Module 19 of 20

Modern Java

The release train and LTS milestones, var and inference, pattern matching for instanceof, switch and records, the library additions from 9 to 21, virtual threads and structured concurrency, modules and the toolchain.

"What's new in Java 17?" and "what's new in Java 21?" are now standard interview questions, and the honest answer is: most of the language you have learned in this track. This module puts it in order.

It starts with the six-month release train and the LTS versions that production actually runs, with a one-screen cheat-sheet of what landed when. Then the features in depth: var and the inference family with the readability rules; pattern matching from instanceof through pattern switch to record patterns, with the exhaustiveness guarantee that retires the visitor pattern; the library additions you will use weekly — collection factories, the new String methods, Stream.toList, teeing, helpful NullPointerExceptions, HttpClient, sequenced collections. Virtual threads and structured concurrency are explained as the answer to the scaling problem the concurrency module left open, and the module system, jshell and the modern toolchain round it off.

The runtime behind the exercises is Java 17-level (OpenJDK 18), so records, sealed types, text blocks, switch expressions and pattern instanceof are all exercised; pattern switch, record patterns and virtual threads are taught as reading and quiz — with the form you will write on 21 shown beside the form that compiles here.

CONTENTS

1	The release train — what changed from Java 9 to 21	13 min
2	var and type inference — what the compiler already knows	12 min
3	Pattern matching — instanceof, switch and record patterns	15 min
4	The API additions worth knowing, 9 to 21	14 min
5	Virtual threads and structured concurrency	13 min
6	Modules, jshell and the modern toolchain	13 min
✓	Checkpoint — Modern Java	26 min

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

LESSON 1 · 13 MIN

The release train — what changed from Java 9 to 21

For twenty years Java shipped a major version every three to five years, each one enormous. Since 2017 it ships **every six months**, each release small, with a **long-term-support** version every two years. Most of the language you have learned in this track — `var`, records, sealed types, text blocks, switch expressions, pattern matching — arrived on that train, and interviewers now ask "what's new in Java 17?" as routinely as they ask about `HashMap`. This lesson is the timetable: how the cadence works, what *preview* means, which versions matter, and a one-screen cheat-sheet of what landed when.

The cadence

A **feature release** every March and September (`Java 18`, `Java 19`, …). Each is supported for six months, until the next one.

An **LTS** release every two years since 17 (previously three): **8**, **11**, **17**, **21**, next **25**. Vendors (Oracle, Eclipse Temurin, Amazon Corretto, Azul, Red Hat) ship security updates for LTS versions for years, which is why production runs on LTS and why "Java 17" and "Java 21" are the versions job adverts name.

Features are developed as **JEPs** (JDK Enhancement Proposals) and often go through **preview** (complete but subject to change; needs `--enable-preview` on both `javac` and `java`) or **incubator** (an API module, `jdk.incubator.*`) before becoming final. Preview code is fine for learning and forbidden in production — it can change or vanish.

`javac --release 17` compiles against the 17 API and rejects newer syntax, the way to guarantee your code runs on the LTS you target.

Nothing is removed lightly: deprecation for removal (`@Deprecated(forRemoval = true)`) precedes removal by at least one release, and the JVM warns at run time when you use such an API.

The cheat-sheet

Version	Year	Language	Library and JVM
9	2017	private interface methods; diamond with anonymous classes; `_` no longer an identifier	**modules** (JPMS), `jshell`, `List.of` / `Set.of` / `Map.of`, `Stream.takeWhile` / `dropWhile` / `iterate(seed, hasNext, next)`, `Optional.ifPresentOrElse` / `or` / `stream`, `Flow` (reactive), G1 default, compact strings
10	2018	**`var`** for locals	`List.copyOf`, `Collectors.toUnmodifiableList`, `Optional.orElseThrow()`, container awareness, parallel full GC for G1
11 (LTS)	2018	`var` in lambda parameters	**`HttpClient`**, `String.isBlank` / `strip` / `lines` / `repeat`, `Files.readString` / `writeString`, `Predicate.not`, single-file `java Hello.java`, ZGC (experimental), Epsilon GC; JavaFX and JAXB removed
12	2019	switch expressions (preview)	`Collectors.teeing`, `String.indent` / `transform`, Shenandoah
13	2019	text blocks (preview)	`String.stripIndent`, ZGC uncommit
14	2020	**switch expressions** (final), records (preview), pattern `instanceof` (preview)	**helpful** **`NullPointerException`** messages, JFR streaming; CMS removed
15	2020	**text blocks** (final), sealed classes (preview)	`String.formatted`, ZGC and Shenandoah production, hidden classes; Nashorn removed
16	2021	**records** (final), **pattern** **`instanceof`** (final)	`Stream.toList()`, `Stream.mapMulti`, `jpackage`, strong encapsulation of JDK internals by default, Unix-domain sockets

Version	Year	Language	Library and JVM
17 (LTS)	2021	**sealed classes** (final), pattern matching for switch (preview)	enhanced pseudo-random generators, macOS/AArch64 port; Applet API deprecated, Security Manager deprecated
18	2022	—	**UTF-8 by default** (JEP 400), simple web server `jwebserver`, code snippets in Javadoc, `finalize` deprecated for removal
19	2022	record patterns (preview)	**virtual threads** (preview), structured concurrency (incubator), `ExecutorService` is `AutoCloseable`
20	2023	—	scoped values (incubator), second previews
21 (LTS)	2023	**pattern matching for switch** (final), **record patterns** (final), unnamed patterns (preview), string templates (preview, later withdrawn), unnamed classes and instance `main` (preview)	**virtual threads** (final), **sequenced collections** (`getFirst` / `getLast` / `reversed`), generational ZGC, key encapsulation API
22–24	2024–25	unnamed variables `_` (final in 22), statements before `super()` (preview), primitive patterns (preview), flexible constructor bodies	`StructuredTaskScope` and scoped values (previews), Foreign Function & Memory API (final in 22), class-file API, stream gatherers (final in 24), `SecurityManager` permanently disabled (24)
25 (LTS)	2025	module import declarations, compact source files and instance `main` (final), flexible constructor bodies (final)	scoped values (final), stable values (preview), ahead-of-time cache improvements

You do not need every row. You need: **8** (lambdas, streams, `Optional`, `java.time`), **11** (`var`, `HttpClient`, string methods), **17** (records, sealed, switch expressions, text blocks, pattern `instanceof`), **21** (pattern switch, record patterns, virtual threads, sequenced collections). Those four LTS milestones are the versions of Java you will actually be asked about.

How to read a JEP

Each JEP page has *Summary*, *Goals*, *Non-goals*, *Motivation*, *Description*, *Alternatives* and *Risks*. The *Motivation* and *Alternatives* sections are where the design rationale lives — why records are not just data classes with less typing, why `var` was not allowed for fields — and reading two or three (JEP 395 records, JEP 409 sealed, JEP 444 virtual threads) teaches more about Java's design philosophy than any blog post.

Choosing a version in practice

New service, no constraints: the latest LTS (21 now, 25 soon).

A library: compile with `--release` set to the *oldest* version you support; test on the newest.

Legacy code on 8: the jump to 11 is the hard one (modules, removed APIs — `javax.xml.bind`, `sun.misc.Unsafe` access); 11 → 17 → 21 are mostly smooth.

Keep preview features out of `main`; a `--enable-preview` build is fine for learning.

Interview angle

"What's the release cadence and what is LTS?" Six-monthly feature releases; LTS every two years (8, 11, 17, 21, 25) with multi-year vendor support; production runs LTS.

"What's new in Java 17?" Sealed classes final; plus everything since 11: records, text blocks, switch expressions, pattern `instanceof`, helpful NPEs.

"What's new in Java 21?" Pattern matching for switch and record patterns final, virtual threads, sequenced collections, generational ZGC.

"What is a preview feature?" Complete but may change; needs `--enable-preview`; not for production.

"Which version added `var`?" 10 for locals; 11 for lambda parameters.

Key takeaways

Six-month releases; LTS 8 → 11 → 17 → 21 → 25; production runs LTS.

Preview and incubator features need `--enable-preview` and stay out of production.

11: `var`, `HttpClient`, string methods. 17: records, sealed, switch expressions, text blocks, pattern `instanceof`. 21: pattern switch, record patterns, virtual threads, sequenced collections.

`--release N` pins the API and syntax you compile against.

JEP *Motivation* sections are the best design documentation Java has.

EXERCISE 1

Modern string toolkit

Read all of standard input. Using only the Java 11+ `String` methods — `lines()`, `strip()`, `isBlank()`, `repeat()`, `formatted()` — print:

`nonBlank=<number of non-blank lines>`

`stripped=<the non-blank lines, stripped, joined by ▷`

`banner=<a line of = as long as the longest stripped line>`

`report=<"%d lines, %d chars".formatted(nonBlank, total characters of the stripped lines)>`

Example input (note the surrounding spaces and the blank line)

```
 hello

world!!
```

→

```
nonBlank=2
stripped=hello|world!!
banner========
report=2 lines, 12 chars
```

INPUT 1

```
 hello

world!!
```

OUTPUT 1

```
nonBlank=2
stripped=hello|world!!
banner========
report=2 lines, 12 chars
```

INPUT 2

OUTPUT 2

```
nonBlank=0
stripped=
banner=
report=0 lines, 0 chars
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes());
        List<String> lines = text.lines().filter(Predicate.not(String::isBlank)).map
        int longest = lines.stream().mapToInt(String::length).max().orElse(0);
        int chars = lines.stream().mapToInt(String::length).sum();
        System.out.println("nonBlank=" + lines.size());
        System.out.println("stripped=" + String.join("|", lines));
        System.out.println("banner=" + "=".repeat(longest));
        System.out.println("report=" + "%d lines, %d chars".formatted(lines.size(),
    }
}
```

EXERCISE 2

Which release?

