Module 15 of 20

Streams

The pipeline model, every intermediate and terminal operation, the collectors — groupingBy, toMap, teeing — Optional as a return type, primitive streams, and parallel execution with its rules.

Streams are how modern Java expresses "take these, keep some, transform them, summarise" without a loop body. Most Java written since 2014 has them, most code review comments about Java mention them, and most interview questions about collections end with "now do it with a stream".

This module builds the model first — a lazy pipeline over a source, ending in one terminal — and then takes each layer in turn: the intermediate operations with the flatMap puzzle solved; the terminals with the three forms of reduce and their identity rule; the collectors, from toMap's duplicate-key trap to groupingBy with downstream collectors and teeing; Optional as the honest return type it was meant to be, with the anti-patterns spelled out; the primitive streams that avoid boxing; and parallel streams — how they split the work, what the lambdas must promise, and when they actually help.

By the end you write a pipeline as readily as a loop, choose between them deliberately, and can explain why a wrong reduce identity breaks in parallel — the question that separates people who use streams from people who understand them.

CONTENTS

1	Streams — pipelines over data	13 min
2	Intermediate operations — map, filter, flatMap and friends	14 min
3	Terminal operations — reduce, match, find, count	14 min
4	Collectors — toMap, joining, groupingBy, partitioningBy	16 min
5	Optional — a return type, not a null replacement	13 min
6	Primitive streams — IntStream, LongStream, DoubleStream	12 min
7	Parallel streams, ordering and performance	13 min
✓	Checkpoint — Streams	30 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Streams — pipelines over data

A stream is a pipeline: a **source** of elements, zero or more **intermediate** operations that transform or filter them lazily, and one **terminal** operation that produces a result. `names.stream().filter(n → n.startsWith("A")).map(String::toUpperCase).toList()` says *what* to compute — no index, no accumulator, no loop body — and that declarative shape is why streams took over Java collection code after Java 8. This lesson is the model: what a stream is and is not, the three kinds of operation, laziness, and the rules (single use, no side effects) that keep pipelines correct.

Anatomy of a pipeline

```
List<String> result = people.stream()                    // source
    .filter(p → p.age() ⩾ 18)                            // intermediate: keep
    .map(Person::name)                                   // intermediate: tran
    .sorted()                                            // intermediate: stat
    .limit(10)                                           // intermediate: shor
    .toList();                                           // terminal: produce
```

Source: a collection (`.stream()`), an array (`Arrays.stream`), literal values (`Stream.of`), a range (`IntStream.range`), a generator (`Stream.iterate` , `Stream.generate`), lines of a file (`Files.lines`), characters (`s.chars()`), a map's views (`map.entrySet().stream()`).

Intermediate operations return a new stream and do **nothing yet**. They are lazy: recorded, not executed.

Terminal operation: triggers execution, consumes the stream, and produces a value (`toList` , `count` , `collect`), a side effect (`forEach`), or an `Optional` (`findFirst` , `max`).

A pipeline with no terminal operation runs nothing — a classic bug: `list.stream().map(this::save);` saves nothing.

What a stream is not

A stream is **not a collection**: it stores no elements, it is computed on demand, and it can be consumed once. It is not a `List` with extra methods; it is a description of a computation over a source. `stream.toList()` produces a collection; the stream itself is gone afterwards.

```
Stream<String> s = names.stream();
s.count();
s.count();          // IllegalStateException: stream has already been operated
```

Make a new stream from the source for each use.

Laziness, and why it matters

```
Stream.of("a", "bb", "ccc")
    .filter(x -> { System.out.println("filter " + x); return x.length() > 1; })
    .map(x -> { System.out.println("map " + x); return x.toUpperCase(); })
    .findFirst();
// filter a / filter bb / map bb  — and stops: "ccc" is never examined
```

Elements flow through the pipeline **one at a time**, vertically, not stage by stage; and short-circuiting terminals (`findFirst` , `anyMatch` , `limit`) stop as soon as they can. This is what makes `Stream.iterate(1, x -> x * 2).limit(10)` — an infinite source — usable: only ten elements are ever produced.

Stateful intermediate ops (`sorted` , `distinct`) must see everything before emitting anything; `sorted()` on an infinite stream never returns.

Sources

```
list.stream(); set.stream(); map.entrySet().stream();
Arrays.stream(array); Arrays.stream(array, from, to);
Stream.of("a", "b"); Stream.empty(); Stream.ofNullable(maybeNull);
IntStream.range(0, 10); IntStream.rangeClosed(1, 10);
Stream.iterate(1, x -> x * 2);                  // infinite: 1, 2, 4, …
Stream.iterate(1, x -> x < 100, x -> x * 2);    // finite, Java 9: with a hasNe
Stream.generate(Math::random);                  // infinite
"hello".chars();                                // IntStream of code units
Files.lines(path);                              // lazy lines — close it (try-w
new Random().ints(5, 0, 100);
```

Intermediate versus terminal, at a glance

Intermediate (lazy, return a stream)	Terminal (eager, end the stream)
`filter`, `map`, `flatMap`, `mapToInt/Obj`	`forEach`, `forEachOrdered`
`sorted`, `distinct` (stateful)	`collect`, `toList`, `toArray`
`limit`, `skip`, `takeWhile`, `dropWhile`	`reduce`, `count`, `sum`, `min`, `max`, `average`
`peek` (debugging)	`anyMatch`, `allMatch`, `noneMatch` (short-circuit)
`boxed`, `parallel`, `sequential`, `unordered`	`findFirst`, `findAny` (short-circuit)

The next two lessons take each list in turn.

Rules for the lambdas inside

Stateless: a lambda must not depend on state that changes during the pipeline. `map(x -> counter++ + x)` is wrong — with parallel streams it is *unpredictable*.

Non-interfering: never modify the source collection while the stream runs (`ConcurrentModificationException`, or undefined results).

Side-effect free, except in `forEach` and `peek`. Collecting into an external list from `map` or `filter` is the classic anti-pattern; use `collect`.

Streams are designed so the runtime may reorder, parallelise or skip work; lambdas that assume sequential order or accumulate outside the pipeline break that contract.

Streams versus loops

```
// loop
int total = 0;
for (Order o : orders) if (o.isPaid()) total += o.amount();
// stream
int total = orders.stream().filter(Order::isPaid).mapToInt(Order::amount).sum(
```

Use a stream when the computation is naturally *filter/map/reduce* and the pipeline reads as a sentence; use a loop for complex control flow, early exits with several conditions, mutation of several variables, or checked exceptions. Streams are not faster for small data (allocation

and indirection cost more than the loop); they are clearer and compose better. In interviews, either is fine — say why you chose.

Debugging a pipeline

`peek(System.out::println)` between stages prints each element as it passes — remember it runs lazily, only for elements the terminal pulls. Breaking a long pipeline into named intermediate streams or extracting lambdas into methods makes stack traces readable (`lambda$main$0` tells you little).

Interview angle

"What is a stream?" A lazy pipeline over a source with intermediate and terminal operations — not a data structure.

"Why lazy?" Elements are processed one at a time and short-circuiting can stop early; infinite sources work.

"Can you reuse a stream?" No — one terminal operation; then it is closed.

"Difference between intermediate and terminal?" Intermediate returns a stream and does nothing until a terminal triggers execution.

"Should stream lambdas have side effects?" No — stateless and non-interfering; side effects only in `forEach`.

Key takeaways

Source → lazy intermediates → one terminal. No terminal, no work; one terminal per stream.

Elements flow one at a time; short-circuiting ops stop early; `sorted` / `distinct` need everything.

Sources: collections, arrays, `Stream.of`, ranges, `iterate` / `generate`, `chars`, `Files.lines`.

Lambdas must be stateless, non-interfering and side-effect free; collect results with terminals, not external lists.

EXERCISE 1

Watch the laziness

Read an integer `n` and `n` words. Build one pipeline over the list: a `filter` that keeps words of **3 or more letters** and prints `filter <word>` for every word it examines, then a `map` to upper case that prints `map <word>` for every word it receives, ending in `findFirst()`. Finally print `first=<the result>`, or `first=none` when nothing survived.

Because the pipeline is lazy and `findFirst` short-circuits, the trace stops the moment one word gets through.

Example: `4` then `ab cat dog emu` →

```
filter ab
filter cat
map cat
first=CAT
```

INPUT 1

```
4
ab cat dog emu
```

OUTPUT 1

```
filter ab
filter cat
map cat
first=CAT
```

INPUT 2

```
2
a b
```

OUTPUT 2

```
filter a
filter b
first=none
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        Optional<String> first = words.stream()
            .filter(w -> { System.out.println("filter " + w); return w.length() >= 3
            .map(w -> { System.out.println("map " + w); return w.toUpperCase(); })
            .findFirst();
        System.out.println("first=" + first.orElse("none"));
    }
}
```

EXERCISE 2

One terminal each

Read an integer `n` and `n` integers into a list. Print four facts, each computed by **its own stream** over the list (a stream cannot be reused): `count=<n>`, `sum=<sum>`, `max=<largest or none>`, and `evens=<the even numbers in input order joined by ,>` (print `-` when there are none).

Example: `5` then `3 8 1 8 2` →

```
count=5
sum=22
max=8
evens=8,8,2
```

INPUT 1

```
5
3 8 1 8 2
```

OUTPUT 1

```
count=5
sum=22
max=8
evens=8,8,2
```

INPUT 2

```
0
```

OUTPUT 2

```
count=0
sum=0
max=none
evens=-
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> nums = new ArrayList<>();
        for (int i = 0; i < n; i++) nums.add(in.nextInt());

        System.out.println("count=" + nums.stream().count());
        System.out.println("sum=" + nums.stream().mapToInt(Integer::intValue).sum())
        System.out.println("max=" + nums.stream().max(Integer::compare).map(String::
        String evens = nums.stream().filter(x -> x % 2 == 0).map(String::valueOf).co
        System.out.println("evens=" + (evens.isEmpty() ? "-" : evens));
    }
}
```

QUIZ

1. `list.stream().map(this::save);` — how many elements are saved?

A. All of them

B. None: there is no terminal operation, so nothing runs

C. The first one

D. It does not compile

2. Predict the output:

```
Stream<String> s = Stream.of("a", "b");
System.out.println(s.count());
System.out.println(s.count());
```

A. `2` then `2`

B. `2` then `IllegalStateException`

C. `2` then `0`

D. Compile error

3. Why does `Stream.iterate(1, x → x * 2).limit(10).toList()` terminate although the source is infinite?

A. `iterate` secretly produces only 10 elements

B. Elements are pulled one at a time and `limit` stops requesting after ten

C. The JIT detects the loop

D. It does not terminate

4. Which lambda breaks the rules for stream operations?

A. `filter(w → w.length() > 3)`

B. `map(w → { seen.add(w); return w; })` where `seen` is an outside list

C. `map(String::trim)`

D. `sorted(Comparator.reverseOrder())`

5. `sorted()` on an infinite stream…

A. Sorts the first 1000 elements

B. Never returns: a stateful operation must see every element first

C. Throws immediately

D. Is optimised away

ANSWER KEY · STREAMS — PIPELINES OVER DATA

1. B Intermediate operations are lazy. Without a terminal such as `forEach`, `toList` or `count`, the pipeline is never executed.

