

Kotlin Collection Operators

The Complete Mental Map You Should Memorize

Use the category to identify the problem, then choose the operator. Examples use a Student/Course model where practical.

FILTERING

- |-- filter
- |-- filterNot
- |-- filterIndexed
- |-- distinct
- |-- distinctBy
- |-- take
- |-- takeWhile
- |-- takeLast
- |-- drop
- |-- dropWhile
- |-- dropLast
- `-- partition

TRANSFORMATION

- |-- map
- |-- mapIndexed
- |-- mapNotNull
- |-- mapIndexedNotNull
- |-- flatMap
- |-- flatMapIndexed
- `-- flatten

SEARCHING

- |-- find
- |-- findLast
- |-- first
- |-- firstOrNull
- |-- last
- |-- lastOrNull
- |-- elementAt
- `-- elementAtOrNull

CHECKING

- |-- any
- |-- all
- `-- none

COUNTING / AGGREGATION

- |-- count
- |-- sum
- |-- sumOf
- |-- average
- |-- minOrNull
- |-- maxOrNull
- |-- minByOrNull
- |-- maxByOrNull
- |-- minWithOrNull
- |-- maxWithOrNull
- |-- fold
- |-- foldRight
- |-- reduce
- `-- reduceRight

SORTING

- |-- sorted
- |-- sortedDescending
- |-- sortedBy
- |-- sortedByDescending
- |-- sortedWith
- `-- reversed

GROUPING / MAP CREATION

- |-- groupBy

```

|-- groupingBy
|-- eachCount
|-- associate
|-- associateBy
`-- associateWith

PAIRING / COMBINING
|-- zip
|-- unzip
|-- zipWithNext
|-- plus (+)
`-- minus (-)

WINDOWING / CHUNKING
|-- chunked
|-- windowed
|-- windowedSequence
`-- runningFold / runningReduce

```

Example data

```

data class Student(
    val name: String,
    val age: Int,
    val courses: List<String>,
    val marks: Int
)

val students = listOf(
    Student("Arun", 22, listOf("Kotlin", "Android"), 80),
    Student("Priya", 24, listOf("Java", "Kotlin"), 90),
    Student("Rahul", 21, listOf("Android", "Flutter"), 70),
    Student("Meena", 23, listOf("Kotlin", "Compose"), 85)
)

```

FILTERING

filter - Keep elements matching a condition.

```

students.filter { it.age > 22 }
// [Priya, Meena]

```

filterNot - Keep elements that do not match.

```

students.filterNot { it.age > 22 }
// [Arun, Rahul]

```

filterIndexed - Filter using both index and element.

```

students.filterIndexed { index, student ->
    index % 2 == 0
}
// [Arun, Rahul]

```

distinct - Remove duplicate values.

```

listOf(1, 2, 2, 3, 3).distinct()
// [1, 2, 3]

```

distinctBy - Remove duplicates based on a selected property.

```

students.distinctBy { it.age }
// one student per age

```

take - Take the first N elements.

```

students.take(2)
// [Arun, Priya]

```

takeWhile - Take elements while the condition remains true.

```

listOf(2, 4, 6, 7, 8).takeWhile { it % 2 == 0 }
// [2, 4, 6]

```

takeLast - Take the last N elements.

```

students.takeLast(2)
// [Rahul, Meena]

```

drop - Skip the first N elements.

```
students.drop(2)
// [Rahul, Meena]
```

dropWhile - Skip elements while the condition remains true.

```
listOf(2, 4, 7, 8).dropWhile { it % 2 == 0 }
// [7, 8]
```

dropLast - Remove the last N elements.

```
students.dropLast(2)
// [Arun, Priya]
```

partition - Split into matching and non-matching lists.

```
val (passed, failed) =
    students.partition { it.marks >= 80 }
```

TRANSFORMATION

map - Transform every element.

```
students.map { it.name }
// [Arun, Priya, Rahul, Meena]
```

mapIndexed - Transform using both index and element.

```
students.mapIndexed { index, student ->
    "$index-${student.name}"
}
// [0-Arun, 1-Priya, 2-Rahul, 3-Meena]
```

mapNotNull - Transform and discard null results.

```
listOf("1", "abc", "2").mapNotNull { it.toIntOrNull() }
// [1, 2]
```

mapIndexedNotNull - Indexed transform while discarding null results.

```
listOf("10", "x", "20").mapIndexedNotNull { index, value ->
    value.toIntOrNull()?.let { index to it }
}
// [(0,10), (2,20)]
```

flatMap - Map each element to a collection, then flatten.

```
students.flatMap { it.courses }
// one flat List<String> of all courses
```

flatMapIndexed - Flatten while also receiving the outer index.

```
students.flatMapIndexed { index, student ->
    student.courses.map { index to it }
}
```

flatten - Flatten a nested collection without transforming it first.

```
listOf(listOf(1, 2), listOf(3, 4)).flatten()
// [1, 2, 3, 4]
```