Build an immutable lookup table with `Map.ofEntries` from feature name to the Java version that finalised it: `modules` 9, `var` 10, `http-client` 11, `switch-expressions` 14, `text-blocks` 15, `records` 16, `pattern-instanceof` 16, `sealed` 17, `pattern-switch` 21, `record-patterns` 21, `virtual-threads` 21, `sequenced-collections` 21. LTS versions are `Set.of(8, 11, 17, 21)`. Read an integer `n` and `n` feature names; print `<feature>: Java <v> (LTS)` or `(not LTS)`, or `<feature>: unknown`.

Example: `3` then `records sealed lambdas` →

```
records: Java 16 (not LTS)
sealed: Java 17 (LTS)
lambdas: unknown
```

INPUT 1

```
3
records sealed lambdas
```

OUTPUT 1

```
records: Java 16 (not LTS)
sealed: Java 17 (LTS)
lambdas: unknown
```

INPUT 2

```
2
virtual-threads var
```

OUTPUT 2

```
virtual-threads: Java 21 (LTS)
var: Java 10 (not LTS)
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Map<String, Integer> since = Map.ofEntries(
            Map.entry("modules", 9), Map.entry("var", 10), Map.entry("http-client",
            Map.entry("switch-expressions", 14), Map.entry("text-blocks", 15), Map.e
            Map.entry("pattern-instanceof", 16), Map.entry("sealed", 17), Map.entry(
            Map.entry("record-patterns", 21), Map.entry("virtual-threads", 21), Map.
        Set<Integer> lts = Set.of(8, 11, 17, 21);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String feature = in.next();
            Integer v = since.get(feature);
            if (v == null) System.out.println(feature + ": unknown");
            else System.out.println(feature + ": Java " + v + (lts.contains(v) ? " (
        }
    }
}
```

QUIZ

1. Java feature releases ship...

A. Every three years

B. Every six months, with an LTS every two years

C. Once a year

D. When ready

2. A *preview* feature…

A. Is production-ready

B. Is complete but may change; needs `--enable-preview` and should stay out of production

C. Is deprecated

D. Only works in `jshell`

3. Records and pattern `instanceof` became final in…

A. Java 11

B. Java 14

C. Java 16

D. Java 21

4. `javac --release 17` does what?

A. Optimises for Java 17

B. Compiles against exactly the Java 17 API and syntax, rejecting newer features

C. Enables previews

D. Targets Java 8

5. Which pairing is correct?

A. `var` — 8; text blocks — 11

B. Pattern switch — 17; virtual threads — 19 final

C. Pattern switch and virtual threads — 21; sealed — 17; `var` — 10

D. Records — 21; sealed — 21

ANSWER KEY · THE RELEASE TRAIN — WHAT CHANGED FROM JAVA 9 TO 21

1. B 8, 11, 17, 21 and 25 are the LTS versions production runs.

2. B Incubator modules (`jdk.incubator.*`) are the API equivalent.

3. C Previewed in 14–15, final in 16; sealed classes followed in 17.

4. B The way to guarantee a library runs on the LTS you claim to support.

5. C The four LTS milestones and their headline features are the ones to memorise.

LESSON 2 · 12 MIN

var and type inference — what the compiler already knows

`var list = new ArrayList<String>();` — Java 10 let you stop writing the type twice. `var` is **local variable type inference**: the variable is still statically typed, the compiler simply works the type out from the initialiser. It is not dynamic typing, not `Object`, and not optional in the places it is allowed — a `var` with no initialiser is a compile error. This lesson covers exactly where `var` is permitted, what type it infers (including the surprising cases), the style rules that keep it readable, and the other inference features it joined: the diamond, generic method inference and lambda parameter inference.

The rules

`var` is allowed for: local variables **with an initialiser**, the index of a `for` loop, the variable of an enhanced `for`, try-with-resources variables, and (Java 11) lambda parameters when every parameter uses it. It is **not** allowed for fields, method parameters, return types, or a local initialised to `null` or to a lambda / method reference (no target type to infer from).

```
var count = 0;                              // int
var name = "Ada";                           // String
var names = new ArrayList<String>();        // ArrayList<String> — the concrete
var map = new HashMap<String, List<Integer>>();
for (var entry : map.entrySet()) { … }      // Map.Entry<String, List<Integer>>
try (var in = Files.newBufferedReader(path)) { … }
var f = (Function<String, Integer>) String::length;   // needs the cast; `var

var x;                  // error: cannot infer type
var n = null;           // error
var list = List.of(); // List<Object> — probably not what you meant
```

`var` is a *reserved type name*, not a keyword: `int var = 1;` still compiles, so old code is unaffected — which is the only reason it could be added at all.

What type does it infer?

The **static type of the initialiser expression**, with two adjustments. First, the type is the *concrete* type, not an interface: `var list = new ArrayList<String>()` is `ArrayList<String>`, so you cannot later assign a `LinkedList` to it — write `List<String> list = new ArrayList<>()` when you want the interface type. Second, for an expression whose type cannot be denoted — an anonymous class, a capture of a wildcard, an intersection type — `var` gives you that undenotable type, which was impossible before:

```
var counter = new Object() { int n; void inc() { n++; } };
counter.inc();                              // legal: the anonymous class's own
```

Literals infer as expected — `var d = 1.0` is `double`, `var l = 1L` is `long`, `var b = (byte) 1` is `byte` — and `var c = 'a'` is `char`, not `int`. Ternaries infer the common supertype: `var v = flag ? 1 : "one"` is `Serializable & Comparable<…>`, a sign that you should not have written that.

Style: when `var` helps and when it hides

The guidelines from the JDK team (JEP 286's style guide) reduce to one test: **can the reader tell the type from the line?**

```
var users = userRepository.findActive();        // hides: is it a List? Set? S
List<User> users = userRepository.findActive(); // says it
var userCount = users.size();                   // fine: obviously an int
var in = new BufferedReader(new InputStreamReader(System.in));   // fine: the
var byCity = people.stream().collect(groupingBy(Person::city, TreeMap::new, co
```

Use `var` when the initialiser names the type (`new X(...)`, a literal, a cast, a factory whose name says the type — `Path.of`, `List.of`), when the type is long and obvious from context (the `Map.Entry` in a for-each), or when the variable is short-lived. Avoid it when the type carries information the reader needs, when the method name is generic (`get`, `process`, `handle`), and when the inferred type is a concrete class you would rather have been an interface. Good variable names matter more with `var`: `var count = …` is clear; `var result = …` is not.

The rest of the inference family

Diamond (Java 7): `new ArrayList<>()` infers the type argument from the assignment target. Since Java 9 it works with anonymous classes too: `new Comparator<>() { … }`.

Generic method inference: `Collections.emptyList()` and `List.of()` infer their type argument from the target — `List<String> l = List.of();`. In an argument position Java 8's *target typing* extended this: `process(List.of())` infers from `process`'s parameter type.

Lambda parameter inference: `(a, b) → a + b` gets its parameter types from the functional interface it is assigned to; `(var a, var b) → a + b` (Java 11) says the same thing but lets you put an annotation on the parameter (`(@Nonnull var s) → …`), which was the whole point.

Pattern variables (next lesson): `if (o instanceof String s)` infers `s` as `String`.

Inference never crosses statement boundaries and never uses later assignments: the type is fixed at the declaration.

`var` in interviews

The question is rarely "does `var` exist" and usually "is Java dynamically typed now?" (no — the type is inferred once and fixed), "can you use `var` for a field?" (no), "what is `var x = new ArrayList<>()`?" (`ArrayList<Object>` — the diamond has no target to infer from, a genuine trap), and "when would you not use it?" (when the type is not obvious from the right-hand side).

Interview angle

"Is `var` dynamic typing?" No — static inference from the initialiser; the type never changes.

"Where can't you use `var`?" Fields, parameters, return types, without an initialiser, with `null`, with a bare lambda or method reference.

"`var list = new ArrayList<>()` has what type?" `ArrayList<Object>` — there is no target for the diamond.

"Why is `var` a reserved type name rather than a keyword?" Backwards compatibility: `var` remained a legal identifier.

"Why allow `var` in lambda parameters?" Uniformity, and to allow annotations on inferred parameters.

Key takeaways

`var` = local inference from the initialiser; static, fixed, concrete type — `ArrayList`, not `List`.

Locals, `for`, for-each, try-with-resources, lambda parameters (all or none); never fields, parameters, `null`, bare lambdas.

Readability test: can the reader tell the type from the line? Name variables well.

Diamond, generic method inference and lambda inference are the same idea from the other side.

`new ArrayList<>()` with `var` is `ArrayList<Object>`; `List.of()` with `var` is `List<Object>`.

EXERCISE 1

Let the compiler type it

Read all of standard input and split it into words. Using `var` for **every local variable**, group the distinct words by length into a `TreeMap<Integer, List<String>>` (words sorted within a group), and print one line per group: `<length>: <words joined by ,>`. Then print `largestGroup=<the length with the most words; ties to the smaller length>`.

Example: `to be or not to be that is it` →

```
2: be,is,it,or,to
3: not
4: that
largestGroup=2
```

INPUT 1

```
to be or not to be that is it
```

OUTPUT 1

```
2: be,is,it,or,to
3: not
4: that
largestGroup=2
```

INPUT 2

```
aaa bb aaa
```

OUTPUT 2

```
2: bb
3: aaa
largestGroup=2
```

REFERENCE SOLUTION

```java
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 {
        var text = new String(System.in.readAllBytes());
        var words = Arrays.stream(text.trim().split("\\s+")).filter(w -> !w.isEmpty(
        var groups = words.stream().collect(groupingBy(String::length, TreeMap::new,
        var largest = -1;
        var largestSize = -1;
        for (var e : groups.entrySet()) {
            System.out.println(e.getKey() + ": " + String.join(",", e.getValue()));
            if (e.getValue().size() > largestSize) { largestSize = e.getValue().size
        }
        System.out.println("largestGroup=" + largest);
    }
}
```

EXERCISE 2

Diamond with an anonymous class

Read an integer `n` and `n` words. Sort them with an **anonymous** `Comparator<>` using the diamond (Java 9) — by length, then alphabetically — and print the sorted words space-separated. Then declare `var tally = new Object() { int calls; int bump() { return ++calls; } };`, call `bump()` once per word, and print `calls=<tally.calls>` — a method on an anonymous class that only `var` can give you a handle to.