2. B A stream can be consumed once. The second terminal operation throws `IllegalStateException: stream has already been operated upon or closed`.

3. B Laziness plus short-circuiting: the terminal pulls, `limit` stops pulling after ten, so the generator is asked for exactly ten values.

4. B Side effects belong in `forEach` / `peek`, not `map`. With a parallel stream this races on the list; use `collect`.

5. B `sorted` and `distinct` are stateful barriers; they cannot emit anything until the source is exhausted.

LESSON 2 · 14 MIN

Intermediate operations — map, filter, flatMap and friends

Intermediate operations are the verbs of a pipeline: keep, transform, flatten, order, deduplicate, truncate. Each returns a new stream and runs lazily. Most are one-to-one and stateless; a few (`sorted`, `distinct`) must buffer; `flatMap` is the one people take longest to internalise. This lesson covers each operation with its signature, its typical use, and the mistake it invites.

`filter` — keep elements matching a predicate

```
orders.stream().filter(o → o.total() > 100).filter(Order::isPaid)     // two
words.stream().filter(Predicate.not(String::isBlank))
```

`Stream<T> filter(Predicate<? super T>)`. Never mutates; elements failing the test simply do not continue downstream.

`map` — transform each element

```
people.stream().map(Person::name)                  // Stream<Person> → Stream<S
strings.stream().map(String::length)               // Stream<String> → Stream<I
strings.stream().mapToInt(String::length)          // → IntStream: no boxing, g
ints.mapToObj(i → "#" + i)                         // IntStream → Stream<String
intStream.boxed()                                  // IntStream → Stream<Intege
```

`<R> Stream<R> map(Function<? super T, ? extends R>)`. One in, one out, same count. Use `mapToInt` / `mapToLong` / `mapToDouble` when the result is numeric and you want arithmetic terminals; `boxed()` to get back to objects for collectors.

`flatMap` — one-to-many, flattened

```
List<List<String>> nested = …;
nested.stream().flatMap(List::stream)                              // Stream<List<
sentences.stream().flatMap(s → Arrays.stream(s.split(" ")))        // words from s
orders.stream().flatMap(o → o.lines().stream())                    // all order li
people.stream().flatMap(p → p.phones().stream()).distinct()        // every phone,
Optional.of(x).stream()                                             // 0 or 1 elem
```

`<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>)`. The function returns a **stream** per element; `flatMap` concatenates them. Rule of thumb: when `map` would give you a `Stream<Stream<X>>` or `Stream<List<X>>` and you wanted `Stream<X>`, you needed `flatMap`. Java 16's `mapMulti` is an imperative alternative for the same job.

`sorted` — order (stateful)

```
.sorted()                                                          // natural order; e
.sorted(Comparator.comparing(Person::lastName).reversed())
```

Buffers the entire stream, sorts, then emits. O(n log n), stable. Anything after it (`limit`) still benefits from laziness downstream; nothing upstream of it does. Sorting then `limit(k)` is fine for small k; for huge inputs a `PriorityQueue` top-k is cheaper.

`distinct` — remove duplicates (stateful)

Uses `equals` / `hashCode`; keeps the first occurrence in encounter order. Buffers what it has seen. `Stream.of(3, 1, 3, 2).distinct()` → 3, 1, 2.

`limit` and `skip` — truncate

```
.limit(10)                      // at most 10 — short-circuits; makes infinite
.skip(5)                        // drop the first 5
.skip(page * size).limit(size)  // pagination
```

`limit` is what makes `Stream.iterate` / `generate` safe. Both respect encounter order.

`takeWhile` and `dropWhile` (Java 9)

```
Stream.iterate(1, x → x * 2).takeWhile(x → x < 100)      // 1 2 4 … 64 — sto
sortedNumbers.stream().dropWhile(x → x < 0)              // skip the leading
```

Condition-based prefixes rather than counts. On an unordered stream they take/drop an arbitrary subset satisfying the predicate.

`peek` — observe without changing

```
.peek(x → log.debug("saw {}", x))
```

For debugging. It is lazy like everything else — nothing prints without a terminal, and short-circuiting terminals see only some elements. Do not use it for real side effects.

Numeric conversions

`mapToInt`, `mapToLong`, `mapToDouble` produce primitive streams with `sum`, `average`, `min`, `max`, `summaryStatistics`; `asLongStream`, `asDoubleStream` widen; `boxed` and `mapToObj` return to objects. Lesson 6 covers primitive streams; note here that `map(String::length)` gives a `Stream<Integer>` with no `sum()`, while `mapToInt(String::length)` gives an `IntStream` that has one.

Ordering and `unordered`

Streams from lists have an *encounter order* that `map` / `filter` / `limit` / `sorted` preserve. `HashSet` sources have none. `unordered()` tells the runtime order does not matter, which lets `distinct` and `limit` run faster in parallel. Sequential pipelines rarely need it.

`parallel` / `sequential`

`stream().parallel()` (or `parallelStream()`) switches the pipeline to run across the common fork-join pool. It is an intermediate operation in form and a global switch in effect —

the whole pipeline is parallel. Lesson 6 covers when that helps and when it hurts.

Composition patterns

```
// top 3 longest distinct words, lower-cased
words.stream().map(String::toLowerCase).distinct()
     .sorted(Comparator.comparingInt(String::length).reversed().thenComparing(
     .limit(3).toList();

// all unique tags across posts
posts.stream().flatMap(p -> p.tags().stream()).distinct().sorted().toList();

// first even square greater than 1000
IntStream.iterate(1, i -> i + 1).map(i -> i * i).filter(sq -> sq % 2 == 0 && s
```

Read each pipeline top to bottom as a sentence; if a stage needs a comment, extract its lambda into a named method.

Mistakes

Mistake	Fix
`map(x -> list.add(x))` — side effect in `map`	Use `forEach` or `collect`
`map(List::stream)` giving `Stream<Stream<T>>`	`flatMap`
`sorted()` on an infinite stream	`limit` before sorting, or don't
`map(String::length).sum()` — no `sum` on `Stream<Integer>`	`mapToInt`
Reusing a stream after `filter` / `map` assigned to a variable and consumed	Build a new stream
`distinct()` on elements without `equals` / `hashCode`	Fix the element class or use a record

Interview angle

"`map` vs `flatMap`?" One-to-one vs one-to-many with flattening; `flatMap`'s function returns a stream.

"Which intermediate operations are stateful?" `sorted`, `distinct` (and `limit`/`skip` in a sense, as they track counts).

"Does `filter` run before or after `map`?" In pipeline order, element by element — put the cheaper/more selective filter first.

"What does `peek` do?" Applies a consumer to each element passing through, for debugging; lazy.

"Why `mapToInt`?" Primitive stream: no boxing and numeric terminals like `sum`.

Key takeaways

`filter` keeps, `map` transforms one-to-one, `flatMap` transforms one-to-many and flattens.

`sorted`/`distinct` buffer; `limit`/`skip`/`takeWhile`/`dropWhile` truncate; `limit` tames infinite sources.

`mapToInt`/`mapToObj`/`boxed` move between object and primitive streams.

Order filters early; no side effects in `map`; `peek` only for debugging.

EXERCISE 1

Flatten the playlists

Read an integer `n` and `n` lines of the form `name:song1,song2,...`. Using `flatMap`, `distinct` and `sorted`, print every distinct song across all playlists in natural order, one per line, then `total=<sum of all playlist lengths> distinct=<number of distinct songs>`.

Example: `2` then `ann:a,b,c` and `bob:c,d` →

```
a
b
c
d
total=5 distinct=4
```

INPUT 1

```
2
ann:a,b,c
bob:c,d
```

OUTPUT 1

```
a
b
c
d
total=5 distinct=4
```

INPUT 2

```
1
solo:zz
```

OUTPUT 2

```
zz
total=1 distinct=1
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        List<List<String>> playlists = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String line = in.nextLine();
            String[] songs = line.substring(line.indexOf(':') + 1).split(",");
            playlists.add(Arrays.asList(songs));
        }

        List<String> distinct = playlists.stream().flatMap(List::stream).distinct().
        distinct.forEach(System.out::println);
        int total = playlists.stream().mapToInt(List::size).sum();
        System.out.println("total=" + total + " distinct=" + distinct.size());
    }
}
```

EXERCISE 2

Slice and window

Read an integer `n`, `n` integers, then an integer `k`. Print four lines, each a space-joined list (print `-` for an empty one):

- `top=` the `k` largest values, largest first (`sorted` with `Comparator.reverseOrder()` then `limit`).
- `rest=` the input **after skipping** its first `k` elements, in input order.
- `prefix=` the leading run of **positive** values (`takeWhile`).

`tail=` everything from the first non-positive value on (`dropWhile`).

Example: `6` then `3 -1 4 1 -5 9` then `2` →

```
top=9 4
rest=4 1 -5 9
prefix=3
tail=-1 4 1 -5 9
```

INPUT 1

```
6
3 -1 4 1 -5 9
2
```

OUTPUT 1

```
top=9 4
rest=4 1 -5 9
prefix=3
tail=-1 4 1 -5 9
```

INPUT 2

```
3
5 6 7
5
```

OUTPUT 2

```
top=7 6 5
rest=-
prefix=5 6 7
tail=-
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    static String show(Stream<Integer> s) {
        String joined = s.map(String::valueOf).collect(Collectors.joining(" "));
        return joined.isEmpty() ? "-" : joined;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> nums = new ArrayList<>();
        for (int i = 0; i < n; i++) nums.add(in.nextInt());
        int k = in.nextInt();

        System.out.println("top=" + show(nums.stream().sorted(Comparator.reverseOrde
        System.out.println("rest=" + show(nums.stream().skip(k)));
        System.out.println("prefix=" + show(nums.stream().takeWhile(x -> x > 0)));
        System.out.println("tail=" + show(nums.stream().dropWhile(x -> x > 0)));
    }
}
```

QUIZ

1. `sentences.stream().map(s → Arrays.stream(s.split(" ")))` has the type…

A. `Stream<String>`

B. `Stream<Stream<String>>` — you wanted `flatMap`

C. `Stream<String[]>`

D. Compile error

2. Predict the output:

```
System.out.println(Stream.of(5, 1, 4, 1).distinct().sorted().skip(1).toList());
```

A. `[1, 4, 5]`

B. `[4, 5]`

C. `[1, 1, 4]`

D. `[5, 4]`

3. `takeWhile(x → x > 0)` on `3, 2, -1, 5` yields…

A. `3, 2, 5` — every positive

B. `3, 2` — it stops at the first failure

C. `5`

D. `3, 2, -1`

4. Which intermediate operations are stateful?

A. `map` and `filter`

B. `sorted` and `distinct`

C. `peek` and `limit`

D. `mapToInt` and `boxed`

5. `peek(System.out::println)` prints nothing in `List.of(1,2,3).stream().peek(...).count()` on Java 9+ because…

A. `peek` is deprecated

B. The count is known from the source, so the pipeline is skipped

C. `peek` runs only in parallel

D. `List.of` streams are empty

ANSWER KEY · INTERMEDIATE OPERATIONS — MAP, FILTER, FLATMAP AND FRIENDS

1. B `map` wraps whatever the function returns. A function returning a stream needs `flatMap` to splice the inner streams together.

