for (i in 4 downTo 1) print(i)
```

Progressions implement `Iterable<N>`, where `N` is `Int`, `Long`, or `Char` respectively, so you can use them in various collection functions like `map`, `filter`, and other.

```
println((1..10).filter { it % 2 == 0 })
```

## Sequences

Along with collections, the Kotlin standard library contains another container type — sequences (Sequence<T>).
Sequences offer the same functions as Iterable but implement another approach to multi-step collection processing.

When the processing of an Iterable includes multiple steps, they are executed eagerly: each processing step completes and returns its result — an intermediate collection.

The following **step** executes on this collection.

In turn, multi-step processing of sequences is executed lazily when possible: actual computing happens only when the result of the whole processing chain is requested.

The order of operations execution is different as well: Sequence performs all the processing steps one-by-one for every single element.

In turn, Iterable completes each step for the whole collection and then proceeds to the next step.

So, the sequences let you avoid building results of intermediate steps, therefore improving the performance of the whole collection processing chain.

However, the lazy nature of sequences adds some overhead which may be significant when processing smaller collections or doing simpler computations.

Hence, you should consider both *Sequence* and *Iterable* and decide which one is better for your case.

## Constructing

**From elements**

To create a sequence, call the sequenceOf() function listing the elements as its arguments.

```
val numbersSequence = sequenceOf(
                      "four", "three", "two", "one")
```

**From Iterable**

If you already have an Iterable object (such as a List or a Set), you can create a sequence from it by calling asSequence().

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

**From function**

One more way to create a sequence is by building it with a function that calculates its elements.

To build a sequence based on a function, call generateSequence() with this function as an argument.

Optionally, you can specify the first element as an explicit value or a result of a function call.

The sequence generation stops when the provided function returns null.

So, the sequence in the example below is infinite.

```
val oddNumbers = generateSequence(1) { it + 2 }
    // `it` is the previous element
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())
    // error: the sequence is infinite
```

To create a finite sequence with generateSequence(), provide a function that returns null after the last element you need.

```
val oddNumbersLessThan10 = generateSequence(1) {
          if (it < 10) it + 2
          else null
}
println(oddNumbersLessThan10.count())
```

### From chunks

Finally, there is a function that lets you produce sequence elements **one by one** or by chunks of arbitrary sizes — the sequence() function.

This function takes a **lambda expression** containing calls of yield() and yieldAll() functions.

They return an element to the sequence consumer and suspend the execution of sequence() until the next element is requested by the consumer.

**yield()** takes a single element as an argument; yieldAll() can take an Iterable object, an **Iterator**, or another **Sequence**.

A **Sequence** argument of yieldAll() can be infinite. However, such a call must be the last: all subsequent calls will never be executed.

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
```

## Sequence operations

The sequence operations can be classified into the following groups regarding their state requirements:

- Stateless operations require no state and process each element independently, for example, map() or filter(). Stateless operations can also require a small constant amount of state to process an element, for example, take() or drop().
- Stateful operations require a significant amount of state, usually proportional to the number of elements in a sequence.

If a sequence operation returns another sequence, which is produced lazily, it's called intermediate.
Otherwise, the operation is terminal.
Examples of terminal operations are toList() or sum().
Sequence elements can be retrieved only with terminal operations.

Sequences can be iterated multiple times; however some sequence implementations might constrain themselves to be iterated only once.
That is mentioned specifically in their documentation.

## Sequence processing example

Let's take a look at the difference between Iterable and Sequence with an example.

### Iterable

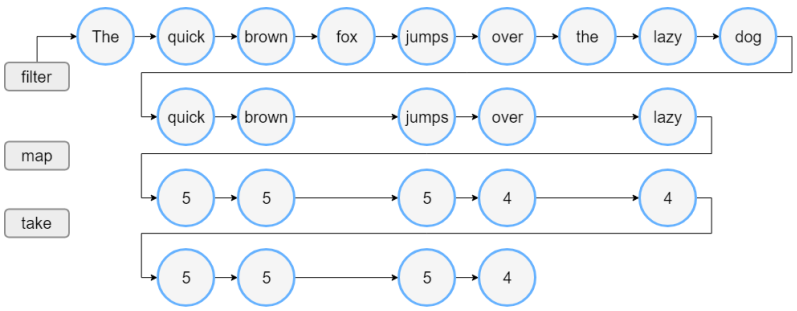
Assume that you have a list of words.
The code below filters the words longer than three characters and prints the lengths of first four such words.