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

```
fig kiwi pear apple
calls=4
```

INPUT 1

```
4
pear fig apple kiwi
```

OUTPUT 1

```
fig kiwi pear apple
calls=4
```

INPUT 2

```
1
z
```

OUTPUT 2

```
z
calls=1
```

REFERENCE SOLUTION

```java
import java.util.*;

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());
        Comparator<String> byLengthThenAlpha = new Comparator<>() {
            @Override public int compare(String a, String b) {
                int c = Integer.compare(a.length(), b.length());
                return c != 0 ? c : a.compareTo(b);
            }
        };
        words.sort(byLengthThenAlpha);
        System.out.println(String.join(" ", words));
        var tally = new Object() {
            int calls;
            int bump() { return ++calls; }
        };
        for (String w : words) tally.bump();
        System.out.println("calls=" + tally.calls);
    }
}
```

QUIZ

1. `var` makes Java…

A. Dynamically typed

B. Still statically typed — the type is inferred from the initialiser once and never changes

C. Untyped

D. Slower

2. Which does **not** compile?

A. `var n = 10;`

B. `var list = new ArrayList<String>();`

C. `var f = String::length;`

D. `for (var e : map.entrySet())`

3. `var list = new ArrayList<>();` infers…

A. `ArrayList<String>`

B. `ArrayList<Object>` — the diamond has no target to infer from

C. `List<Object>`

D. Compile error

4. `var` is allowed for…

A. Fields

B. Method parameters

C. Local variables with an initialiser, loop variables, try-with-resources, and (11+) lambda parameters

D. Return types

5. The readability test for `var` is…

A. Is the line shorter?

B. Can the reader tell the type from the line — the initialiser or the name?

C. Is it a loop?

D. Is the type generic?

ANSWER KEY · VAR AND TYPE INFERENCE — WHAT THE COMPILER ALREADY KNOWS

1. B It is local variable type inference, not `Object` and not dynamic typing.

2. C A bare method reference or lambda has no type without a target; cast it or declare the functional interface.

3. B A genuine trap: give the type argument when you use `var` with `new`.

4. C Never fields, parameters or return types; never without an initialiser or with `null`.

5. B `var users = repo.find()` hides; `var in = new BufferedReader(...)` does not.

LESSON 3 · 15 MIN

Pattern matching — instanceof, switch and record patterns

Pattern matching is the biggest change to how Java code *reads* since lambdas. It began quietly in Java 16 — `if (o instanceof String s)` binds `s` without a cast — and by Java 21 it lets a `switch` take a value of a sealed type apart by shape, down through nested records, with the compiler checking that every case is covered. The result is that the "sum type + function by cases" style of functional languages is now idiomatic Java, and the visitor pattern is mostly retired. This lesson covers all three pieces, what compiles on the runtime you have (Java 17-level: `instanceof` patterns, records, sealed types) and what needs 21 (pattern `switch`, record patterns), and the exhaustiveness rule that makes it safe.

Pattern `instanceof` (Java 16)

```
// before
if (o instanceof String) { String s = (String) o; use(s.length()); }
// after
if (o instanceof String s) { use(s.length()); }
if (o instanceof String s && s.length() > 3) { … }          // s is in scope
if (!(o instanceof String s)) return;                        // flow scoping:
use(s);
```

The **type pattern** `String s` tests and binds in one step. Scope follows *definite matching*: `s` exists wherever the compiler can prove the test succeeded — the `if` body, the right of `&&`, and after an early return on the negated test. It does **not** exist on the right of `||`, and a pattern variable may shadow a field but not a local. The cast is gone, and with it the `ClassCastException` you get when the check and the cast drift apart during a refactor.

Sealed types set up the switch

Recall from the interfaces module: a `sealed interface Shape permits Circle, Square, Rect` fixes the set of implementations. That fixed set is what lets a switch over it be **exhaustive** without a `default`. Until pattern `switch` arrived, the way to dispatch on a sealed hierarchy was a chain of pattern `instanceof`s — still the form you write on Java 17:

```
sealed interface Shape permits Circle, Square, Rect {}
record Circle(double r) implements Shape {}
record Square(double side) implements Shape {}
record Rect(double w, double h) implements Shape {}

static double area(Shape s) {
    if (s instanceof Circle c) return Math.PI * c.r() * c.r();
    if (s instanceof Square q) return q.side() * q.side();
    if (s instanceof Rect r) return r.w() * r.h();
    throw new IllegalStateException("unreachable: " + s);      // the compiler
}
```

Pattern matching for `switch` (Java 21)

```
static double area(Shape s) {
    return switch (s) {
        case Circle c → Math.PI * c.r() * c.r();
        case Square q → q.side() * q.side();
        case Rect r   → r.w() * r.h();
    };                                              // no default: the compiler
}
```

What changed: `case` may be a **type pattern**; the selector may be any reference type (not just primitives, strings and enums); the switch is checked for **exhaustiveness** — for a sealed type it must cover every permitted subtype (or have a `default`); and cases are checked for **dominance** — a `case Object o` before `case String s` is a compile error because it would swallow it. Add a subtype to `Shape` and every exhaustive switch in the codebase fails to compile until it handles the new case. That is the safety the visitor pattern promised and delivered only with boilerplate.

Guards refine a case with `when`:

```
case Circle c when c.r() == 0 → 0;
case Circle c → Math.PI * c.r() * c.r();
```

`null` can be a case — `case null → …` — or `case null, default →`; a switch without it throws `NullPointerException` on a null selector, as switches always have.

Record patterns (Java 21)

A record pattern deconstructs a record into its components, and nests:

```
record Point(int x, int y) {}
record Line(Point from, Point to) {}

if (o instanceof Point(int x, int y)) System.out.println(x + y);
static String describe(Object o) {
    return switch (o) {
        case Line(Point(var x1, var y1), Point(var x2, var y2)) when x1 == x2
        case Line(Point from, Point to) → "line";
        case Point(int x, int y) when x == 0 && y == 0 → "origin";
        case Point p → "point";
        default → "other";
    };
}
```

`var` is allowed for component types. Nesting goes as deep as the data does, and the compiler checks that the components exist and are compatible. Java 22 adds `_` for components you do not need (`case Point(int x, _)`). Records + sealed interfaces + pattern switch is the trio: **algebraic data types with exhaustive matching**, and it is how modern Java models results (`Ok` / `Err`), commands, AST nodes and states.

Where this replaces old idioms

Old	New
`instanceof` + cast	pattern `instanceof`
visitor pattern over a hierarchy	sealed + exhaustive pattern `switch`
chain of `if (x instanceof …)`	pattern `switch` with dominance and exhaustiveness checks
`equals`-based `switch` on a "type" string field	switch on the actual type
getters to unpack a data object	record pattern

The visitor is not dead — it still allows adding operations without touching the hierarchy's *definition* module — but for code that owns both the types and the operations, patterns win on every axis.

Compiler guarantees, summarised

Exhaustiveness: a switch expression must cover all cases; over a sealed type, every permitted subtype (recursively for sealed subtypes); over an enum, every constant; otherwise a `default` is required. A switch *statement* with patterns must also be exhaustive (older statement switches on enums need not be).

Dominance: an earlier case may not dominate a later one; unguarded type patterns come after guarded ones for the same type.

Scope: pattern variables are definitely assigned only where the match is proven.

Separate compilation: if a sealed hierarchy changes and a switch is not recompiled, the JVM throws `MatchException` at run time — the compile-time promise degrades gracefully.

Interview angle

"What does `if (o instanceof String s)` do?" Tests and binds in one; `s` is scoped where the match is definite; no cast.

"What does sealed buy you with `switch`?" Exhaustiveness without `default`: adding a subtype breaks every switch that forgets it.

"Pattern `switch` versus visitor?" Same exhaustiveness guarantee, no boilerplate, but operations live with the types; visitor still allows extension across modules.

"What is a record pattern?" Deconstructing a record into components in a `case` or `instanceof`, nestable.

"What happens on `switch (null)`?" `NullPointerException` unless there is a `case null`.

Key takeaways

Pattern `instanceof` (16): test + bind, flow-scoped, no cast; on Java 17 an `if`-chain is how you dispatch over a sealed type.

Pattern `switch` (21): type patterns as cases, `when` guards, `case null`, exhaustiveness and dominance checked by the compiler.

Record patterns (21) deconstruct, nest, and take `var`; sealed + records + switch = algebraic data types.

Exhaustive switches turn "I added a subtype" into compile errors at every site that must change.

Prefer patterns to visitors when you own both the hierarchy and the operations.

EXERCISE 1

Shapes with pattern instanceof

Declare `sealed interface Shape permits Circle, Square, Rect` with records `Circle(double r)`, `Square(double side)` and `Rect(double w, double h)`. Read an integer `n` and `n` shapes — `circle <r>`, `square <side>` or `rect <w> <h>` — and for each print `<Kind> area=<two decimals> perimeter=<two decimals>` using **pattern** **`instanceof`** (no casts). Then print `largest=<record toString of the largest-area shape>` and `total=<sum of areas, two decimals>`. Use `Locale.ROOT`.