SEARCHING

find - First matching element, or null.

```
students.find { it.name == "Priya" }
// Priya
```

findLast - Last matching element, or null.

```
listOf(1, 2, 3, 2).findLast { it == 2 }
// 2 (the last one)
```

first - First element/match; throws if absent.

```
students.first { it.marks >= 80 }
// Arun
```

firstOrNull - First match, or null if absent.

```
students.firstOrNull { it.marks > 100 }
```

```
// null
```

last - Last element/match; throws if absent.

```
students.last { it.marks >= 80 }  
// Meena
```

lastOrNull - Last match, or null if absent.

```
students.lastOrNull { it.marks > 100 }  
// null
```

elementAt - Get an element by zero-based index; throws if invalid.

```
students.elementAt(1)  
// Priya
```

elementAtOrNull - Get by index, or null if invalid.

```
students.elementAtOrNull(10)  
// null
```

CHECKING

any - True when at least one element matches.

```
students.any { it.marks > 85 }  
// true
```

all - True only when every element matches.

```
students.all { it.marks >= 40 }  
// true
```

none - True when no element matches.

```
students.none { it.marks < 40 }  
// true
```

COUNTING / AGGREGATION

count - Count elements, optionally by condition.

```
students.count { it.marks >= 80 }  
// 3
```

sum - Sum numeric elements.

```
listOf(10, 20, 30).sum()  
// 60
```

sumOf - Sum a selected numeric property.

```
students.sumOf { it.marks }  
// 325
```

average - Average numeric values.

```
students.map { it.marks }.average()  
// 81.25
```

minOrNull - Smallest comparable value, or null.

```
listOf(8, 3, 5).minOrNull()  
// 3
```

maxOrNull - Largest comparable value, or null.

```
listOf(8, 3, 5).maxOrNull()  
// 8
```

minByOrNull - Object with the smallest selected value.

```
students.minByOrNull { it.age }  
// Rahul
```

maxByOrNull - Object with the largest selected value.

```
students.maxByOrNull { it.marks }  
// Priya
```

minWithOrNull - Object with the smallest value according to a comparator.

```
students.minWithOrNull(compareBy { it.marks })
```

```
// Rahul
```

maxWithOrNull - Object with the largest value according to a comparator.

```
students.maxWithOrNull(compareBy { it.marks })  
// Priya
```

fold - Accumulate from an explicit initial value.

```
listOf(1,2,3,4).fold(0) { acc, n -> acc + n }  
// 10
```

foldRight - Accumulate from right to left with an initial value.

```
listOf(1,2,3).foldRight(0) { n, acc -> n - acc }  
// 2
```

reduce - Accumulate using the first element as the initial value.

```
listOf(1,2,3,4).reduce { acc, n -> acc + n }  
// 10
```

reduceRight - Reduce from right to left.

```
listOf(1,2,3).reduceRight { n, acc -> n - acc }  
// 2
```

SORTING

sorted - Sort comparable values ascending.

```
listOf(5, 2, 8, 1).sorted()  
// [1, 2, 5, 8]
```

sortedDescending - Sort comparable values descending.

```
listOf(5, 2, 8, 1).sortedDescending()  
// [8, 5, 2, 1]
```

sortedBy - Sort objects by a selected property.

```
students.sortedBy { it.age }  
// Rahul, Arun, Meena, Priya
```

sortedByDescending - Sort objects by a property descending.

```
students.sortedByDescending { it.marks }  
// Priya, Meena, Arun, Rahul
```

sortedWith - Sort using a custom comparator.

```
students.sortedWith(compareBy<Student> { it.age }.thenBy { it.name })
```

reversed - Reverse the current order.

```
students.reversed()  
// Meena, Rahul, Priya, Arun
```