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

**output:**

```
filter: The
filter: quick
filter: brown
filter: fox
filter: jumps
filter: over
filter: the
filter: lazy
filter: dog
length: 5
length: 5
length: 5
length: 4
length: 4
Lengths of first 4 words longer than 3 chars:
[5, 5, 5, 4]
```




When you run this code, you'll see that the filter() and map() functions are executed in the same order as they appear in the code. First, you see filter: for all elements, then length: for the elements left after filtering, and then the output of the two last lines.

## Sequence

Now let's write the same with sequences:

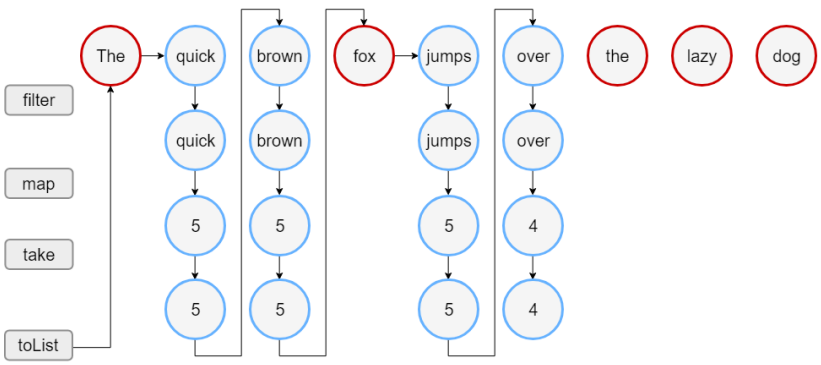
```
val words = "The quick brown fox jumps
              over the lazy dog".split(" ")
//convert the List to a Sequence
val wordsSequence = words.asSequence()

val lengthsSequence = wordsSequence.filter {
              println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars")
// terminal operation: obtaining the result as a List
println(lengthsSequence.toList())
```

**Output:**

```
Lengths of first 4 words longer than 3 chars
filter: The
filter: quick
length: 5
filter: brown
length: 5
filter: fox
filter: jumps
length: 5
filter: over
length: 4
[5, 5, 5, 4]
```




The output of this code shows that the filter() and map() functions are called only when building the result list. So, you first see the line of text “Lengths of..” and then the sequence processing starts. Note that for elements left after filtering, the map executes before filtering the next element. When the result size reaches 4, the processing stops because it's the largest possible size that take(4) can return.

## Collection Operations Overview

The Kotlin standard library offers a broad variety of functions for performing operations on collections.

This includes simple operations, such as getting or adding elements, as well as more complex ones including search, sorting, filtering, transformations, and so on.

## Extension and member functions

Collection operations are declared in the standard library in two ways: member functions of collection interfaces and extension functions.

Member functions define operations that are essential for a collection type.

For example, Collection contains the function **isEmpty()** for checking its emptiness; List contains **get()** for index access to elements, and so on.

When you create own implementations of collection interfaces, you must implement their member functions. To make the creation of new implementations easier, use the skeletal implementations of collection interfaces from the standard library: AbstractCollection, AbstractList, AbstractSet, AbstractMap, and their mutable counterparts.

Other collection operations are declared as extension functions. These are filtering, transformation, ordering, and other collection processing functions.

## Collection Operations Overview

The Kotlin standard library offers a broad variety of functions for performing operations on collections. This includes simple operations, such as getting or adding elements, as well as more complex ones including search, sorting, filtering, transformations, and so on.

## Extension and member functions

Collection operations are declared in the standard library in two ways: member functions of collection interfaces and extension functions.

Member functions define operations that are essential for a collection type.

For example, Collection contains the function **isEmpty()** for checking its emptiness; List contains **get()** for index access to elements, and so on.

When you create own implementations of collection interfaces, you must implement their member functions.

To make the creation of new implementations easier, use the skeletal implementations of collection interfaces from the standard library: AbstractCollection, AbstractList, AbstractSet, AbstractMap, and their mutable counterparts.

Other collection operations are declared as extension functions.

These are filtering, transformation, ordering, and other collection processing functions.