Example: `2` then `circle 1`, `rect 2 3` →

```
Circle area=3.14 perimeter=6.28
Rect area=6.00 perimeter=10.00
largest=Rect[w=2.0, h=3.0]
total=9.14
```

INPUT 1

```
2
circle 1
rect 2 3
```

OUTPUT 1

```
Circle area=3.14 perimeter=6.28
Rect area=6.00 perimeter=10.00
largest=Rect[w=2.0, h=3.0]
total=9.14
```

INPUT 2

```
1
square 2.5
```

OUTPUT 2

```
Square area=6.25 perimeter=10.00
largest=Square[side=2.5]
total=6.25
```

REFERENCE SOLUTION

```java
import java.util.*;

sealed interface Shape permits Circle, Square, Rect { }
record Circle(double r) implements Shape { }
record Square(double side) implements Shape { }
record Rect(double w, double h) implements Shape { }

public class Main {
    static double area(Shape s) {
        if (s instanceof Circle c) return Math.PI * c.r() * c.r();
        if (s instanceof Square q) return q.side() * q.side();
        if (s instanceof Rect r) return r.w() * r.h();
        throw new IllegalStateException();
    }
    static double perimeter(Shape s) {
        if (s instanceof Circle c) return 2 * Math.PI * c.r();
        if (s instanceof Square q) return 4 * q.side();
        if (s instanceof Rect r) return 2 * (r.w() + r.h());
        throw new IllegalStateException();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Shape> shapes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            shapes.add(switch (kind) {
                case "circle" -> new Circle(in.nextDouble());
                case "square" -> new Square(in.nextDouble());
                default -> new Rect(in.nextDouble(), in.nextDouble());
            });
        }
        Shape largest = null;
        double total = 0;
        for (Shape s : shapes) {
            double a = area(s);
            total += a;
            if (largest == null || a > area(largest)) largest = s;
            System.out.println(String.format(Locale.ROOT, "%s area=%.2f perimeter=%.
        }
        System.out.println("largest=" + largest);
        System.out.println(String.format(Locale.ROOT, "total=%.2f", total));
    }
}
```

EXERCISE 2

An expression tree

Declare `sealed interface Expr permits Num, Add, Mul, Neg` with records `Num(int value)`, `Add(Expr left, Expr right)`, `Mul(Expr left, Expr right)` and `Neg(Expr inner)`. Read one line of **reverse Polish** tokens — integers, `+`, `*`, and `~` for negation — and build the tree with a stack. Then, with pattern `instanceof`, print `expr=<rendering>` where a `Num` is its value, `Add` is `(l + r)`, `Mul` is `(l * r)` and `Neg` is `-(x)`; `value=<evaluation>`; and `depth=<height of the tree, a lone Num being 1>`.

Example: `1 2 3 * +` →

```
expr=(1 + (2 * 3))
value=7
depth=3
```

INPUT 1

```
1 2 3 * +
```

OUTPUT 1

```
expr=(1 + (2 * 3))
value=7
depth=3
```

INPUT 2

```
4 ~
```

OUTPUT 2

```
expr=-(4)
value=-4
depth=2
```

REFERENCE SOLUTION

```
import java.util.*;

sealed interface Expr permits Num, Add, Mul, Neg { }
record Num(int value) implements Expr { }
record Add(Expr left, Expr right) implements Expr { }
record Mul(Expr left, Expr right) implements Expr { }
record Neg(Expr inner) implements Expr { }

public class Main {
    static String render(Expr e) {
        if (e instanceof Num n) return String.valueOf(n.value());
        if (e instanceof Add a) return "(" + render(a.left()) + " + " + render(a.rig
        if (e instanceof Mul m) return "(" + render(m.left()) + " * " + render(m.rig
        if (e instanceof Neg g) return "-(" + render(g.inner()) + ")";
        throw new IllegalStateException();
    }
    static int eval(Expr e) {
        if (e instanceof Num n) return n.value();
        if (e instanceof Add a) return eval(a.left()) + eval(a.right());
        if (e instanceof Mul m) return eval(m.left()) * eval(m.right());
        if (e instanceof Neg g) return -eval(g.inner());
        throw new IllegalStateException();
    }
    static int depth(Expr e) {
        if (e instanceof Num) return 1;
        if (e instanceof Add a) return 1 + Math.max(depth(a.left()), depth(a.right()
        if (e instanceof Mul m) return 1 + Math.max(depth(m.left()), depth(m.right()
        if (e instanceof Neg g) return 1 + depth(g.inner());
        throw new IllegalStateException();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Deque<Expr> stack = new ArrayDeque<>();
        while (in.hasNext()) {
            String t = in.next();
            switch (t) {
                case "+" -> { Expr r = stack.pop(), l = stack.pop(); stack.push(new
                case "*" -> { Expr r = stack.pop(), l = stack.pop(); stack.push(new
                case "~" -> stack.push(new Neg(stack.pop()));
                default -> stack.push(new Num(Integer.parseInt(t)));
            }
        }
        Expr root = stack.pop();
        System.out.println("expr=" + render(root));
        System.out.println("value=" + eval(root));
        System.out.println("depth=" + depth(root));
```

```
    }
}
```

QUIZ

1. In `if (!(o instanceof String s)) return; use(s);` the variable `s` …

A. Is out of scope at `use`

B. Is in scope at `use` — flow scoping knows the test succeeded past the early return

C. Is `null`

D. Causes a compile error

2. A pattern `switch` over a sealed interface with no `default` …

A. Does not compile

B. Compiles if every permitted subtype has a case — the compiler checks exhaustiveness

C. Throws at runtime

D. Matches nothing

3. `case Object o → …` placed before `case String s → …` …

A. Is fine

B. Is a compile error: the earlier case dominates the later one

C. Matches strings first

D. Is a warning

4. `case Line(Point(var x1, var y1), Point p2) when x1 == 0 → …` is…

A. Invalid syntax

B. A nested record pattern with a guard (Java 21)

C. A type pattern only

D. Java 8 code

5. `switch (obj)` with a `null` selector and no `case null` …

A. Matches `default`

B. Throws `NullPointerException`

C. Skips the switch

D. Matches the first case

6. On a Java 17 runtime, dispatching over a sealed hierarchy is done with…

A. Pattern `switch`

B. A chain of pattern `instanceof` checks (pattern `switch` is 21)

C. Reflection

D. `equals`

ANSWER KEY · PATTERN MATCHING — INSTANCEOF, SWITCH AND RECORD PATTERNS

1. B Pattern variables are in scope wherever the match is definitely true.

2. B Adding a subtype later breaks every such switch at compile time — the safety the visitor pattern offered with more boilerplate.

3. B Dominance checking keeps unreachable cases from compiling.

4. B Record patterns nest arbitrarily and accept `var` for components; `when` refines the case.

5. B As switches always have; `case null →` or `case null, default →` handles it explicitly.

6. B Same records, same sealed interface; the exhaustive switch form arrives in 21.

LESSON 4 · 14 MIN

The API additions worth knowing, 9 to 21

Language features get the headlines; the library additions save you more lines per day. Since Java 9 the standard library has gained immutable collection factories, a dozen `String` methods, a modern HTTP client, `Stream.toList()`, `Collectors.teeing`, helpful `NullPointerException` messages and, in 21, the sequenced collections that finally give `List` a `getLast()`. This lesson is the tour of the ones you will use weekly, grouped by what they replace, with the version each arrived in so you know what your target JDK has.

Collection factories (9) and copies (10)

```
List<String> names = List.of("ann", "bob");        // immutable; nulls rejecte
Set<Integer> primes = Set.of(2, 3, 5);              // immutable; duplicates r
Map<String, Integer> ages = Map.of("ann", 30, "bob", 25);          // up to 10
Map<String, Integer> big = Map.ofEntries(Map.entry("a", 1), Map.entry("b", 2))
List<String> frozen = List.copyOf(mutableList);     // immutable snapshot; re
```

They replace `Arrays.asList` (fixed-size but *mutable* via `set`, and backed by the array) and `Collections.unmodifiableList` (a live *view* — the underlying list can still change). The factories are truly immutable, compact (a two-element list is one small object), and **null-hostile**. Any mutator throws `UnsupportedOperationException`. `Set.of` and `Map.of` iterate in an order that is randomised per JVM run on purpose, so nobody comes to depend on it.

Strings (11, 12, 13, 15)

```
"  x ".strip();  " x".stripLeading();  "x ".stripTrailing();    // Unicode-awa
"   ".isBlank();                                               // true — wh
"a\nb\r\nc".lines();                                           // Stream<St
"ab".repeat(3);                                                // "ababab"
"line".indent(4);                                              // 12: "
"hello".transform(String::toUpperCase);                         // 12: appl
"""
  a
    b
  """.stripIndent();                                           // 13: remov
"%s has %d".formatted("ann", 3);                               // 15: Strin
"héllo".chars(); "😀".codePoints();                             // 9: codeP
```

`strip` versus `trim`: `trim` removes characters ≤ `U+0020`; `strip` uses `Character.isWhitespace`, so it also removes Unicode spaces like `U+2003`. `lines()` plus `strip` plus `isBlank` is the modern way to clean text input; `repeat` retires the `StringBuilder` loop for padding.

Streams and collectors (9, 12, 16)

```
Stream.iterate(1, x → x < 100, x → x * 2);       // 9: finite iterate
stream.takeWhile(p); stream.dropWhile(p);          // 9
Stream.ofNullable(maybeNull);                      // 9: 0 or 1 elements
Collectors.filtering(p, downstream); Collectors.flatMapping(f, downstream);
Collectors.teeing(c1, c2, merger);                 // 12
stream.toList();                                   // 16: unmodifiable, short
stream.mapMulti((x, sink) → { if (…) sink.accept(…); });   // 16: an imperati
Predicate.not(String::isBlank);                    // 11: readable negation c
```

`Optional` (9, 10, 11)

`ifPresentOrElse(action, emptyAction)`, `or(supplier)` and `stream()` in 9; `orElseThrow()` with no arguments in 10 (the honest replacement for `get()`); `isEmpty()` in 11. Covered in the streams module — mentioned here so you know which JDK has them.