GROUPING / MAP CREATION

groupBy - Create Map>.

```
students.groupBy { it.age }  
// 22 -> [Arun], 24 -> [Priya], ...
```

groupingBy - Create a Grouping for grouped aggregations.

```
students.groupingBy { it.age }.eachCount()  
// 22 -> 1, 24 -> 1, 21 -> 1, 23 -> 1
```

eachCount - Count elements inside each group.

```
students.groupingBy { it.courses.size }.eachCount()  
// 2 -> 4
```

associate - Create key-value pairs from each element.

```
students.associate { it.name to it.marks }  
// {Arun=80, Priya=90, Rahul=70, Meena=85}
```

associateBy - Use a selected property as key; object becomes the value.

```
students.associateBy { it.name }  
// Arun -> Student(...), Priya -> Student(...)
```

associateWith - Use each element as key and calculate its value.

```
students.associateWith { it.marks }  
// Student(...) -> 80, ...
```

PAIRING / COMBINING

zip - Pair elements from two collections by index.

```
listOf("Arun", "Priya").zip(listOf(80, 90))  
// [(Arun, 80), (Priya, 90)]
```

unzip - Split pairs into two collections.

```
listOf("Arun" to 80, "Priya" to 90).unzip()  
// ([Arun, Priya], [80, 90])
```

zipWithNext - Pair adjacent elements.

```
listOf(10, 20, 30).zipWithNext()  
// [(10,20), (20,30)]
```

plus (+) - Create a collection containing the original elements plus another collection/value.

```
listOf(1, 2) + 3  
// [1, 2, 3]
```

minus (-) - Create a collection with matching element(s) removed.

```
listOf(1, 2, 3, 2) - 2  
// [1, 3, 2]
```

WINDOWING / CHUNKING

chunked - Split into fixed-size chunks.

```
listOf(1,2,3,4,5).chunked(2)  
// [[1,2], [3,4], [5]]
```

windowed - Create overlapping windows.

```
listOf(1,2,3,4).windowed(3)  
// [[1,2,3], [2,3,4]]
```

windowedSequence - Lazy windowed processing using a Sequence.

```
listOf(1,2,3,4).asSequence().windowed(2)  
// sequence of [1,2], [2,3], [3,4]
```

runningFold / runningReduce - Produce intermediate accumulated results instead of only the final result.

```
listOf(1,2,3).runningFold(0) { acc, n -> acc + n }  
// [0, 1, 3, 6]
```

High-Value Interview Patterns

FILTER -> MAP

```
students
  .filter { it.age > 22 }
  .map { it.name }
// [Priya, Meena]
```

FLATMAP -> DISTINCT

```
students
  .flatMap { it.courses }
  .distinct()
// unique courses across all students
```

FILTER -> MAP -> SORT

```
students
  .filter { it.marks >= 80 }
  .map { it.name }
  .sorted()
// [Arun, Meena, Priya]
```

GROUPBY -> MAPVALUES

```
students
  .groupBy { it.age / 10 }
  .mapValues { (_, group) -> group.size }
```

GROUPBY -> AGGREGATE

```
students
  .groupBy { it.age / 10 }
  .mapValues { (_, group) ->
    group.map { it.marks }.average()
  }
```

ASSOCIATEBY FOR LOOKUP

```
val studentsByName = students.associateBy { it.name }
val priya = studentsByName["Priya"]
```

PARTITION

```
val (passed, failed) =
  students.partition { it.marks >= 80 }
```

FOLD

```
val total = students.fold(0) { acc, student ->
  acc + student.marks
}
```

Fast Mental Rules

Which elements do I want?	-> filter
What should each element become?	-> map
Nested collections into one collection?	-> flatMap / flatten
Remove duplicates?	-> distinct / distinctBy
Find one element?	-> find / first / last
Does at least one match?	-> any
Do all match?	-> all
Does nobody match?	-> none
How many?	-> count
Total of a property?	-> sumOf
Highest / lowest object?	-> maxByOrNull / minByOrNull
Group by a key?	-> groupBy
Create a key -> object map?	-> associateBy
Create a key -> calculated value map?	-> associate / associateWith
Need both sides of a condition?	-> partition
Pair two lists?	-> zip
Split pairs?	-> unzip
Fixed-size groups?	-> chunked
Overlapping windows?	-> windowed
Build an accumulated result?	-> fold / reduce

Note: Not every operator is equally important in interviews. Master filter, map, flatMap, distinct, sortBy, find, any/all/none, count, sumOf, min/max, groupBy, associateBy, partition, zip, chunked, fold and reduce first.