2. B `distinct` → 5, 1, 4; `sorted` → 1, 4, 5; `skip(1)` drops the 1.

3. B `takeWhile` is a prefix operation, not a filter: once the predicate fails, nothing further is taken.

4. B They must buffer or remember elements; `limit` / `skip` are bounded and `map` / `filter` are per-element.

5. B An optimisation: with a sized source and no filtering, `count()` need not traverse. Never rely on `peek` for side effects.

LESSON 3 · 14 MIN

Terminal operations — reduce, match, find, count

A terminal operation is where the pipeline finally *does* something: it pulls elements through every lazy stage and produces a value, an `Optional`, a collection or a side effect. Choosing the right terminal is most of the skill — `reduce` when you fold to one value, `anyMatch` when a boolean is enough, `findFirst` when you want the first survivor and nothing more. This lesson covers every terminal except the collectors (next lesson), with the three forms of `reduce` explained properly, because `reduce` is where most stream code goes wrong.

The whole list

Group	Operations	Result
Iterate	`forEach`, `forEachOrdered`	nothing (side effect)
Fold	`reduce` (3 forms), `count`, `sum`, `min`, `max`, `average`	value or `Optional`
Search	`findFirst`, `findAny`	`Optional<T>`
Test	`anyMatch`, `allMatch`, `noneMatch`	`boolean`
Materialise	`toList`, `toArray`, `collect`, `iterator`	collection / array

Every one of them **consumes** the stream. Two terminals on the same stream is an `IllegalStateException`, not a second pass.

`forEach` and `forEachOrdered`

```
names.stream().forEach(System.out::println);
names.parallelStream().forEachOrdered(System.out::println);   // encounter ord
```

`forEach` on a parallel stream makes no promise about order; `forEachOrdered` does, at the cost of the parallelism. On a plain `List`, `list.forEach(...)` is shorter than

`list.stream().forEach(...)` and does the same thing. A `forEach` whose lambda fills an outside list is a `collect` in disguise — write the `collect`.

`reduce` — folding to one value

The one-argument form takes an **accumulator** and returns an `Optional`, because an empty stream has nothing to fold:

```
Optional<Integer> sum = Stream.of(1, 2, 3).reduce((a, b) → a + b);      // Op
Optional<Integer> none = Stream.<Integer>empty().reduce(Integer::sum);    //
```

The two-argument form takes an **identity** and always returns a value:

```
int sum = Stream.of(1, 2, 3).reduce(0, Integer::sum);                    // 
int product = IntStream.rangeClosed(1, 5).reduce(1, (a, b) → a * b);     // 1
String joined = Stream.of("a", "b", "c").reduce("", String::concat);      //
```

The identity must really be one: `identity op x == x` for every `x`. `0` for addition, `1` for multiplication, `""` for concatenation, `Integer.MIN_VALUE` for max. Pass `1` as the identity of a sum and the parallel version returns garbage — each chunk starts from 1 and the extras add up. That is not a theoretical bug; it is the standard "why does my parallel reduce give a different answer" question.

The three-argument form `reduce(identity, accumulator, combiner)` changes the result type: accumulate `T`s into a `U`, and combine two `U`s when chunks meet in parallel.

```
int totalLength = words.stream().reduce(0, (len, w) → len + w.length(), Integ
```

Sequentially the combiner is never called; it must still be correct, because `parallel()` is one method call away. If you find yourself reducing into a mutable container (a `StringBuilder`, an `ArrayList`), stop — that is what `collect` is for. `reduce` must not mutate; `collect` is built to.

`count`, `min`, `max`, `sum`, `average`

```
long n = words.stream().filter(w → w.length() > 3).count();
Optional<String> shortest = words.stream().min(Comparator.comparingInt(String
Optional<Person> oldest = people.stream().max(Comparator.comparingInt(Person::
int total = orders.stream().mapToInt(Order::amount).sum();                  //
OptionalDouble mean = IntStream.of(3, 4, 5).average();
```

`min` / `max` on an object stream **need a comparator** and return `Optional`. `sum` and `average` exist only on `IntStream` / `LongStream` / `DoubleStream`; on `Stream<Integer>` write `mapToInt(Integer::intValue).sum()` or `reduce(0, Integer::sum)`. Since Java 9, `count()` may skip the pipeline entirely when the size is known from the source (`List.of(1,2,3).stream().peek(...).count()` prints nothing) — another reason not to rely on `peek` for side effects.

`findFirst` and `findAny`

```
Optional<String> first = words.stream().filter(w → w.startsWith("q")).findFi
Optional<String> any = words.parallelStream().filter(w → w.startsWith("q")).
```

Both short-circuit: as soon as one element survives the filters, the pipeline stops. `findFirst` respects encounter order, which in parallel means waiting to be sure nothing earlier qualifies; `findAny` takes whatever finishes first and is the faster parallel choice when any survivor will do. Sequentially they behave identically.

`anyMatch`, `allMatch`, `noneMatch`

```
boolean hasAdult   = people.stream().anyMatch(p → p.age() ⩾ 18);
boolean allAdults  = people.stream().allMatch(p → p.age() ⩾ 18);
boolean noMinors   = people.stream().noneMatch(p → p.age() < 18);
```

All three short-circuit. Two edge cases are worth a second look: on an **empty stream**, `allMatch` is `true` and `anyMatch` is `false` (vacuous truth, exactly as in mathematics), and `allMatch(p) == noneMatch(p.negate())`. Use these instead of `filter(...).count() > 0`, which scans everything and reads worse.

`toList`, `toArray`, `iterator`

```
List<String> l1 = s.toList();                          // Java 16: unmodifiable,
List<String> l2 = s.collect(Collectors.toList());      // an ArrayList in practi
String[] a1 = s.toArray(String[]::new);                // typed array
Object[] a0 = s.toArray();                             // Object[] — usually not
Iterator<String> it = s.iterator();                    // for the rare loop that
```

`Stream.toList()` returns a list you cannot add to; `Collectors.toList()` returns one you (currently) can. When the rest of the code mutates the result, say so with `collect(Collectors.toCollection(ArrayList::new))` — that is the only one that *guarantees* mutability.

A worked example: three ways to the same answer

Longest word in a list, ties to the earliest:

```
Optional<String> a = words.stream().max(Comparator.comparingInt(String::length
Optional<String> b = words.stream().reduce((x, y) -> y.length() > x.length() 
String c = words.stream().sorted(Comparator.comparingInt(String::length).rever
```

`max` with `comparingInt` returns the **first** maximal element in encounter order (its implementation keeps the current best unless the new one is strictly greater), `b` spells that rule out, and `c` sorts everything just to take one — O(n log n) for an O(n) job. Prefer `a`; be able to explain why `c` is wasteful.

Interview angle

> *"What does `reduce` return with one argument?"* An `Optional`, because the stream might be empty.
>
> *"What must be true of a `reduce` identity?"* `identity op x == x` for all `x`; a wrong identity gives wrong parallel results.
>
> *"`allMatch` on an empty stream?"* `true`; `anyMatch` is `false`.
>
> *"`findFirst` versus `findAny`?"* Same sequentially; `findAny` is faster in parallel because it need not respect encounter order.

"How do you sum a `Stream<Integer>`?" `mapToInt(Integer::intValue).sum()` or `reduce(0, Integer::sum)`; `sum()` lives on the primitive streams.

Key takeaways

One terminal per stream; it pulls everything through the lazy stages.

`reduce`: one argument → `Optional`; two → identity + accumulator; three → change the type with a combiner. Identity must be a real identity; never mutate inside `reduce`.

`min` / `max` need a comparator on object streams; `sum` / `average` live on primitive streams.

`find*` and `*Match` short-circuit; `allMatch` of nothing is `true`.

`toList()` is unmodifiable; `Collectors.toCollection(ArrayList::new)` when you need to mutate.

EXERCISE 1

Fold the ledger

Read an integer `n` and `n` signed integers (deposits positive, withdrawals negative). Using terminal operations only — no loops — print:

`balance=` the sum via `reduce(0, Integer::sum)`

`largestDeposit=` the largest positive value, or `none`

`firstWithdrawal=` the first negative value in input order, or `none`

`allDeposits=` whether every value is positive (`allMatch`)

`deposits=` how many values are positive

Note what `allMatch` says about an empty ledger.

Example: `5` then `100 -30 45 -80 5` →

```
balance=40
largestDeposit=100
firstWithdrawal=-30
allDeposits=false
deposits=3
```

INPUT 1

```
5
100 -30 45 -80 5
```

OUTPUT 1

```
balance=40
largestDeposit=100
firstWithdrawal=-30
allDeposits=false
deposits=3
```

INPUT 2

```
0
```

OUTPUT 2

```
balance=0
largestDeposit=none
firstWithdrawal=none
allDeposits=true
deposits=0
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> tx = new ArrayList<>();
        for (int i = 0; i < n; i++) tx.add(in.nextInt());

        System.out.println("balance=" + tx.stream().reduce(0, Integer::sum));
        System.out.println("largestDeposit=" + tx.stream().filter(x -> x > 0).max(Ir
        System.out.println("firstWithdrawal=" + tx.stream().filter(x -> x < 0).findF
        System.out.println("allDeposits=" + tx.stream().allMatch(x -> x > 0));
        System.out.println("deposits=" + tx.stream().filter(x -> x > 0).count());
    }
}
```

EXERCISE 2

Longest word, three ways

Read an integer `n` and `n` words (`n ≥ 1`). Print:

`firstLongest=` the longest word, **first** among ties — use `max` with `Comparator.comparingInt(String::length)`.

`lastLongest=` the longest word, **last** among ties — use the one-argument `reduce` with a tie rule that prefers the later word.

`letters=` the total number of letters — use the three-argument `reduce(0, (len, w) → …, Integer::sum)`.

Example: `4` then `fig pear kiwi plum` →

```
firstLongest=pear
lastLongest=plum
letters=15
```

INPUT 1

```
4
fig pear kiwi plum
```

OUTPUT 1

```
firstLongest=pear
lastLongest=plum
letters=15
```

INPUT 2

```
1
solo
```

OUTPUT 2

```
firstLongest=solo
lastLongest=solo
letters=4
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        String first = words.stream().max(Comparator.comparingInt(String::length)).c
        String last = words.stream().reduce((a, b) -> b.length() >= a.length() ? b :
        int letters = words.stream().reduce(0, (len, w) -> len + w.length(), Integer
        System.out.println("firstLongest=" + first);
        System.out.println("lastLongest=" + last);
        System.out.println("letters=" + letters);
    }
}
```

QUIZ

1. `Stream.<Integer>empty().reduce(Integer::sum)` returns…

A. `0`

B. `Optional.empty()`

C. `null`

D. Throws `NoSuchElementException`

2. Predict the output:

```
System.out.println(Stream.of(1, 2, 3).parallel().reduce(1, Integer::sum));
```

A. Always `7`

B. `7` sequentially but possibly `8`, `9` or `10` in parallel — 1 is not the identity of addition

C. Always `6`

D. Compile error

3. `allMatch(p)` on an empty stream returns…

A. `false`

B. `true`

C. Throws

D. `null`

4. `findFirst` versus `findAny` in a sequential stream…

A. `findAny` is random

B. They return the same element; the difference matters only in parallel

C. `findFirst` scans everything

D. `findAny` returns the last

5. Why should `reduce` never accumulate into an `ArrayList`?

A. Lists cannot be reduced

B. `reduce` assumes immutable values; a mutated accumulator is shared between parallel chunks — that job is `collect`'s

C. It is slower

D. The compiler forbids it

ANSWER KEY · TERMINAL OPERATIONS — REDUCE, MATCH, FIND, COUNT

1. B The one-argument `reduce` has no identity to fall back on, so an empty stream yields an empty `Optional`.

2. B Each parallel chunk starts from the identity. A wrong identity is added once per chunk, so the answer depends on how the stream was split.