`Objects` and helpful NPEs (9, 14)

```
String name = Objects.requireNonNullElse(input, "anonymous");         // 9
int len = Objects.requireNonNullElseGet(list, ArrayList::new).size(); // 9
Objects.checkIndex(i, size);                                          // 9: th
```

Java 14's **helpful** `NullPointerException` (on by default since 15) says *what* was null: `Cannot invoke "String.length()" because the return value of "Person.name()" is null` — instead of a bare line number. It makes the most common exception in Java self-explanatory; when a local is involved the message names it if the class was compiled with `-g`, otherwise `<local4>`.

`HttpClient` (11)

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest req = HttpRequest.newBuilder(URI.create("https://api.example.com/v
    .header("Accept", "application/json").timeout(Duration.ofSeconds(5)).GET()
HttpResponse<String> res = client.send(req, HttpResponse.BodyHandlers.ofStrin
CompletableFuture<HttpResponse<String>> async = client.sendAsync(req, BodyHan
```

HTTP/2, WebSockets, asynchronous via `CompletableFuture`, immutable request objects, a builder API — everything `HttpURLConnection` was not. For most services it removes the need for a third-party client.

Files, time and misc (9, 11, 16, 18)

`Files.readString` / `writeString` (11), `Files.mismatch` (12); `Path.of` (11); `InputStream.readAllBytes` (9) and `transferTo` (9) — `in.transferTo(out)` is the copy loop you never write again; `Duration.toSeconds` / `toDaysPart` (9); `Math.floorMod` overloads; `Random`'s `RandomGenerator` interface with better algorithms (17: `RandomGenerator.of("L64X128MixRandom")`); `Thread.onSpinWait` (9); `Process.pid()` and `ProcessHandle` (9); the built-in `jwebserver` static file server (18); `System.Logger` (9) as a facade over whatever logging framework is present.

Java 21: sequenced collections and friends

```
SequencedCollection<E>: getFirst(), getLast(), addFirst(e), addLast(e), remove
list.getLast();                    // instead of list.get(list.size() - 1)
list.reversed();                   // a live reversed VIEW, not a copy
new LinkedHashSet<>(…).getFirst(); // insertion-ordered sets finally expose th
sortedMap.firstEntry(); linkedHashMap.pollLastEntry();  // SequencedMap
```

`List`, `Deque`, `LinkedHashSet`, `SortedSet`, `LinkedHashMap` and `SortedMap` all gained these; `getLast()` on an empty collection throws `NoSuchElementException`. Also in 21: `StringBuilder.repeat`, `Math.clamp`, `String.indexOf(ch, from, to)`, `Character.isEmoji`, and the `HexFormat` class (17) for the hex dumps you wrote by hand last module.

Interview angle

"*`List.of` versus `Arrays.asList` versus `Collections.unmodifiableList`?*" Immutable & null-hostile; fixed-size mutable view over the array; live read-only view over a list that can still change underneath.

"*`strip` versus `trim`?*" `strip` is Unicode-aware (`Character.isWhitespace`); `trim` removes ≤ `U+0020` only.

"What is a helpful NPE?" The Java 14+ message describing exactly which expression was null.

"How would you call an HTTP API in plain Java?" `java.net.http.HttpClient` (11) — sync `send` or async `sendAsync`, HTTP/2.

"What did Java 21 add to `List`?" `getFirst`, `getLast`, `addFirst`, `addLast`, `removeFirst`, `removeLast`, `reversed()` via `SequencedCollection`.

Key takeaways

`List.of` / `Set.of` / `Map.of` / `copyOf`: immutable, null-hostile, compact; mutators throw.

Strings: `strip`, `isBlank`, `lines`, `repeat`, `indent`, `transform`, `stripIndent`, `formatted`.

Streams: `toList`, `takeWhile` / `dropWhile`, finite `iterate`, `teeing`, `mapMulti`, `Predicate.not`.

`Objects.requireNonNullElse`; helpful NPE messages; `HttpClient`; `transferTo`; `readString` / `writeString`.

21: sequenced collections (`getFirst` / `getLast` / `reversed`), `Math.clamp`, `HexFormat` (17).

EXERCISE 1

Text block invoice

Read a customer name, an integer `n`, and `n` items `name qty price`. Print an invoice built from a **text block** and `formatted`:

```
Invoice for <name>
<a line of - as long as the line above>
<item> x<qty> @ <price, 2 dp> = <line total, 2 dp>
…
Total: <sum, 2 dp>
```

Use `"-".repeat(...)` for the rule and `Locale.ROOT` formatting (`String.format(Locale.ROOT, …)` for the numbers; `formatted` uses the default locale, so keep it for the text parts).

Example: `Ada`, `2`, `pen 3 1.5`, `pad 1 4` →

```
Invoice for Ada
---------------
pen x3 @ 1.50 = 4.50
pad x1 @ 4.00 = 4.00
Total: 8.50
```

INPUT 1

```
Ada
2
pen 3 1.5
pad 1 4
```

OUTPUT 1

```
Invoice for Ada
---------------
pen x3 @ 1.50 = 4.50
pad x1 @ 4.00 = 4.00
Total: 8.50
```

INPUT 2

```
Bo
0
```

OUTPUT 2

```
Invoice for Bo
--------------
Total: 0.00
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String name = in.next();
        int n = in.nextInt();
        StringBuilder items = new StringBuilder();
        double total = 0;
        for (int i = 0; i < n; i++) {
            String item = in.next();
            int qty = in.nextInt();
            double price = in.nextDouble();
            double line = qty * price;
            total += line;
            items.append(String.format(Locale.ROOT, "%s x%d @ %.2f = %.2f%n", item,
        }
        String template = """
            Invoice for %s
            %s
            %sTotal: %s""";
        String header = "Invoice for " + name;
        System.out.println(template.formatted(name, "-".repeat(header.length()), ite
    }
}
```

EXERCISE 2

The factories say no

Read an integer `n` and `n` words, where `-` stands for `null`. Then:

Try `Set.of(words...)`: print `set=ok size=<n>`, or on `IllegalArgumentException` print `set=<the exception message>` (it reads `duplicate element: x`), or on `NullPointerException` print `set=null rejected`.

Build a mutable `ArrayList`, then `List.copyOf` it (skip if any element is null — print `copy=skipped`); try `copy.add("z")` and print `copy=immutable` if it throws `UnsupportedOperationException`.

Print `asList=<result of Arrays.asList(array).set(0, "q") — "set ok">` and then try `.add("z")` on that list, printing `asList=fixed-size` when it throws.

Example: `3` then `a b a` →

```
set=duplicate element: a
copy=immutable
asList=set ok
asList=fixed-size
```

INPUT 1

```
3
a b a
```

OUTPUT 1

```
set=duplicate element: a
copy=immutable
asList=set ok
asList=fixed-size
```

INPUT 2

```
2
x y
```

OUTPUT 2

```
set=ok size=2
copy=immutable
asList=set ok
asList=fixed-size
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String[] words = new String[n];
        for (int i = 0; i < n; i++) { String w = in.next(); words[i] = w.equals("-")

        try {
            Set<String> set = Set.of(words);
            System.out.println("set=ok size=" + set.size());
        } catch (IllegalArgumentException e) {
            System.out.println("set=" + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("set=null rejected");
        }

        boolean hasNull = Arrays.asList(words).contains(null);
        if (hasNull) {
            System.out.println("copy=skipped");
        } else {
            List<String> copy = List.copyOf(new ArrayList<>(Arrays.asList(words)));
            try { copy.add("z"); System.out.println("copy=mutable"); }
            catch (UnsupportedOperationException e) { System.out.println("copy=immut
        }

        List<String> view = Arrays.asList(words.clone());
        view.set(0, "q");
        System.out.println("asList=set ok");
        try { view.add("z"); System.out.println("asList=grew"); }
        catch (UnsupportedOperationException e) { System.out.println("asList=fixed-s
    }
}
```

QUIZ

1. `List.of(1, 2).add(3)` …

A. Returns a new list

B. Throws `UnsupportedOperationException` — the factories are immutable

C. Appends 3

D. Returns `false`

2. `Arrays.asList(arr)` differs from `List.of(arr)` in that it…

A. Is immutable

B. Is a fixed-size **mutable view** — `set` writes through to the array; `add` throws

C. Allows `add`

D. Copies the array

3. `" x\u2003".strip()` versus `.trim()`:

A. Identical

B. `strip` removes the Unicode space (`Character.isWhitespace`); `trim` removes only chars ≤ U+0020

C. `trim` is Unicode-aware

D. Both throw

4. A helpful `NullPointerException` (Java 14+) message tells you…

A. The line number only

B. Exactly which expression was null, e.g. `because the return value of "Person.name()" is null`

C. The thread name

D. Nothing new

5. Java 21's `SequencedCollection` adds to `List` …

A. `sort`

B. `getFirst` , `getLast` , `addFirst` , `addLast` , `removeFirst` , `removeLast` , `reversed()`

C. `stream`

D. `toArray`

6. The modern replacement for `HttpURLConnection` is…

A. `java.net.Socket`

B. `java.net.http.HttpClient` (Java 11): HTTP/2, sync `send` and async `sendAsync`

C. `URL.openStream` only

D. A third-party library is required

ANSWER KEY · THE API ADDITIONS WORTH KNOWING, 9 TO 21

1. B They also reject `null` elements, and `Set.of` rejects duplicates.

2. B `Collections.unmodifiableList` is a third thing: a read-only view over a list that may still change underneath.

3. B `isBlank`, `stripLeading` / `stripTrailing` follow the same rule.

4. B On by default since 15; locals are named when compiled with `-g`.

5. B `reversed()` is a live view; `getLast()` on an empty list throws `NoSuchElementException`.

6. B Immutable requests via a builder, `BodyHandlers` for the response body, `CompletableFuture` for async.

LESSON 5 · 13 MIN

Virtual threads and structured concurrency

Java 21's headline feature answers a problem the concurrency module left open: a platform thread is an operating-system thread, costs about a megabyte and a system call, and a server that blocks one per request tops out at a few thousand concurrent requests. The industry's answer was reactive programming — callbacks and `CompletableFuture` chains that never block — at the cost of code nobody could read. **Virtual threads** (Project Loom, JEP 444) keep the blocking style and make threads cheap: millions of them, scheduled by the JVM onto a small pool of carrier threads. **Structured concurrency** (still in preview) then gives concurrent code the shape of a block: subtasks that start together, finish together and fail together. This lesson is conceptual — the exercises run on a Java 18 runtime, where none of this compiles — and it is what "what's new in Java 21?" is really asking about.

What a virtual thread is

A virtual thread is a `java.lang.Thread` whose stack lives on the **heap** as a small, growable object rather than in a fixed megabyte of native stack. The JVM runs virtual threads on a `ForkJoinPool` of **carrier** platform threads, one per core. When a virtual thread blocks — on a socket read, `Thread.sleep`, a `BlockingQueue.take`, a lock — the JVM **unmounts** it from its carrier, parks the tiny stack, and mounts another virtual thread. Blocking becomes free; the carrier is never idle. A million virtual threads sleeping costs a few hundred megabytes; a million platform threads is impossible.