## Common operations

Common operations are available for both read-only and mutable collections. Common operations fall into these groups:

- Transformations
- Filtering
- plus and minus operators
- Grouping
- Retrieving collection parts
- Retrieving single elements
- Ordering
- Aggregate operations

Operations described on these pages return their results without affecting the original collection.
For example, a filtering operation produces a new collection that contains all the elements matching the filtering predicate.
Results of such operations should be either stored in variables, or used in some other way,
for example, passed in other functions.

```
val numbers = listOf("one", "two", "three", "four")
numbers.filter { it.length > 3 }
    // nothing happens with `numbers`, result is lost
println("numbers are still $numbers")
val longerThan3 = numbers.filter { it.length > 3 } // result is stored in `longerThan3`
println("numbers longer than 3 chars are $longerThan3")
```

**output:**

- numbers are still [one, two, three, four]
- numbers longer than 3 chars are [three, four]

**For certain collection operations,**
there is an option to specify the **destination object**.
**Destination** is a mutable collection to which the function appends its resulting items instead of returning them in a new object.
For performing operations with destinations, there are separate functions with the To postfix in their names,
**for example**, filterTo() instead of filter() or associateTo() instead of associate().
These functions take the destination collection as an additional parameter.

```
val numbers = listOf("one", "two", "three", "four")
val filterResults = mutableListOf<String>()  //destination object
numbers.filterTo(filterResults) { it.length > 3 }
numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
println(filterResults) // contains results of both operations
```

**Output:** [three, four, one]

**For convenience**, these functions return the ***destination collection*** back, so you can create it right in the corresponding argument of the function call:

```
// filter numbers right into a new hash set,
// thus eliminating duplicates in the result
val result = numbers.mapTo(HashSet()) { it.length }
println("distinct item lengths are $result")
```

**Output:** distinct item lengths are [3, 4, 5]

## Write operations

- For mutable collections, there are also write operations that change the collection state.
- Such operations include adding, removing, and updating elements.
- Write operations are listed in the Write operations and corresponding sections of **List** specific operations and **Map** specific operations.

For certain operations, there are pairs of functions for performing the same operation: one applies the operation in-place and the other returns the result as a separate collection.
**For example**,
✔ **sort()** sorts a mutable collection in-place, so it's state changes;
✔ **sorted()** creates a new collection that contains the same elements in the sorted order.

```
val numbers = mutableListOf("one", "two", "three", "four")
val sortedNumbers = numbers.sorted()
println(numbers == sortedNumbers)  // false
numbers.sort()
println(numbers == sortedNumbers)  // true
```

## Collection Write Operations

Mutable collections support operations for changing the collection contents, for example, **adding** or **removing** elements. We'll describe write operations available for all implementations of ***MutableCollection***.

### Adding elements

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
println(numbers)
```

**Output:** [1, 2, 3, 4, 5]

```
addAll():
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)  //  [1, 2, 5, 6, 7, 8]
    numbers.addAll(2, setOf(3, 4))
    println(numbers)  // [1, 2, 3, 4, 5, 6, 7, 8]
```

You can also add elements using the **in-place** version of the plus operator - plusAssign (+=) When applied to a mutable collection, **+=** appends the second operand (an element or another collection) to the end of the collection.

```
val numbers = mutableListOf("one", "two")
numbers += "three"
println(numbers)     // [one, two, three]
numbers += listOf("four", "five")
println(numbers)     // [one, two, three, four, five]
```

### Removing elements:

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.remove(3)    // removes the first `3`
println(numbers)
numbers.remove(5)    // removes nothing
println(numbers)
```

For removing multiple elements at once, there are the following functions :

- **removeAll()** removes all elements that are present in the argument collection.
  - Alternatively, you can call it with a predicate as an argument; in this case the function removes all elements for which the predicate yields true.
- **retainAll()** is the opposite of removeAll(): it removes all elements except the ones from the argument collection.
  - When used with a predicate, it leaves only elements that match it.
- **clear()** removes all elements from a list and leaves it empty.

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers) // [1, 2, 3, 4]
numbers.retainAll { it >= 3 }
println(numbers) // [3, 4]
numbers.clear()
println(numbers) // []

val numbersSet = mutableSetOf("one", "two", "three",
"four")
numbersSet.removeAll(setOf("one", "two"))
println(numbersSet)  // [three, four]
```

Another way to remove elements from a collection is with the minusAssign (-=) operator — the in-place version of minus.

```
val numbers = mutableListOf("one", "two", "three",
"three", "four")
numbers -= "three"
println(numbers) // [one, two, three, four]
numbers -= listOf("four", "five")
//numbers -= listOf("four")    // does the same as above
println(numbers) // [one, two, three]
```

**List** Specific Operations

https://kotlinlang.org/docs/reference/list-operations.html

**Set** Specific Operations

https://kotlinlang.org/docs/reference/set-operations.html

**Map** Specific Operations

https://kotlinlang.org/docs/reference/map-operations.html