3. B Vacuous truth: there is no element for which the predicate fails. `anyMatch` on the same stream is `false`.

4. B Sequentially both stop at the first survivor. In parallel `findAny` may return any survivor and is faster.

5. B `collect` has a supplier and a combiner precisely so each chunk gets its own container.

LESSON 4 · 16 MIN

Collectors — toMap, joining, groupingBy, partitioningBy

`collect` is the terminal that builds something: a list, a set, a map, a string, a summary. Its argument is a `Collector` — a recipe with a supplier (make the container), an accumulator (add one element), a combiner (merge two containers, for parallel) and a finisher (convert at the end). You will write your own perhaps once a year; you will use the ones in `java.util.stream.Collectors` every day, and `groupingBy` with a downstream collector is the single most useful line in the whole streams API. This lesson is that toolbox.

The basics

```
import static java.util.stream.Collectors.*;

List<String> list    = s.collect(toList());
Set<String> set      = s.collect(toSet());                        // a HashSet: r
Set<String> ordered = s.collect(toCollection(TreeSet::new));    // any collecti
Deque<String> dq     = s.collect(toCollection(ArrayDeque::new));
```

Static-import `Collectors.*` in code that uses several; a pipeline reads far better as `groupingBy(..., counting())` than with the class name repeated four times.

`toMap` — and the two traps

```
Map<String, Integer> lengths = words.stream().collect(toMap(w → w, String::le
```

Trap one: **duplicate keys throw** `IllegalStateException: Duplicate key`. Fix with a merge function:

```
Map<String, Integer> counts = words.stream().collect(toMap(w → w, w → 1, Int
Map<Character, String> byInitial = words.stream().collect(toMap(w → w.charAt(
```

Trap two: the map is a `HashMap`, so **iteration order is unspecified**. Name the map type with the four-argument form:

```
Map<String, Integer> inOrder = words.stream().collect(toMap(w → w, String::le
TreeMap<String, Integer> sorted = words.stream().collect(toMap(w → w, String
```

Also: `toMap` rejects `null` values with a `NullPointerException` (it uses `Map.merge` internally). And `Function.identity()` is the idiomatic spelling of `w → w` for the key.

`joining` — strings without a `StringBuilder`

```
String csv    = names.stream().collect(joining(", "));                  // "ar
String pretty = names.stream().collect(joining(", ", "[", "]"));        // "[
String all    = names.stream().collect(joining());                       // "
```

Elements must be `CharSequence`s — `map(String::valueOf)` or `map(Object::toString)` first for anything else. `String.join(", ", list)` does the same without a stream when there is nothing to filter or map.

`groupingBy` — the one to master

`groupingBy(classifier)` puts elements into a `Map<K, List<T>>` keyed by what the classifier returns:

```
Map<Integer, List<String>> byLength = words.stream().collect(groupingBy(String
// {3=[ann, bob], 5=[carol]}
```

Add a **downstream collector** to say what to do with each group instead of listing it:

```
Map<Integer, Long> countByLength     = words.stream().collect(groupingBy(Strin
Map<String, Integer> totalByCity     = people.stream().collect(groupingBy(Pers
Map<String, Double> avgAgeByCity     = people.stream().collect(groupingBy(Pers
Map<String, Set<String>> namesByCity = people.stream().collect(groupingBy(Pers
Map<String, Optional<Person>> oldest = people.stream().collect(groupingBy(Pers
Map<String, List<String>> sortedNames = people.stream().collect(groupingBy(Per
```

The three-argument form names the map (`TreeMap::new` for sorted keys, `LinkedHashMap::new` for first-seen order). Groups can nest: `groupingBy(Person::country, groupingBy(Person::city, counting()))` gives `Map<String, Map<String, Long>>`.

The keys are whatever the classifier returns, so `groupingBy(w → w.charAt(0))` groups by first letter, `groupingBy(n → n % 3)` by remainder, and `groupingBy(p → p.age() / 10 * 10)` by decade. A `null` classifier result throws — map it to a sentinel first.

`partitioningBy` — exactly two groups

```
Map<Boolean, List<Integer>> evenOdd = nums.stream().collect(partitioningBy(n -
Map<Boolean, Long> passFail = scores.stream().collect(partitioningBy(s → s ⩾
```

A `partitioningBy` map **always has both keys**, `true` and `false`, even when one side is empty — `groupingBy` on a boolean would omit an empty group. That is the reason to prefer it for yes/no splits.

Downstream collectors worth knowing

Collector	Produces
`counting()`	`Long`
`summingInt/Long/Double(f)`	the sum
`averagingInt/Long/Double(f)`	`Double`
`minBy(cmp)`, `maxBy(cmp)`	`Optional<T>`
`mapping(f, downstream)`	apply `f` first, then collect
`filtering(pred, downstream)` (Java 9)	filter inside the group
`flatMapping(f, downstream)` (Java 9)	flatten inside the group
`reducing(identity, op)`	a fold per group
`summarizingInt(f)`	`IntSummaryStatistics`: count, sum, min, max, average in one pass
`collectingAndThen(downstream, finisher)`	post-process: `collectingAndThen(toList(), Collections::unmodifiableList)`
`teeing(c1, c2, merger)` (Java 12)	run two collectors over the same elements and merge

`teeing` deserves an example, because it removes the "two passes for two facts" problem:

```
record MinMax(int min, int max) {}
MinMax mm = nums.stream().collect(teeing(
    minBy(Integer::compare), maxBy(Integer::compare),
    (lo, hi) → new MinMax(lo.orElseThrow(), hi.orElseThrow())));
```

Writing a collector

```
Collector<String, StringBuilder, String> concat =
    Collector.of(StringBuilder::new, StringBuilder::append, StringBuilder::app
```

Supplier, accumulator, combiner, finisher. You will rarely need it — but knowing the shape explains why `collect` is allowed to mutate (the container is yours) while `reduce` is not.

`collect` versus `reduce`, settled

`reduce` folds immutable values: new value in, new value out. `collect` fills a mutable container. Reducing into an `ArrayList` "works" sequentially and breaks in parallel; collecting into one is exactly what the combiner is for. Rule: if the accumulator mutates its first argument, it is a collector.

Interview angle

"What happens on duplicate keys in `toMap`?" `IllegalStateException` — supply a merge function.

"`groupingBy` returns…?" `Map<K, List<T>>` by default; a downstream collector replaces the `List`.

"`partitioningBy` versus `groupingBy` on a boolean?" `partitioningBy` always has both keys.

"How do you count occurrences of each word?" `groupingBy(identity(), counting())` — or `toMap(w -> w, w -> 1, Integer::sum)`.

"Why does `collect` accept a mutable container when `reduce` does not?" A collector has a combiner that merges containers, so parallel chunks never share one.

Key takeaways

`toMap` throws on duplicates and gives a `HashMap`; add a merge function and a map supplier when you care.

`joining(sep, prefix, suffix)` for strings; `String.join` when there is no pipeline.

`groupingBy(classifier, mapFactory, downstream)` — memorise the shape; `counting`, `mapping`, `summingInt`, `maxBy` as downstreams.

`partitioningBy` for two-way splits; both keys always present.

`teeing` for two facts in one pass; `collectingAndThen` to finish; `Collector.of` when the library has no fit.

EXERCISE 1

Group the roster

Read an integer `n` and `n` lines `name city age`. Using `groupingBy` into a `TreeMap` with downstream collectors, print for every city in natural order:

`<city>: n=<people> avg=<average age, one decimal> names=<names in natural order joined by ,>`

Then, using `partitioningBy` (both keys are always present), print `adults=<count of age ≥ 18> minors=<count of age < 18>`.

Example: `3` then `ann oslo 30`, `bob oslo 20`, `cy rome 12` →

```
oslo: n=2 avg=25.0 names=ann,bob
rome: n=1 avg=12.0 names=cy
adults=2 minors=1
```

INPUT 1

```
3
ann oslo 30
bob oslo 20
cy rome 12
```

OUTPUT 1

```
oslo: n=2 avg=25.0 names=ann,bob
rome: n=1 avg=12.0 names=cy
adults=2 minors=1
```

INPUT 2

```
2
zed pune 40
amy pune 41
```

OUTPUT 2

```
pune: n=2 avg=40.5 names=amy,zed
adults=2 minors=0
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;

record Person(String name, String city, int age) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < n; i++) people.add(new Person(in.next(), in.next(), in.r

        Map<String, List<Person>> byCity = people.stream().collect(groupingBy(Person
        Map<String, Double> avg = people.stream().collect(groupingBy(Person::city, a
        Map<String, List<String>> names = people.stream().collect(groupingBy(Person
        for (String city : byCity.keySet()) {
            List<String> sorted = names.get(city).stream().sorted().toList();
            System.out.println(String.format(Locale.ROOT, "%s: n=%d avg=%.1f names=%
        }
        Map<Boolean, Long> split = people.stream().collect(partitioningBy(p -> p.age
        System.out.println("adults=" + split.get(true) + " minors=" + split.get(fals
    }
}
```

EXERCISE 2

toMap without the traps

Read an integer `n` and `n` words. Print:

One line per first letter, in letter order: `<letter>=<words starting with it, in input order, joined by +>` — a `toMap` whose **merge function** joins duplicates and whose map supplier is `TreeMap::new`.

`lengths=` the `toString` of a `LinkedHashMap` from each **distinct** word to its length, first occurrence wins (a four-argument `toMap`).

`csv=` the distinct words in natural order, `joining(", ", "[", "]")`.