```
Thread vt = Thread.ofVirtual().name("req-", 0).start(() → handle(request));
Thread.startVirtualThread(() → …);
try (ExecutorService pool = Executors.newVirtualThreadPerTaskExecutor()) {
    for (Request r : requests) pool.submit(() → handle(r));      // one virtu
}                                                                  // close()
```

The API is the same `Thread`. Existing blocking code — JDBC, `HttpClient.send`, `InputStream.read` — runs unchanged and scales. Virtual threads are always **daemon** and have no meaningful priority.

When they help, and when they do not

Virtual threads make **blocking I/O** cheap; they do not make **CPU work** faster — there are still only as many cores. The rule from the JEP: *never pool virtual threads*; create one per task, and let the JVM schedule. They are a win for servers handling many concurrent requests that mostly wait (databases, downstream calls), for fan-out (call five services in parallel and wait), and for replacing reactive code with straight-line code. They are neutral for a CPU-bound batch job and irrelevant for a single-threaded CLI.

Two costs to know. **Pinning**: a virtual thread inside a `synchronized` block, or in a native call, cannot unmount — it pins its carrier while blocked. In 21 this means "prefer `ReentrantLock` to `synchronized` around blocking calls"; Java 24 (JEP 491) removed the `synchronized` pinning. `ThreadLocal`: still works, but a million threads each holding a per-thread cache is a million caches — hence **scoped values** (`ScopedValue`, preview in 21, final in 25), an immutable per-call-tree value that child threads inherit and that costs nothing.

Structured concurrency (preview)

Unstructured concurrency is `executor.submit` here, `future.get` there, a forgotten task leaking in between. **Structured concurrency** (JEP 453 and successors) makes the lifetime of subtasks match a block of code, the way a `try` block scopes a resource:

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Subtask<User>  user  = scope.fork(() → fetchUser(id));        // each fo
    Subtask<Order> order = scope.fork(() → fetchOrder(id));
    scope.join().throwIfFailed();                                 // wait fo
    return new Profile(user.get(), order.get());
}                                                                 // leaving
```

The rules: a subtask cannot outlive its scope; if one fails, the others are cancelled (`ShutdownOnFailure`) or the first success wins and the rest are cancelled (`ShutdownOnSuccess`); cancellation propagates through interruption; and a thread dump shows the parent–child tree. Compare the same fan-out with `CompletableFuture.allOf` — which cannot cancel the siblings when one fails and leaves nothing in the stack trace to say which request spawned which task. The API is still evolving (renamed and reshaped in 25), so learn the idea rather than the method names.

How this changes the concurrency answers

"How would you handle 10 000 concurrent requests?" Before 21: an async, non-blocking stack (Netty, WebFlux) with careful pool sizing. Now: virtual threads with plain blocking code, one per request.

"How do you size a thread pool?" For virtual threads you do not; for CPU-bound platform pools, still about the core count.

"Is `synchronized` still fine?" Yes for short critical sections; around blocking I/O on virtual threads prefer `ReentrantLock` (until 24).

"What replaces `ThreadLocal`?" `ScopedValue` for request context passed down a call tree.

Migration checklist

Start with the server framework's switch (Spring Boot 3.2+: `spring.threads.virtual.enabled=true`; Helidon, Quarkus, Tomcat and Jetty have equivalents).

Find `synchronized` around blocking calls; consider `ReentrantLock`.

Find `ThreadLocal`s that cache heavy objects; bound them or move to scoped values.

Remove thread-pool tuning that no longer means anything; keep bounded pools where the *downstream* (a database) is the limit — a `Semaphore` is the right limiter now.

Use JFR's `jdk.VirtualThreadPinned` event to find pinning in production.

Interview angle

"What is a virtual thread?" A JVM-scheduled thread with a heap-allocated stack, mounted on a small pool of carrier threads; blocking unmounts it, so millions can block cheaply.

"When do virtual threads not help?" CPU-bound work; and avoid pooling them.

"What is pinning?" A blocked virtual thread that cannot unmount — inside `synchronized` (before 24) or native code — holding its carrier.

"What is structured concurrency?" Subtasks scoped to a block: they finish or are cancelled before the block exits, and failures propagate — `StructuredTaskScope`, preview.

"Which Java?" Virtual threads final in 21; structured concurrency and scoped values preview in 21, scoped values final in 25.

Key takeaways

Virtual threads: same `Thread` API, heap stacks, carriers per core; blocking unmounts. One per task, never pooled.

Big win for I/O-bound concurrency and readability; no gain for CPU-bound work.

Pinning (`synchronized` around blocking, before 24) and heavy `ThreadLocal`s are the two things to audit; `ScopedValue` is the successor.

Structured concurrency scopes subtasks to a block with fail-fast cancellation and a readable thread dump — still preview.

The JDK 21 story: blocking code that scales, without reactive frameworks.

QUIZ

1. A virtual thread differs from a platform thread in that…

A. It runs faster

B. Its stack lives on the heap and the JVM schedules it onto a small pool of carrier threads; blocking unmounts it

C. It cannot block

D. It has no `Thread` API

2. Virtual threads help most with…

A. CPU-bound number crunching

B. I/O-bound work with many concurrent blocking operations

C. Single-threaded scripts

D. Garbage collection

3. The JEP's advice on pooling virtual threads is…

A. Pool them like platform threads

B. Never pool them — create one per task (`newVirtualThreadPerTaskExecutor`)

C. Pool exactly `cores` of them

D. Use `newCachedThreadPool`

4. *Pinning* means…

A. A thread with high priority

B. A blocked virtual thread that cannot unmount — inside `synchronized` (before Java 24) or native code — holding its carrier

C. A thread bound to a CPU core

D. A daemon thread

5. Structured concurrency (`StructuredTaskScope`)…

A. Is final in Java 21

B. Scopes subtasks to a block — they complete or are cancelled before it exits, failures propagate; still preview

C. Replaces threads

D. Is part of `java.io`

6. The successor to `ThreadLocal` for request context in virtual-thread code is…

A. `static` fields

B. `ScopedValue`

C. `synchronized`

D. `InheritableThreadLocal` only

ANSWER KEY · VIRTUAL THREADS AND STRUCTURED CONCURRENCY

1. B Millions can block cheaply; the API is the same `java.lang.Thread`.

2. B There are still only as many cores; the win is that waiting no longer costs a thread.

3. B Limit access to scarce downstream resources with a `Semaphore` instead.

4. B Prefer `ReentrantLock` around blocking calls on 21; JFR's `jdk.VirtualThreadPinned` finds it.

5. B Virtual threads are final in 21; structured concurrency and scoped values were preview there (scoped values final in 25).

6. B Immutable, inherited by child threads, no per-thread copy — preview in 21, final in 25.

LESSON 6 · 13 MIN

Modules, jshell and the modern toolchain

Java 9's biggest change was not a language feature but the **module system** (JPMS), which split the JDK itself into ~70 modules and lets applications declare what they export and require. Most application code still runs on the classpath and never writes a `module-info.java`, yet the module system shapes everything around it: which JDK internals you may touch, why `--add-opens` flags appear in build files, and how `jlink` produces a 40 MB runtime instead of a 300 MB one. Alongside it came a modern toolchain — `jshell` for trying things, single-file execution, `jpackage` for installers, JFR for production profiling. This lesson is the map of that tooling, and the module rules you need to explain in an interview.

The module system in one page

A **module** is a named set of packages with an explicit boundary, declared in `module-info.java` at the root of its source tree:

```
module com.shop.inventory {
    requires java.sql;                          // I use this module's exporte
    requires transitive com.shop.model;         // and whoever requires me car
    exports com.shop.inventory.api;             // only this package is visibl
    exports com.shop.inventory.spi to com.shop.plugins;   // qualified export
    opens com.shop.inventory.entity;            // reflective access (for fra
    uses com.shop.inventory.spi.Pricing;        // ServiceLoader consumer
    provides com.shop.inventory.spi.Pricing with com.shop.inventory.DefaultPri
}
```

Readability: module A can use a type in module B only if A `requires` B (directly, or via a `requires transitive` chain) *and* B `exports` the type's package. **Strong encapsulation**: a package that is not exported is invisible — not even reflection can reach it without `opens`. That is why `sun.misc.Unsafe` tricks and `setAccessible` on JDK internals stopped working (fully enforced in 16), and why frameworks need `--add-opens java.base/java.lang=ALL-UNNAMED` when they insist. `java.base` is required implicitly by every module. The **unnamed module** is the classpath: everything on it reads all modules and exports everything, so legacy code runs unchanged.

Why bother: reliable configuration (missing modules fail at start-up, not at first use), encapsulation of internals, and `jlink` — assemble a custom runtime image containing only

the modules your application needs: `jlink --add-modules java.base,java.sql --output runtime` yields a small, fast-starting JRE for a container. `jdeps --list-deps app.jar` tells you which modules a jar needs. Automatic modules (a plain jar on the module path, named after its file) bridge unmodularised libraries.

Practical advice: modularise libraries and the JDK; leave most applications on the classpath unless you need `jlink`. Know the vocabulary — `requires`, `exports`, `opens`, `transitive`, unnamed module, split packages (two modules with the same package is an error) — because build errors and framework documentation assume it.

`jshell`: a REPL for Java

`jshell` (9) is an interactive shell: type expressions and see results without a class or a `main`.

```
jshell> var xs = List.of(3, 1, 2)
xs ⟹ [3, 1, 2]
jshell> xs.stream().sorted().toList()
$2 ⟹ [1, 2, 3]
jshell> /methods
jshell> /open Util.java
```

Use it to check an API's behaviour, try a regex, or verify what `Integer.valueOf(128) == Integer.valueOf(128)` prints before saying it in an interview. Semicolons are optional at the top level; `/vars`, `/list`, `/edit` and tab completion are the commands you need.

Running without a build: single-file and beyond

Single-file source launch (11): `java Hello.java` compiles in memory and runs. Scripts and interview take-home tasks stop needing a project.

Shebang files (11): put `#!/usr/bin/java --source 21` at the top and `chmod +x` — a Java script.

Multi-file source launch (22) extends this to a directory of sources.

Unnamed classes and instance `main` (preview 21, final 25 as *compact source files*): `void main() { System.out.println("hi"); }` is a complete program — the on-ramp for beginners that removes `public static void main(String[] args)` and `class` from the first lesson.

The command-line tools

Tool	Does
`javac --release N`	compile against exactly JDK N's API and syntax
`java -XX:+PrintFlagsFinal`, `-Xlog:gc`, `-XX:+HeapDumpOnOutOfMemoryError`	the runtime flags from the memory module
`jar`, `jar --describe-module`	package; inspect a modular jar
`jdeps`	dependency analysis: which modules and, with `--jdk-internals`, which forbidden internals a jar uses
`jlink`	custom runtime image
`jpackage` (16)	native installers (`.msi`, `.dmg`, `.deb`) bundling your app and a runtime
`jcmd`, `jstack`, `jmap`, `jstat`, `jinfo`	live diagnostics: thread dumps, heap dumps, GC stats
`jfr` + JDK Mission Control	Flight Recorder: always-on low-overhead production profiling (`-XX:StartFlightRecording`)
`jwebserver` (18)	a static file server for local testing
`javadoc`, with `{@snippet}` (18)	documentation with compiled, validated code examples

Build tools and libraries, briefly

You will build with **Maven** (`pom.xml`, convention over configuration, the default in most companies) or **Gradle** (Kotlin/Groovy DSL, faster incremental builds, Android). Both resolve dependencies from Maven Central, run tests (JUnit 5 is the standard), and apply `--release`. For measurement, **JMH** is the benchmark harness (the memory module explained why nothing else is trustworthy). For quality gates: `-Xlint:all`, Error Prone or SpotBugs, and `jdeps --jdk-internals` in CI to catch reliance on internals before a JDK upgrade does.

Upgrading a JDK: what actually breaks

From 8 to 11+: removed `javax.xml.bind` / `javax.activation` /CORBA (add dependencies), illegal reflective access to internals (find with `--illegal-access=warn` on 11–15, then fix or `--add-opens`), tools that parsed the version string as `1.8`. From 11 to 17: strong encapsulation enforced, `SecurityManager` deprecated, some `sun.*` APIs gone. From 17 to 21: little; check libraries for `synchronized`-heavy code if you adopt virtual threads. The habit that makes upgrades cheap is to compile with `--release` and run `jdeps --jdk-internals` regularly.

Interview angle

"What is the Java module system?" Named modules with explicit `requires` / `exports`; strong encapsulation of non-exported packages; reliable configuration; enables `jlink`.

"`requires` versus `requires transitive`?" Transitive re-exports the dependency to whoever requires you.

"What is the unnamed module?" The classpath: reads everything, exports everything — how legacy code keeps running.

"Why do frameworks need `--add-opens`?" To reflect into packages that a module does not `opens`; strong encapsulation is enforced since 16.

"What is `jshell` for?" A REPL to try API calls and expressions instantly.

Key takeaways

`module-info.java`: `requires`, `requires transitive`, `exports`, `exports … to`, `opens`, `uses` / `provides`; readability + exports = accessibility.

Strong encapsulation broke `sun.*` hacks; `--add-opens` is the escape hatch, `jdeps --jdk-internals` the detector.

`jlink` builds small runtimes; `jpackage` builds installers; most apps stay on the classpath (the unnamed module).

`jshell` to try things; `java File.java` to run one file; instance `main` for the smallest program.

`--release N`, JFR, `jcmd` family, Maven/Gradle, JUnit 5, JMH: the toolchain a modern Java developer is assumed to know.

EXERCISE

Resolve the module graph

Read an integer `m` and `m` module declarations `<name>: <deps>`, where `deps` is a comma-separated list (or `-`) and a dependency prefixed with `+` is `requires transitive`. Then read `q` and `q` module names. A module **reads** its direct dependencies, plus — for every module it reads — that module's `requires transitive` dependencies, recursively along transitive edges. Print `<name> reads <sorted, comma-separated>` or `<name> reads -`. Ignore `java.base`.

Example

```
4
app: web,+model
web: +http,util
http: -
model: -
1
app
```

→

```
app reads http,model,web
```

(`app` reads `web` and `model` directly, and `http` because `web` re-exports it — but not `util`.)

INPUT 1

```
4
app: web,+model
web: +http,util
http: -
model: -
1
app
```

OUTPUT 1

```
app reads http,model,web
```

INPUT 2

```
3
a: +b
b: +c
c: -
2
a
c
```

OUTPUT 2

```
a reads b,c
c reads -
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static Map<String, List<String>> direct = new HashMap<>();
    static Map<String, List<String>> transitive = new HashMap<>();

    static void addTransitive(String module, Set<String> out) {
        for (String t : transitive.getOrDefault(module, List.of())) {
            if (out.add(t)) addTransitive(t, out);
        }
    }

    static Set<String> reads(String module) {
        Set<String> out = new TreeSet<>();
        for (String d : direct.getOrDefault(module, List.of())) {
            out.add(d);
            addTransitive(d, out);
        }
        return out;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int m = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < m; i++) {
            String[] parts = in.nextLine().split(":");
            String name = parts[0].trim();
            direct.put(name, new ArrayList<>());
            transitive.put(name, new ArrayList<>());
            for (String dep : parts[1].trim().split(",")) {
                dep = dep.trim();
                if (dep.isEmpty() || dep.equals("-")) continue;
                if (dep.startsWith("+")) { transitive.get(name).add(dep.substring(1)
                else direct.get(name).add(dep);
            }
        }
        int q = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < q; i++) {
            String name = in.nextLine().trim();
            Set<String> r = reads(name);
            System.out.println(name + " reads " + (r.isEmpty() ? "-" : String.join('
        }
    }
}
```

QUIZ

1. Module A can use a public type in module B's package only if…

A. B is on the classpath

B. A `requires` B (directly or via `requires transitive`) **and** B `exports` that package

C. The type is `public`

D. A `opens` the package

2. `requires transitive M` means…

A. M is optional

B. Modules that require me also read M

C. M is loaded lazily

D. M is a test dependency

3. `--add-opens java.base/java.lang=ALL-UNNAMED` is needed when…

A. Compiling with `--release`

B. Code on the classpath reflects into a JDK package that is not `opens`-ed — strong encapsulation since 16

C. Running `jshell`

D. Using records

4. The *unnamed module* is…

A. `java.base`

B. The classpath: it reads all modules and exports everything, so legacy code keeps running

C. A module without `module-info`

D. An error

5. `jlink` produces…

A. An installer

B. A custom runtime image containing only the modules your application needs

C. A jar

D. Documentation

6. `java Hello.java` (Java 11)…

A. Requires a compiled class

B. Compiles the single source file in memory and runs it

C. Starts `jshell`

D. Only works with modules

ANSWER KEY · MODULES, JSHELL AND THE MODERN TOOLCHAIN

1. B Readability plus exports equals accessibility; `public` alone is no longer enough across modules.

2. B Use it when your exported API mentions M's types.

3. B `jdeps --jdk-internals` finds such uses before an upgrade breaks them.

4. B Jars on the module path without a descriptor become *automatic* modules instead.

5. B `jpackage` (16) wraps that into native installers.

6. B Shebang scripts and, later, multi-file launch and compact source files extend the idea.

CHECKPOINT · 26 MIN

Checkpoint — Modern Java

This checkpoint covers the release cadence and the LTS milestones, `var` and inference, pattern `instanceof`, pattern `switch` and record patterns with exhaustiveness, the library additions from 9 to 21, virtual threads and structured concurrency, and the module system with the modern toolchain.