Example: `4` then `apple bean avocado bean` →

```
a=apple+avocado
b=bean+bean
lengths={apple=5, bean=4, avocado=7}
csv=[apple, avocado, bean]
```

INPUT 1

```
4
apple bean avocado bean
```

OUTPUT 1

```
a=apple+avocado
b=bean+bean
lengths={apple=5, bean=4, avocado=7}
csv=[apple, avocado, bean]
```

INPUT 2

```
1
zoo
```

OUTPUT 2

```
z=zoo
lengths={zoo=3}
csv=[zoo]
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        Map<Character, String> byInitial = words.stream()
            .collect(toMap(w -> w.charAt(0), w -> w, (a, b) -> a + "+" + b, TreeMap
        byInitial.forEach((k, v) -> System.out.println(k + "=" + v));

        Map<String, Integer> lengths = words.stream()
            .collect(toMap(w -> w, String::length, (a, b) -> a, LinkedHashMap::new))
        System.out.println("lengths=" + lengths);

        System.out.println("csv=" + words.stream().distinct().sorted().collect(joini
    }
}
```

QUIZ

1. `words.stream().collect(toMap(w → w, String::length))` with a repeated word…

A. Keeps the first

B. Keeps the last

C. Throws `IllegalStateException: Duplicate key`

D. Stores a list

2. `groupingBy(String::length)` returns…

A. `Map<Integer, String>`

B. `Map<Integer, List<String>>`

C. `Map<String, Integer>`

D. `List<List<String>>`

3. Predict the output:

```
System.out.println(Stream.of(2, 4).collect(partitioningBy(x → x % 2 == 1)));
```

A. `{false=[2, 4]}`

B. `{false=[2, 4], true=[]}`

C. `{true=[]}`

D. `{}`

4. Which collector computes two facts over the same elements in one pass?

A. `collectingAndThen`

B. `teeing`

C. `reducing`

D. `summarizingInt` only

5. `joining(", ")` requires the stream elements to be…

A. `Integer`

B. `CharSequence` (map with `String::valueOf` first otherwise)

C. `Object`

D. `Comparable`

6. Why is mutating a container inside `collect` fine but inside `reduce` wrong?

A. It is not fine in either

B. A collector has a supplier and a combiner, so each parallel chunk gets its own container and they are merged

C. `collect` is always sequential

D. `reduce` cannot see lists

ANSWER KEY · COLLECTORS — TOMAP, JOINING, GROUPINGBY, PARTITIONINGBY

1. C Supply a merge function — `(a, b) → a` — or count with `Integer::sum`.

2. B The default downstream is `toList()`; replace it with `counting()`, `mapping(...)`, etc.

3. B `partitioningBy` always has both keys, even when one side is empty; `groupingBy` on the same boolean would omit `true`.

4. B `teeing(c1, c2, merger)` (Java 12) runs both collectors and merges their results.

5. B `joining` concatenates `CharSequence`s; other types need a `map` first.

6. B The `Collector` contract is designed around mutable accumulation; `reduce` is designed around immutable values.

LESSON 5 · 13 MIN

Optional — a return type, not a null replacement

`Optional<T>` is a box that holds one value or nothing. It arrived with streams because `findFirst`, `max` and the one-argument `reduce` needed an honest way to say "there may be no answer", and it has since become the standard return type for lookups that can fail. It is **not** a general replacement for `null`, and most of the ugliness in Optional-heavy code comes from using it where a plain value, a plain `null` or an exception was right. This lesson covers the API, the idioms, and the six anti-patterns interviewers watch for.

Creating one

```
Optional<String> some = Optional.of("x");            // throws NullPointerExce
Optional<String> maybe = Optional.ofNullable(value);  // empty if value is nu
Optional<String> none = Optional.empty();
```

`of` says "this is definitely a value"; `ofNullable` is the bridge from a nullable API. Reaching for `ofNullable` everywhere hides bugs — if a value must not be null, let `of` fail early.

Reading it: the ladder from worst to best

```
Optional<User> u = repo.findByEmail(email);

if (u.isPresent()) { use(u.get()); }                // 1. the null check wear
u.ifPresent(this::use);                             // 2. do something if pre
u.ifPresentOrElse(this::use, this::signUpPrompt);   // 3. Java 9: either brar
User user = u.orElse(User.GUEST);                   // 4. a default value
User user = u.orElseGet(User::guest);               // 5. a default computed
User user = u.orElseThrow();                        // 6. Java 10: NoSuchEler
User user = u.orElseThrow(() -> new NotFoundException(email));   // 7. your ex
```

`get()` on an empty Optional throws `NoSuchElementException` — the very failure the type was meant to prevent, so `get()` is effectively deprecated in style guides; use `orElseThrow()`, which says the same thing honestly.

`orElse` versus `orElseGet`: `orElse(expensive())` evaluates `expensive()` *every time*, present or not, because arguments are evaluated before the call. `orElseGet(() → expensive())` runs it only when empty. For a constant, `orElse`; for a method call, `orElseGet`.

Transforming: `map`, `flatMap`, `filter`, `or`

```
Optional<String> city = repo.findByEmail(email)
    .map(User::address)              // Optional<Address>, or empty
    .map(Address::city)              // Optional<String>
    .filter(c → !c.isBlank());       // empty if the predicate fails

Optional<String> zip = repo.findByEmail(email)
    .flatMap(User::maybeAddress)     // maybeAddress returns Optional<Address
    .map(Address::zip);

Optional<Config> cfg = local().or(() → remote());   // Java 9: first non-empt
```

`map` when the function returns a plain value; `flatMap` when it returns an `Optional`. This chain is the payoff: five null checks become one readable sentence, and the "empty" travels through automatically.

Optional and streams

```
Stream<String> zeroOrOne = opt.stream();                    // Java 9
List<String> present = optionals.stream().flatMap(Optional::stream).toList();
```

`Optional.stream()` turns a list of Optionals into a stream of the present values in one `flatMap` — cleaner than `filter(Optional::isPresent).map(Optional::get)`.

The primitive variants

`OptionalInt`, `OptionalLong`, `OptionalDouble` come back from `IntStream.max()`, `average()` and friends. They have `getAsInt()`, `orElse`, `orElseGet`, `orElseThrow`, `ifPresent` — but **no `map` or `filter`**; convert with `boxed()` earlier in the pipeline if you need to chain.

Where Optional belongs — and where it does not

Use it as the return type of a method that may legitimately have no result: `findById`, `max`, `parseOrEmpty`, a cache lookup.

Do not use it:

As a field. It is not `Serializable`, doubles the object count, and the class's own methods know how to handle a missing value. A nullable field with a documented `Optional`-returning getter is the idiom.

As a method parameter. `void f(Optional<String> name)` forces every caller to wrap; overload or accept `null`/a default instead.

Wrapping a collection. An empty list already means "nothing"; `Optional<List<T>>` has two ways to be empty.

`isPresent()` + `get()`. That is the null check with more characters.

`Optional.of(x).orElse(y)` on a value you already have. `x ≠ null ? x : y` — or `Objects.requireNonNullElse(x, y)`.

Storing `Optional` in a collection or map value. Use the absence of the key.

Optional is a *return-type* tool. When you find one anywhere else, ask what it is protecting against.

Optional and `equals`

`Optional.of("a").equals(Optional.of("a"))` is `true` — equality is by content, and two empties are equal. But `Optional` is a value-based class: never synchronise on one and never compare with `==`.

Interview angle

"What problem does Optional solve?" An honest return type for "maybe no result", so the caller cannot forget the empty case the way they forget `null`.

"`orElse` versus `orElseGet`?" The argument of `orElse` is always evaluated; `orElseGet`'s supplier only when empty.

"`map` versus `flatMap` on Optional?" `flatMap` when the function itself returns an `Optional`.

"Should a field be an Optional?" No — not serialisable, wasteful, and the class controls access anyway; return one from the getter.

"What does `get()` do on empty?" Throws `NoSuchElementException`; prefer `orElseThrow()`.

Key takeaways

`of` (never null), `ofNullable` (maybe null), `empty()`.

Read with `ifPresent`, `orElse`, `orElseGet`, `orElseThrow` — never `isPresent()` + `get()`.

Chain with `map` / `flatMap` / `filter` / `or`; the empty propagates for you.

`Optional::stream` in a `flatMap` drops the empties from a collection of Optionals.

A return type, not a field, parameter, collection wrapper or null substitute.

EXERCISE 1

Lookup ladder

Read an integer `n` and `n` lines `key value` into a map, then an integer `m` and `m` query keys. For each query print `<key> → <value>` when the key exists and `<key> → (none)` otherwise, going through `Optional.ofNullable(map.get(key))` — no `if`, no `containsKey`. Compute the default with `orElseGet` and a supplier that **increments a counter** every time it runs; after the queries print `defaults=<counter>`. The counter proves that `orElseGet` is evaluated only for the misses.

Example: `2` then `a 1`, `b 2`, then `3` then `a x b` →

```
a → 1
x → (none)
b → 2
defaults=1
```

INPUT 1

```
2
a 1
b 2
3
a x b
```

OUTPUT 1

```
a → 1
x → (none)
b → 2
defaults=1
```

INPUT 2

```
1
k v
2
q r
```

OUTPUT 2

```
q → (none)
r → (none)
defaults=2
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static int defaults = 0;

    static String fallback() {
        defaults++;
        return "(none)";
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, String> map = new HashMap<>();
        for (int i = 0; i < n; i++) map.put(in.next(), in.next());
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            String key = in.next();
            String shown = Optional.ofNullable(map.get(key)).orElseGet(Main::fallbac
            System.out.println(key + " → " + shown);
        }
        System.out.println("defaults=" + defaults);
    }
}
```

EXERCISE 2

Chain through the empties

Read an integer `n` and `n` records `name city zip`, where `-` means the field is missing. For each record print `<name>: <result>`, where the result comes from **one Optional chain** and no `if`:

start from `Optional.ofNullable(city)` (treat `-` as null), `flatMap` to the zip (again `-` is empty), `filter` to zips of exactly **5 digits**, `map` to `"<CITY in upper case> <zip>"`, and `orElse("unknown")`.

Finally print `located=<how many records produced a value>` — count them with `Optional::stream` in a `flatMap`, not with a counter.

Example: `3` then `ann oslo 01234`, `bob - 99999`, `cy rome 12` →

```
ann: OSLO 01234
bob: unknown
cy: unknown
located=1
```

INPUT 1

```
3
ann oslo 01234
bob - 99999
cy rome 12
```

OUTPUT 1

```
ann: OSLO 01234
bob: unknown
cy: unknown
located=1
```

INPUT 2

```
2
x pune 41100
y pune -
```

OUTPUT 2

```
x: PUNE 41100
y: unknown
located=1
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

record Rec(String name, String city, String zip) { }

public class Main {
    static String orNull(String s) { return s.equals("-") ? null : s; }

    static Optional<String> locate(Rec r) {
        return Optional.ofNullable(r.city())
            .flatMap(city → Optional.ofNullable(r.zip()).map(zip → new String[] {
            .filter(pair → pair[1].matches("\\d{5}"))
            .map(pair → pair[0].toUpperCase() + " " + pair[1]);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Rec> recs = new ArrayList<>();
        for (int i = 0; i < n; i++) recs.add(new Rec(in.next(), orNull(in.next()), c
        for (Rec r : recs) System.out.println(r.name() + ": " + locate(r).orElse("ur
        long located = recs.stream().map(Main::locate).flatMap(Optional::stream).cou
        System.out.println("located=" + located);
    }
}
```

QUIZ

1. `Optional.of(null)` …

A. Returns an empty Optional

B. Throws `NullPointerException`

C. Returns `Optional[null]`

D. Does not compile

2. `opt.orElse(load())` versus `opt.orElseGet(() → load())`: when is `load()` called?