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:

- Which versions are LTS, and which of records, sealed, `var`, text blocks, pattern switch and virtual threads arrived in each.
- Where `var` is not allowed, and what `var list = new ArrayList<>()` infers.
- How a pattern variable is scoped, and what sealed buys an exhaustive `switch`.
- `List.of` versus `Arrays.asList` versus `Collections.unmodifiableList`; `strip` versus `trim`.
- What a virtual thread is, when it helps, and what pinning means.
- `requires transitive`, `opens`, and what the unnamed module is.

The programs — which compile on the Java 17-level runtime — are a shape calculator over a sealed hierarchy with pattern `instanceof`, a modern text pipeline using the new `String` and `Stream` methods, and an immutable configuration built from the collection factories.

EXERCISE 1

Shape report

Declare `sealed interface Shape permits Circle, Square, Rect` with records `Circle(double r)`, `Square(double side)`, `Rect(double w, double h)`. Read an integer `n` and `n` shapes (`circle r`, `square s`, `rect w h`). Using pattern `instanceof` for the area, print for every kind present, in alphabetical order, `<Kind>: n=<count> area=<total area, 2 dp>`; then `largest=<record toString of the largest shape>` and `kinds=<number of distinct kinds>`. Use `Locale.ROOT`.

Example: `3` then `circle 1`, `rect 2 3`, `circle 2` →

```
Circle: n=2 area=15.71
Rect: n=1 area=6.00
largest=Circle[r=2.0]
kinds=2
```

INPUT 1

```
3
circle 1
rect 2 3
circle 2
```

OUTPUT 1

```
Circle: n=2 area=15.71
Rect: n=1 area=6.00
largest=Circle[r=2.0]
kinds=2
```

INPUT 2

```
1
square 3
```

OUTPUT 2

```
Square: n=1 area=9.00
largest=Square[side=3.0]
kinds=1
```

REFERENCE SOLUTION

```java
import java.util.*;

sealed interface Shape permits Circle, Square, Rect { }
record Circle(double r) implements Shape { }
record Square(double side) implements Shape { }
record Rect(double w, double h) implements Shape { }

public class Main {
    static double area(Shape s) {
        if (s instanceof Circle c) return Math.PI * c.r() * c.r();
        if (s instanceof Square q) return q.side() * q.side();
        if (s instanceof Rect r) return r.w() * r.h();
        throw new IllegalStateException();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, int[]> counts = new TreeMap<>();
        Map<String, double[]> areas = new TreeMap<>();
        Shape largest = null;
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            Shape s = switch (kind) {
                case "circle" -> new Circle(in.nextDouble());
                case "square" -> new Square(in.nextDouble());
                default -> new Rect(in.nextDouble(), in.nextDouble());
            };
            String name = s.getClass().getSimpleName();
            counts.computeIfAbsent(name, k -> new int[1])[0]++;
            areas.computeIfAbsent(name, k -> new double[1])[0] += area(s);
            if (largest == null || area(s) > area(largest)) largest = s;
        }
        for (String kind : counts.keySet()) {
            System.out.println(String.format(Locale.ROOT, "%s: n=%d area=%.2f", kinc
        }
        System.out.println("largest=" + largest);
        System.out.println("kinds=" + counts.size());
    }
}
```

EXERCISE 2

Word bars

Read all of standard input. Build one modern pipeline: `lines()` → `strip` → drop blank lines with `Predicate.not(String::isBlank)` → split into words → lower-case → count with `groupingBy(…,`

`counting())`. Print the **top 5** words (most frequent first, ties alphabetical) as `"%-8s %s %d".formatted(word, "#".repeat(count), count)`, then `distinct=<distinct words>`.

Example: `the cat the dog` / `The end` →

```
the      ### 3
cat      # 1
dog      # 1
end      # 1
distinct=4
```

INPUT 1

```
the cat the dog
The end
```

OUTPUT 1

```
the      ### 3
cat      # 1
dog      # 1
end      # 1
distinct=4
```

INPUT 2

```
a b c d e f g
```

OUTPUT 2

```
a        # 1
b        # 1
c        # 1
d        # 1
e        # 1
distinct=7
```

REFERENCE SOLUTION

```java
import java.util.*;
import java.util.function.*;
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());
        Map<String, Long> counts = text.lines()
            .map(String::strip)
            .filter(Predicate.not(String::isBlank))
            .flatMap(l -> Arrays.stream(l.split("\\s+")))
            .map(String::toLowerCase)
            .collect(groupingBy(w -> w, counting()));
        List<Map.Entry<String, Long>> top = counts.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue(Comparator.reverseOrder
            .limit(5)
            .toList();
        for (Map.Entry<String, Long> e : top) {
            System.out.println("%-8s %s %d".formatted(e.getKey(), "#".repeat(e.getVa
        }
        System.out.println("distinct=" + counts.size());
    }
}
```

EXERCISE 3

Immutable configuration

Read `key=value` lines until end of input (skip blanks) into a mutable map, then freeze it with `Map.copyOf`. Print `port=<value or 8080>`, `host=<value or localhost>` and `debug=<value or false>` using `Objects.requireNonNullElse` for the defaults; then `keys=<all keys, sorted, comma-separated>`; then try `config.put("x", "y")` and print `frozen=true` if it throws `UnsupportedOperationException`.

Example: `port=9090` / `debug=true` →

```
port=9090
host=localhost
debug=true
keys=debug,port
frozen=true
```

INPUT 1

```
port=9090
debug=true
```

OUTPUT 1

```
port=9090
host=localhost
debug=true
keys=debug,port
frozen=true
```

INPUT 2

```

```

OUTPUT 2

```
port=8080
host=localhost
debug=false
keys=
frozen=true
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes());
        Map<String, String> mutable = new HashMap<>();
        for (String line : text.lines().map(String::strip).filter(l -> !l.isBlank())
            int eq = line.indexOf('=');
            mutable.put(line.substring(0, eq).strip(), line.substring(eq + 1).strip(
        }
        Map<String, String> config = Map.copyOf(mutable);
        System.out.println("port=" + Objects.requireNonNullElse(config.get("port"),
        System.out.println("host=" + Objects.requireNonNullElse(config.get("host"),
        System.out.println("debug=" + Objects.requireNonNullElse(config.get("debug")
        System.out.println("keys=" + String.join(",", new TreeSet<>(config.keySet())
        try {
            config.put("x", "y");
            System.out.println("frozen=false");
        } catch (UnsupportedOperationException e) {
            System.out.println("frozen=true");
        }
    }
}
```

CHECKPOINT QUESTIONS

1. Which are LTS releases?

A. 9, 12, 15

B. 8, 11, 17, 21

C. 10, 14, 18

D. Every release

2. Sealed classes became final in…

A. Java 15

B. Java 16

C. Java 17

D. Java 21

3. `var` cannot be used for…

A. Loop variables

B. Fields and method parameters

C. Locals with initialisers

D. Try-with-resources variables

4. Predict: `var s = "hi"; var l = List.of();` — the type of `l` is…

A. `List<String>`

B. `List<Object>`

C. Compile error

D. `ArrayList<Object>`

5. After `if (o instanceof Integer i && i > 3) { … }`, `i` is in scope…

A. Everywhere in the method

B. Inside the `if` body and on the right of the `&&` only

C. Only in the condition

D. Nowhere

6. An exhaustive pattern `switch` over a sealed type needs `default`…

A. Always

B. Never — unless some permitted subtype is not covered

C. Only for records

D. Only in statements

7. Record patterns let you…

A. Create records

B. Deconstruct a record into its components in `case` or `instanceof`, nesting as deep as the data goes

C. Serialise records

D. Extend records

8. `Set.of("a", "a")`…

A. Returns a one-element set

B. Throws `IllegalArgumentException: duplicate element: a`

C. Returns an empty set

D. Returns a list

9. `Stream.toList()` (16) compared to `collect(Collectors.toList())` …

A. Is identical

B. Returns an unmodifiable list and is shorter; `Collectors.toList()` makes no immutability promise

C. Returns an `ArrayList`

D. Is slower

10. `"abc".repeat(0)` returns…

A. `"abc"`

B. `""`

C. `null`

D. Throws

11. Virtual threads are final in…

A. Java 17

B. Java 19

C. Java 21

D. Java 25

12. Virtual threads do **not** speed up…

A. Many blocking HTTP calls

B. CPU-bound computation

C. JDBC-heavy request handling

D. Fan-out to several services

13. A package that a module does not `exports` …

A. Is visible to everyone

B. Is inaccessible to other modules — even via reflection unless `opens`

C. Is visible to the classpath only

D. Cannot contain classes

14. `jshell` is…

A. A build tool

B. A REPL for trying Java expressions and API calls interactively

C. A profiler

D. A debugger

15. Java 18 changed the default charset to…

A. Latin-1

B. UTF-8 everywhere (JEP 400)

C. UTF-16

D. The platform's

ANSWER KEY · CHECKPOINT — MODERN JAVA

1. B Plus 25; production runs LTS with multi-year vendor support.

2. C Previewed in 15 and 16, final in the 17 LTS.

3. B Nor without an initialiser, with `null`, or with a bare lambda.

4. B No target type, so the generic method infers `Object`.

5. B Flow scoping: wherever the match is definitely true.

6. B Adding a permitted subtype makes every such switch fail to compile until handled.

7. B Final in 21; `var` is allowed for component types.

8. B The factories are strict: duplicates and nulls throw.

9. B Use `toCollection(ArrayList::new)` when you need to mutate.

10. B Zero repetitions is the empty string; a negative count throws `IllegalArgumentException`.

11. C Previewed in 19 and 20; structured concurrency remained preview.

12. B They make waiting cheap, not computing.

13. B Strong encapsulation; `--add-opens` is the run-time escape hatch.

14. B Check what `Integer.valueOf(128) == Integer.valueOf(128)` prints before saying it in an interview.

15. B Removing the classic 'works on my machine' encoding bug — still name the charset explicitly.