A. Both: only when empty

B. `orElse`: always; `orElseGet`: only when empty

C. Both: always

D. `orElse`: never

3. Which is the correct idiom to convert a `List<Optional<T>>` into the present values?

A. `.map(Optional::get)`

B. `.flatMap(Optional::stream)`

C. `.filter(Optional::isEmpty)`

D. `.map(Optional::orElseThrow)`

4. Where does `Optional` **not** belong?

A. As the return type of `findById`

B. As a class field or a method parameter

C. As the result of `Stream.max`

D. As the return type of a cache lookup

5. Predict the output:

```
Optional<String> o = Optional.of("hi").filter(s -> s.length() > 5).map(String::
System.out.println(o.isPresent() + " " + o.orElse("-"));
```

A. `true HI`

B. `false -`

C. `true hi`

D. Throws

ANSWER KEY · OPTIONAL — A RETURN TYPE, NOT A NULL REPLACEMENT

1. B `of` asserts the value is present; `ofNullable` is the bridge from a nullable value.

2. B Method arguments are evaluated before the call. `orElse` receives an already-computed value.

3. B `Optional.stream()` yields zero or one element, so `flatMap` drops the empties without a `get`.

4. B It is not serialisable, forces wrapping on every caller, and the class already controls access to its own fields.

5. B The filter fails, the Optional becomes empty, `map` is skipped, and `orElse` supplies the dash.

LESSON 6 · 12 MIN

Primitive streams — IntStream, LongStream, DoubleStream

A `Stream<Integer>` boxes every element: each `int` becomes an object on the heap, every arithmetic step unboxes and re-boxes, and `sum()` does not even exist on it. The three primitive specialisations — `IntStream`, `LongStream`, `DoubleStream` — fix that. They carry `int`s, `long`s and `double`s directly, add the numeric terminals (`sum`, `average`, `summaryStatistics`), and make the range loop `for (int i = 0; i < n; i++)` expressible as a pipeline. This lesson covers them, the conversions between the worlds, and the two places (`chars()` and `range`) where they show up whether you asked or not.

Sources

```
IntStream.range(0, 5);           // 0 1 2 3 4
IntStream.rangeClosed(1, 5);     // 1 2 3 4 5
IntStream.of(3, 1, 4);
IntStream.iterate(1, x → x * 2).limit(10);          // powers of two
IntStream.iterate(1, x → x ⩽ 1000, x → x * 2);     // finite (Java 9)
IntStream.generate(() → 7).limit(3);
Arrays.stream(new int[]{1, 2, 3});                  // int[] → IntStream (an
"hello".chars();                                     // IntStream of UTF-16 c
"hello".codePoints();                                // IntStream of code po
new Random().ints(5, 0, 100);                        // five ints in [0, 100)
LongStream.rangeClosed(1, 20);                       // for factorials: long,
```

There is no `range` for `DoubleStream`; build one with `IntStream.range(...).mapToDouble(i → i * 0.1)`.

Numeric terminals — the reason they exist

```
int sum = IntStream.rangeClosed(1, 100).sum();                  // 5050
OptionalInt max = IntStream.of(3, 9, 2).max();                  // OptionalInt
OptionalDouble avg = IntStream.of(1, 2, 4).average();           // 2.333…
long count = IntStream.range(0, 10).filter(i -> i % 3 == 0).count();

IntSummaryStatistics st = IntStream.of(4, 8, 15, 16, 23, 42).summaryStatistics
st.getCount(); st.getSum(); st.getMin(); st.getMax(); st.getAverage();   // a
```

`max` / `min` need no comparator (the natural order of numbers is built in); `average` returns `OptionalDouble` (empty for no elements); `sum` of an empty stream is `0`. **Overflow is silent**: `IntStream.rangeClosed(1, 20).reduce(1, (a, b) -> a * b)` wraps around — use `LongStream` or `asLongStream()` before multiplying, and `Math.multiplyExact` if you would rather fail than be wrong.

Moving between the worlds

From → to	Method
`Stream<T>` → `IntStream`	`mapToInt(ToIntFunction)` — `mapToInt(String::length)`
`IntStream` → `Stream<Integer>`	`boxed()`
`IntStream` → `Stream<R>`	`mapToObj(IntFunction)` — `mapToObj(i -> "#" + i)`
`IntStream` → `LongStream` / `DoubleStream`	`asLongStream()`, `asDoubleStream()`, or `mapToLong` / `mapToDouble`
`Stream<Integer>` → `int[]`	`mapToInt(Integer::intValue).toArray()`
`IntStream` → `List<Integer>`	`boxed().toList()`

`IntStream.toArray()` gives an `int[]`; `Stream<Integer>.toArray()` gives an `Object[]` — a favourite source of surprise. Collectors work only on object streams: `boxed()` before `collect(groupingBy(...))`.

Working with characters

```
long vowels = s.chars().filter(c → "aeiou".indexOf(c) ⩾ 0).count();
String upper = s.chars().mapToObj(c → String.valueOf((char) Character.toUpper
boolean allDigits = !s.isEmpty() && s.chars().allMatch(Character::isDigit);
Map<Character, Long> freq = s.chars().mapToObj(c → (char) c).collect(groupin
String reversed = new StringBuilder(s).reverse().toString();   // not a strea
```

`chars()` yields `int`s, so the `char` cast is on you; `mapToObj(c → (char) c)` gets you a `Stream<Character>` for collectors. For text that may contain characters outside the basic plane (emoji, some CJK), `codePoints()` and `Character.toChars` are the correct pair.

Index-driven pipelines

Streams have no index. When you need one, generate it:

```
IntStream.range(0, list.size()).filter(i → i % 2 == 0).mapToObj(list::get).t
IntStream.range(0, a.length).mapToObj(i → a[i] + "@" + i).forEach(System.out
IntStream.range(0, n).map(i → n - 1 - i);
```

If the loop body needs *two* indices or mutates as it goes, the `for` loop is the right tool; a stream contorted around an index is worse than the loop it replaced.

Boxing costs, measured honestly

On a million integers, `Stream<Integer>` summing via `reduce(0, Integer::sum)` allocates a million `Integer`s (well, most of them — the small ones come from the cache) and runs several times slower than `IntStream.sum()`. It rarely matters at interview scale; it always matters in a hot loop over an `int[]`. The habit is cheap: reach for `mapToInt` the moment the pipeline turns numeric, and stay unboxed until a collector needs objects.

Interview angle

> *"Why do primitive streams exist?"* To avoid boxing and to provide `sum` / `average` / `summaryStatistics`, which `Stream<Integer>` lacks.
>
> *"`Arrays.stream(int[])` returns…?"* `IntStream`; `Arrays.stream(Integer[])` returns `Stream<Integer>`.

"How do you get from `IntStream` to a `List<Integer>`?" `boxed().toList()` (or `collect(toList())`).

"What does `String.chars()` return?" An `IntStream` of UTF-16 code units — cast to `char`, or use `codePoints()` for supplementary characters.

"What does `average()` return on an empty stream?" `OptionalDouble.empty()`; `sum()` returns `0`.

Key takeaways

`IntStream` / `LongStream` / `DoubleStream` hold primitives: no boxing, numeric terminals built in.

`range` / `rangeClosed` replace index loops; `iterate` with a predicate for other progressions.

Convert with `mapToInt`, `boxed`, `mapToObj`, `asLongStream`; collectors need objects.

`sum` of nothing is 0; `average` / `max` / `min` return the `Optional*` types; overflow is silent — go `long` early.

`chars()` is an `IntStream`; cast to `char` or use `codePoints()`.

EXERCISE 1

Statistics in one pass

Read an integer `n` (`n ≥ 1`), `n` integers, then an integer `k` (`0 ≤ k ≤ 20`). Using `IntStream` / `LongStream`:

`stats=` count, sum, min, max and average (two decimals) from **one** `summaryStatistics()` call, formatted `count=<c> sum=<s> min=<lo> max=<hi> avg=<a>`.

`evenSquares=` the sum of the squares of the even values.

`factorial=` `k!` computed with `LongStream.rangeClosed(1, k)` and `reduce` — `0!` is `1`.

Example: `4` then `3 8 1 6` then `5` →

```
stats=count=4 sum=18 min=1 max=8 avg=4.50
evenSquares=100
factorial=120
```

INPUT 1

```
4
3 8 1 6
5
```

OUTPUT 1

```
stats=count=4 sum=18 min=1 max=8
avg=4.50
evenSquares=100
factorial=120
```

INPUT 2

```
1
7
0
```

OUTPUT 2

```
stats=count=1 sum=7 min=7 max=7 avg=7.00
evenSquares=0
factorial=1
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int k = in.nextInt();

        IntSummaryStatistics st = Arrays.stream(a).summaryStatistics();
        System.out.println(String.format(Locale.ROOT, "stats=count=%d sum=%d min=%d
            st.getCount(), st.getSum(), st.getMin(), st.getMax(), st.getAverage()));
        long evenSquares = Arrays.stream(a).filter(x -> x % 2 == 0).mapToLong(x -> (
        System.out.println("evenSquares=" + evenSquares);
        long factorial = LongStream.rangeClosed(1, k).reduce(1, (x, y) -> x * y);
        System.out.println("factorial=" + factorial);
    }
}
```

EXERCISE 2

Character census

Read one line of text. Using `chars()` pipelines (no index loops), print:

`letters=` how many characters are letters, `digits=` how many are digits, `vowels=` how many are vowels (`aeiou`, either case).

`upper=` the text in upper case, rebuilt from the stream with `mapToObj` and `joining`.

`freq=` the `toString` of a `TreeMap<Character, Long>` counting each **lower-cased letter**.

Example: `Hi there, 42!` →

```
letters=7 digits=2 vowels=3
upper=HI THERE, 42!
freq={e=2, h=2, i=1, r=1, t=1}
```

INPUT 1

```
Hi there, 42!
```

OUTPUT 1

```
letters=7 digits=2 vowels=3
upper=HI THERE, 42!
freq={e=2, h=2, i=1, r=1, t=1}
```

INPUT 2

```
2024
```

OUTPUT 2

```
letters=0 digits=4 vowels=0
upper=2024
freq={}
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s = in.hasNextLine() ? in.nextLine() : "";

        long letters = s.chars().filter(Character::isLetter).count();
        long digits = s.chars().filter(Character::isDigit).count();
        long vowels = s.chars().filter(c -> "aeiouAEIOU".indexOf(c) >= 0).count();
        System.out.println("letters=" + letters + " digits=" + digits + " vowels=" +

        String upper = s.chars().mapToObj(c -> String.valueOf((char) Character.toUpp
        System.out.println("upper=" + upper);

        Map<Character, Long> freq = s.chars().filter(Character::isLetter)
            .mapToObj(c -> (char) Character.toLowerCase(c))
            .collect(groupingBy(c -> c, TreeMap::new, counting()));
        System.out.println("freq=" + freq);
    }
}
```

QUIZ

1. `Arrays.stream(new int[]{1, 2, 3})` has the type…

A. `Stream<Integer>`

B. `IntStream`

C. `Stream<int[]>`

D. `List<Integer>`

2. `IntStream.empty().average()` and `IntStream.empty().sum()` return…

A. `0.0` and `0`

B. `OptionalDouble.empty()` and `0`

C. Both throw

D. `NaN` and `0`

3. Predict the output:

```
System.out.println("abc".chars().map(c -> c + 1).mapToObj(c -> String.valueOf((
```

A. `abc`

B. `bcd`

C. `98 99 100`

D. Compile error

4. To use `groupingBy` on an `IntStream` you must first call…

A. `sum()`

B. `boxed()` (or `mapToObj`)

C. `parallel()`

D. `toArray()`

5. `IntStream.rangeClosed(1, 20).reduce(1, (a, b) → a * b)` gives a wrong answer because…

A. `reduce` needs an Optional

B. `20!` overflows `int` silently — use `LongStream`

C. `rangeClosed` excludes 20

D. The identity should be 0

ANSWER KEY · PRIMITIVE STREAMS — INTSTREAM, LONGSTREAM, DOUBLESTREAM

1. B An `int[]` gives an `IntStream`; an `Integer[]` would give `Stream<Integer>`.

2. B The average of nothing is undefined, hence the Optional; the sum of nothing is zero.

3. B `chars()` yields code units, `+1` shifts each, and the cast back to `char` rebuilds the text.

4. B Collectors operate on object streams; the primitive streams have no `collect(Collector)`.

5. B Integer overflow wraps without warning. `LongStream` holds 20! (about 2.4×10^{18}); beyond 20! you need `BigInteger`.

LESSON 7 · 13 MIN

Parallel streams, ordering and performance

`.parallel()` is one method call that splits a pipeline across every core in the machine. It is also the easiest way in Java to make a correct program wrong, or a fast program slower, so it deserves a lesson of its own: how the work is split, what the pipeline must promise in return, when it actually pays, and how streams compare to loops when the clock is running.

How a parallel stream runs

```
long evens = IntStream.range(0, 10_000_000).parallel().filter(i → i % 2 == 0)
```

The source is split with a `Spliterator` into chunks; the chunks are processed as tasks on the **common** `ForkJoinPool` (size = cores − 1, plus the calling thread); partial results are merged with the combiner. Every core busy, one line of code. `parallelStream()` on a collection and `parallel()` on a stream do the same thing; `sequential()` switches back — the *last* call wins for the whole pipeline.

Splitting quality depends on the source. `ArrayList`, arrays and `IntStream.range` split perfectly (known size, random access). `HashSet` and `TreeMap` split adequately. `LinkedList`, `Stream.iterate` and `BufferedReader.lines()` split badly — the spliterator has to walk to find the middle — and often run *slower* in parallel.

What the pipeline must promise

Parallel execution is only correct when the lambdas keep the rules from the first lesson, and two more:

No shared mutable state. `list.parallelStream().forEach(x → results.add(x))` corrupts the `ArrayList` — lost elements, `ArrayIndexOutOfBoundsException`, or silent garbage. Use `collect`, whose combiner exists for exactly this.

`reduce` **identity and associativity.** The identity must be a real identity and the operation associative (`(a op b) op c == a op (b op c)`); subtraction and string prefixing are not, and the parallel answer differs run to run.

Stateless lambdas. A counter captured from outside, a `Random` shared between threads, a `SimpleDateFormat` — all break.

No blocking. The common pool is shared by every parallel stream *and* `CompletableFuture` in the JVM. A lambda that waits on I/O parks a pool thread and starves everyone else; parallel streams are for CPU work.

Ordering

A stream from a `List` has an **encounter order**, and most operations respect it even in parallel: `toList`, `forEachOrdered`, `findFirst`, `limit`, `skip`, `sorted`. Respecting order costs — `limit(10)` in parallel must buffer to be sure it kept the *first* ten. Three ways to pay less:

```
list.parallelStream().forEach(...)                          // no order promise
list.parallelStream().unordered().distinct()                 // distinct withou
list.parallelStream().findAny()                              // any survivor, r
```

A `HashSet` source has no encounter order to begin with, so ordering is free (and meaningless) there.

When parallel actually helps

The rule of thumb from the JDK's own authors: `N × Q > 10 000`, where `N` is the element count and `Q` the cost per element in "simple operations". Ten million cheap integers, or ten thousand expensive image transforms, might benefit; a hundred strings never will. Even then:

The work must be **CPU-bound** and **independent** per element.

The source must **split well** (array, `ArrayList`, range).

The machine must have **idle cores** — on a busy server the pool is already saturated.

The terminal must **merge cheaply** — `count`, `sum`, `toList` do; `groupingBy` into a `HashMap` merges maps and can cost more than it saved (`groupingByConcurrent` helps when order does not matter).

Measure before and after. If you cannot measure it, do not parallelise it.

Streams versus loops: performance, not just taste

For small collections a stream costs more than the loop: the pipeline objects, the lambdas, the iterator, and a cold JIT. On a million-element `int[]`, `IntStream.sum()` and a `for` loop are within noise once warmed up — the JIT inlines the lambdas — while `Stream<Integer>` with boxing is several times slower. The rules that follow:

Boxing is the main cost, not the stream. `mapToInt` early.

`sorted()` is O(n log n) and buffers everything; sorting to take one element is the classic waste — `min` / `max` instead.

`distinct()` builds a `HashSet`; on a sorted source, hand-written adjacent-duplicate removal is cheaper.

Multiple passes over the same source (`count()` then `sum()` then `max()`) are three iterations; `summaryStatistics()` or `teeing` is one.

In interviews the difference rarely matters; the ability to *explain* it does. "A stream here is the same complexity; I would switch to a loop if profiling showed the pipeline overhead" is the right sentence.

Debugging and safety checklist

Before shipping `.parallel()`:

Every lambda stateless and side-effect free? Results collected with `collect`, not an outside list?

`reduce` with a true identity and an associative operation?

Nothing blocking (no I/O, no locks, no `Thread.sleep`) inside the pipeline?

Source splits well? Enough elements and per-element work to matter?

Measured, on data of production size, with the JIT warmed?

If any answer is "not sure", stay sequential. A sequential stream is never the bug.

Interview angle

"How does a parallel stream execute?" Spliterator chunks the source; the common `ForkJoinPool` runs the chunks; combiners merge.

"When does `parallel()` make code slower?" Small data, cheap operations, badly-splitting sources (`LinkedList`, `iterate`), order-preserving operations, expensive merges, blocking lambdas.

"What goes wrong with `forEach(list::add)` in parallel?" A race on an unsynchronised `ArrayList`; use `collect`.

"Why must a `reduce` be associative?" Chunks are combined in an unpredictable tree; only associativity guarantees the same result.

"Which pool runs parallel streams?" The common `ForkJoinPool`, shared JVM-wide — a blocking lambda starves other users.

Key takeaways

`parallel()` splits the source and runs on the common ForkJoinPool; the last `parallel` / `sequential` call wins.

Correctness needs stateless, non-interfering, side-effect-free lambdas and associative reductions with true identities; collect instead of mutating.

Order costs: `unordered()`, `findAny`, plain `forEach` when order does not matter.

Helps only for large, CPU-bound, well-splitting work on idle cores — and only if you measured.

Boxing, `sorted` and repeated passes are the real stream costs; the loop is the fallback, not the enemy.

EXERCISE 1

Parallel, but correct

Read an integer `n`, `n` integers, and a threshold `t`. Compute every answer on a **parallel** stream, choosing operations whose results do not depend on scheduling:

`sum=` via `reduce(0, Integer::sum)` (a true identity, an associative operation).

`squares=` the squares in **input order**, collected with `toList()` — encounter order is preserved even in parallel.

`over=` how many values exceed `t`.

`firstOver=` the first value in input order that exceeds `t` (`findFirst`, not `findAny`), or `none`.

Example: `5` then `4 9 2 7 5` then `6` →

```
sum=27
squares=[16, 81, 4, 49, 25]
over=2
firstOver=9
```

INPUT 1

```
5
4 9 2 7 5
6
```

OUTPUT 1

```
sum=27
squares=[16, 81, 4, 49, 25]
over=2
firstOver=9
```

INPUT 2

```
3
1 2 3
10
```

OUTPUT 2

```
sum=6
squares=[1, 4, 9]
over=0
firstOver=none
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Integer> nums = new ArrayList<>();
        for (int i = 0; i < n; i++) nums.add(in.nextInt());
        int t = in.nextInt();

        System.out.println("sum=" + nums.parallelStream().reduce(0, Integer::sum));
        System.out.println("squares=" + nums.parallelStream().map(x -> x * x).toList
        System.out.println("over=" + nums.parallelStream().filter(x -> x > t).count(
        System.out.println("firstOver=" + nums.parallelStream().filter(x -> x > t).f
    }
}
```

EXERCISE 2

How a combiner thinks

Simulate the way a parallel reduction splits work. Read an integer `n`, `n` integers, and a chunk size `c`. Split the input into consecutive chunks of `c` (the last may be shorter). Using `IntStream.range` over the chunk indices and `Arrays.stream(a, from, to)` for each chunk, print one line per chunk

`chunk <i>: <sum>`, then `combined=<sum of the chunk sums>`, then `direct=<sum of the whole array>` — the two totals agree because addition is associative with identity 0.

Example: `5` then `1 2 3 4 5` then `2` →

```
chunk 0: 3
chunk 1: 7
chunk 2: 5
combined=15
direct=15
```

INPUT 1

```
5
1 2 3 4 5
2
```

OUTPUT 1

```
chunk 0: 3
chunk 1: 7
chunk 2: 5
combined=15
direct=15
```

INPUT 2

```
3
10 20 30
5
```

OUTPUT 2

```
chunk 0: 60
combined=60
direct=60
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int c = in.nextInt();
        int chunks = (n + c - 1) / c;

        int[] partial = IntStream.range(0, chunks)
            .map(i -> Arrays.stream(a, i * c, Math.min(n, (i + 1) * c)).sum())
            .toArray();
        IntStream.range(0, chunks).forEach(i -> System.out.println("chunk " + i + ":
        System.out.println("combined=" + Arrays.stream(partial).reduce(0, Integer::s
        System.out.println("direct=" + Arrays.stream(a).sum());
    }
}
```

QUIZ

1. Parallel streams run their tasks on…

A. A new thread per element

B. The common `ForkJoinPool`, shared by the whole JVM

C. The main thread only

D. A dedicated pool per stream

2. `list.parallelStream().forEach(results::add)` on an `ArrayList` is…

A. Fine

B. A data race: lost elements or an exception — use `collect`

C. Slower but correct

D. A compile error

3. Which source splits **worst** for parallel processing?

A. `int[]`

B. `ArrayList`

C. `LinkedList` or `Stream.iterate`

D. `IntStream.range`

4. Predict the output:

```
System.out.println(List.of(3, 1, 2).parallelStream().map(x → x * 10).toList())
```

A. Some permutation of 30, 10, 20

B. `[30, 10, 20]` — `toList` preserves encounter order

C. `[10, 20, 30]`

D. Undefined

5. The rough rule for when `parallel()` might pay is…

A. Always on a multicore machine

B. Elements × per-element cost above about 10 000, CPU-bound, well-splitting source, idle cores — and measured

C. Only for I/O

D. Only for `sorted()`

ANSWER KEY · PARALLEL STREAMS, ORDERING AND PERFORMANCE

1. B Which is why a blocking lambda in one parallel stream can starve every other parallel stream and `CompletableFuture` in the process.

2. B `ArrayList` is not thread-safe and `forEach` in parallel runs on several threads at once.

3. C Their spliterators cannot jump to the middle; splitting means walking, which eats the gain.

4. B A list source has an encounter order and `toList` respects it; only `forEach` and `findAny` give up ordering.

5. B Small or cheap workloads lose to the splitting and merging overhead; blocking work belongs in an executor, not the common pool.

CHECKPOINT · 30 MIN

Checkpoint — Streams

This checkpoint covers the stream model (sources, lazy intermediates, one terminal), every intermediate operation including `flatMap`, `sorted`, `distinct`, `limit` / `skip` and `takeWhile`, the terminals with the three forms of `reduce`, the collectors — `toMap`, `joining`, `groupingBy` with downstreams, `partitioningBy`, `teeing` — `Optional` as a return type, the primitive streams, and parallel execution with its rules.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- Why a pipeline with no terminal does nothing, and why a stream cannot be reused.
- When `map` is wrong and `flatMap` is right.
- What the one-argument `reduce` returns, and what a wrong identity does in parallel.
- How `toMap` fails on duplicate keys, and how to fix it.
- The shape of `groupingBy(classifier, mapFactory, downstream)`.
- `orElse` versus `orElseGet`; where `Optional` does not belong.
- Why `Stream<Integer>` is slower than `IntStream`, and what `sum()` of an empty stream is.
- The four things a lambda must not do inside a parallel pipeline.

The programs are a sales report built from `groupingBy` and `teeing`, a text analyser over `chars()` and `flatMap`, and a scheduler that finds the first free slot with `takeWhile`, `IntStream` and `Optional`.

EXERCISE 1

Sales report

Read an integer `n` and `n` lines `region product qty price` (integers for `qty` and `price`). Revenue of a line is `qty × price`. Using collectors, print for every region in natural order:

`<region>: revenue=<total revenue> items=<total qty> top=<product with the highest revenue in the region; ties by product name>`

Then one line `range=<region with the lowest total revenue>..<region with the highest>` (ties by region name, alphabetically first), computed with `teeing` over the per-region totals.

Example: `4` then `east pen 10 2`, `east ink 1 30`, `west pen 5 2`, `east pad 3 10` →

```
east: revenue=80 items=14 top=ink
west: revenue=10 items=5 top=pen
range=west..east
```

INPUT 1

```
4
east pen 10 2
east ink 1 30
west pen 5 2
east pad 3 10
```

OUTPUT 1

```
east: revenue=80 items=14 top=ink
west: revenue=10 items=5 top=pen
range=west..east
```

INPUT 2

```
2
north a 1 5
south b 1 6
```

OUTPUT 2

```
north: revenue=5 items=1 top=a
south: revenue=6 items=1 top=b
range=north..south
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;

record Sale(String region, String product, int qty, int price) {
    int revenue() { return qty * price; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Sale> sales = new ArrayList<>();
        for (int i = 0; i < n; i++) sales.add(new Sale(in.next(), in.next(), in.next

        Map<String, Integer> revenue = sales.stream().collect(groupingBy(Sale::regio
        Map<String, Integer> items = sales.stream().collect(groupingBy(Sale::region,
        Map<String, Map<String, Integer>> byProduct = sales.stream()
            .collect(groupingBy(Sale::region, groupingBy(Sale::product, summingInt(S

        Comparator<Map.Entry<String, Integer>> byValueThenKey =
            Map.Entry.<String, Integer>comparingByValue().thenComparing(Map.Entry.co
        for (String region : revenue.keySet()) {
            String top = byProduct.get(region).entrySet().stream().max(byValueThenKe
            System.out.println(region + ": revenue=" + revenue.get(region) + " items
        }

        Comparator<Map.Entry<String, Integer>> lowFirst = Map.Entry.<String, Integer
        String range = revenue.entrySet().stream().collect(teeing(
            minBy(lowFirst), maxBy(byValueThenKey),
            (lo, hi) -> lo.orElseThrow().getKey() + ".." + hi.orElseThrow().getKey()
        System.out.println("range=" + range);
    }
}
```

EXERCISE 2

Text analyser

Read **all** of standard input (several lines). Words are separated by whitespace; compare them case-insensitively. Print:

- `lines=<non-blank lines> words=<total words>`
- `unique=<distinct lower-cased words>`
- `longest=<the first longest word as written in the input>`

`common=<the three most frequent lower-cased words as word:count, most frequent first, ties by word, joined by ,>` (fewer if there are fewer words)

`letters=<the total number of letters across all words, via chars()>`

Build the word stream with `lines().flatMap(...)`.

Example input

```
the cat sat
the mat

The end
```

→

```
lines=3 words=7
unique=5
longest=the
common=the:3,cat:1,end:1
letters=21
```

INPUT 1

```
the cat sat
the mat

The end
```

OUTPUT 1

```
lines=3 words=7
unique=5
longest=the
common=the:3,cat:1,end:1
letters=21
```

INPUT 2

```
Hello
```

OUTPUT 2

```
lines=1 words=1
unique=1
longest=Hello
common=hello:1
letters=5
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;

public class Main {
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes());
        List<String> words = text.lines()
            .flatMap(l -> Arrays.stream(l.trim().split("\\s+")))
            .filter(w -> !w.isEmpty())
            .toList();

        long lines = text.lines().filter(l -> !l.isBlank()).count();
        System.out.println("lines=" + lines + " words=" + words.size());
        System.out.println("unique=" + words.stream().map(String::toLowerCase).disti
        System.out.println("longest=" + words.stream().max(Comparator.comparingInt(S

        Map<String, Long> freq = words.stream().map(String::toLowerCase).collect(gro
        String common = freq.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue(Comparator.reverseOrder
            .limit(3)
            .map(e -> e.getKey() + ":" + e.getValue())
            .collect(joining(","));
        System.out.println("common=" + common);
        System.out.println("letters=" + words.stream().flatMapToInt(String::chars).f
    }
}
```

EXERCISE 3

First free slot

A day has hours `0..23`. Read an integer `n` and `n` bookings `start end` (a booking occupies hours `start` to `end - 1`), then a duration `d`. Print:

- `busy=<number of distinct booked hours>` — flatten each booking to its hours with `flatMap` / `IntStream.range`, then `distinct`.
- `slot=<h>-<h+d>` for the **earliest** start `h` (`0 ≤ h ≤ 24 - d`) such that no booking overlaps `[h, h + d)` — an `IntStream` over candidate starts, `filter` with `noneMatch` over the bookings, `findFirst`; or `slot=none`.
- `free=<hours from 0 up to the first booked hour>` using `takeWhile` over `IntStream.range(0, 24)`.

Example: `2` then `9 12`, `13 15`, then `3` →

```
busy=5
slot=0-3
free=9
```

INPUT 1

```
2
9 12
13 15
3
```

OUTPUT 1

```
busy=5
slot=0-3
free=9
```

INPUT 2

```
1
0 24
1
```

OUTPUT 2

```
busy=24
slot=none
free=0
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

record Booking(int start, int end) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Booking> bookings = new ArrayList<>();
        for (int i = 0; i < n; i++) bookings.add(new Booking(in.nextInt(), in.nextIr
        int d = in.nextInt();

        Set<Integer> booked = bookings.stream()
            .flatMap(b -> IntStream.range(b.start(), b.end()).boxed())
            .collect(Collectors.toSet());
        System.out.println("busy=" + booked.size());

        OptionalInt slot = IntStream.rangeClosed(0, 24 - d)
            .filter(h -> bookings.stream().noneMatch(b -> h < b.end() && b.start() <
            .findFirst();
        System.out.println("slot=" + (slot.isPresent() ? slot.getAsInt() + "-" + (sl

        long free = IntStream.range(0, 24).takeWhile(h -> !booked.contains(h)).count
        System.out.println("free=" + free);
    }
}
```

CHECKPOINT QUESTIONS

1. A stream pipeline executes when…

A. The source is created

B. Each intermediate operation is added

C. The terminal operation is called

D. The stream is garbage collected

2. `Stream<List<Integer>>` to `Stream<Integer>` needs…

A. `map(List::stream)`

B. `flatMap(List::stream)`

C. `collect(toList())`

D. `boxed()`

3. Predict the output:

```
System.out.println(Stream.of("b", "a", "c").sorted().skip(1).findFirst().get())
```

A. `a`

B. `b`

C. `c`

D. Throws

4. The identity of a `reduce` must satisfy…

A. `identity == 0`

B. `op(identity, x) == x` for every `x`

C. `identity ≠ null`

D. Nothing in particular

5. `toMap(k, v)` on a stream containing two elements with the same key…

A. Keeps the first

B. Throws `IllegalStateException`

C. Keeps the last

D. Stores both in a list

6. `groupingBy(Person::city, TreeMap::new, counting())` returns…

A. `TreeMap<String, Long>`

B. `Map<String, List<Person>>`

C. `TreeMap<String, Integer>`

D. `Map<Long, String>`

7. `partitioningBy` differs from `groupingBy` on a boolean because…

A. It is faster

B. It always has both `true` and `false` keys

C. It returns a `List`

D. It cannot take a downstream

8. Which reads an Optional correctly?

A. `if (o.isPresent()) use(o.get());`

B. `o.ifPresent(this::use);`

C. `use(o.get());`

D. `if (o ≠ null) use(o.get());`

9. `orElseGet` is preferable to `orElse` when…

A. The default is a constant

B. The default is expensive to compute

C. The Optional is never empty

D. Never

10. `Stream<Integer>` lacks `sum()` because…

A. Integers cannot be summed

B. Summing is defined on the primitive streams; use `mapToInt(...).sum()` or `reduce(0, Integer::sum)`

C. It is deprecated

D. Only `LongStream` has it

11. `"héllo".chars()` yields…

A. A `Stream<Character>`

B. An `IntStream` of UTF-16 code units

C. A `char[]`

D. A `Stream<String>`

12. Which is safe inside a parallel pipeline?

A. Appending to a shared `ArrayList` in `map`

B. Incrementing a captured `int[]` counter

C. Collecting with `collect(toList())`

D. Calling `Thread.sleep` per element

13. `findAny` in parallel is faster than `findFirst` because…

A. It skips the filter

B. It need not respect encounter order, so the first chunk to finish wins

C. It caches results

D. It is not faster

14. `sorted()` to take the largest element instead of `max` …

A. Is equivalent

B. Is O(n log n) and buffers everything for an O(n) job

C. Is faster

D. Does not compile

15. `Stream.toList()` (Java 16) returns a list that is…

A. An `ArrayList` you may modify

B. Unmodifiable (nulls allowed)

C. A `LinkedList`

D. Sorted

ANSWER KEY · CHECKPOINT — STREAMS

1. C Intermediates only describe; the terminal pulls elements through.

2. B `map` would give `Stream<Stream<Integer>>`; `flatMap` splices the inner streams.

3. B Sorted → a, b, c; skip one → b, c; first → b.

4. B Otherwise parallel chunks each add the identity's effect once and the result changes with the split.

5. B Pass a merge function as the third argument to decide.

6. A The map factory names the map type; `counting()` produces `Long`.

7. B An empty partition is present as an empty group; `groupingBy` would omit the key.

8. B `isPresent` / `get` is a null check in disguise and `get` on empty throws; `ifPresent`, `orElse*` and `map` are the idioms.

9. B `orElse`'s argument is evaluated on every call; `orElseGet`'s supplier only when the Optional is empty.

10. B The primitive specialisations exist to provide numeric terminals without boxing.

11. B Cast to `char` when you need one; use `codePoints()` for characters outside the basic plane.

12. C Collectors are built for concurrent accumulation; the others race or block the common pool.

13. B `findFirst` must ensure no earlier element qualifies; `findAny` takes any survivor.

14. B `max` is a single pass; `sorted().findFirst()` sorts the whole stream first.

15. B For a list you can grow, use `collect(toCollection(ArrayList::new))`.