Module 18 of 20

I/O, files and time

Byte and character streams with charsets, buffering and the fast-I/O template, Path and Files, serialization and CSV done right, and java.time — dates, zones, durations and formatting.

Every program eventually talks to the outside world: it reads input faster than a Scanner can, writes a report, walks a directory, parses a CSV somebody exported from a spreadsheet, and gets a date wrong across a time zone. This module is the practical layer of Java that interviews probe with "how would you read a 10 GB file" and that judges probe with a two-second time limit.

It starts with the model behind java.io — byte streams, character streams, the charset between them, and the decorator stack — then gives you the fast-I/O template and the reasons it is fast. The java.nio.file lesson covers Path arithmetic (resolve, normalize, relativize) and the Files one-liners that replaced fifteen lines of java.io.File. Serialization is explained honestly: the mechanism, its knobs, and why the platform's own architects say to use JSON instead — followed by the CSV parser that split(",") is not. The module closes with java.time: the right type for each question, month arithmetic that clamps, zone conversion, Period versus Duration, and formatting with an explicit Locale.

The exercises are the tools themselves: a hex dump, a fast reader, a quote-aware CSV parser, a log analyser measuring spans in java.time.

CONTENTS

1	Bytes and characters — streams, readers and encodings	14 min
2	Buffering and fast I/O — the template that passes time limits	13 min
3	Files and paths — java.nio.file	14 min
4	Serialization, CSV and the formats you actually use	14 min
5	java.time — dates, times, zones and durations	15 min
✓	Checkpoint — I/O, files and time	28 min

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

LESSON 1 · 14 MIN

Bytes and characters — streams, readers and encodings

Java's I/O library looks enormous — sixty-odd classes in `java.io` alone — and is built from three ideas. **Byte streams** (`InputStream` / `OutputStream`) move raw bytes. **Character streams** (`Reader` / `Writer`) move text, and the bridge between the two is a **charset**, the rule for turning characters into bytes. And every stream class is a **decorator**: you wrap a raw source in layers that add buffering, encoding, or typed reads. Once those three ideas are in place the sixty classes are four small families, and the encoding bugs that fill Stack Overflow — mojibake, "unmappable character", wrong byte counts — become predictable.

Two hierarchies

	Bytes	Characters
read	`InputStream` — `int read()`, `int read(byte[])`, `byte[] readAllBytes()`	`Reader` — `int read()`, `int read(char[])`, `String readLine()` via `BufferedReader`
write	`OutputStream` — `write(int)`, `write(byte[])`, `flush()`, `close()`	`Writer` — `write(String)`, `append`, `flush()`, `close()`
sources	`FileInputStream`, `ByteArrayInputStream`, `System.in`, sockets	`FileReader`, `StringReader`, `InputStreamReader`
sinks	`FileOutputStream`, `ByteArrayOutputStream`, `System.out` (a `PrintStream`)	`FileWriter`, `StringWriter`, `OutputStreamWriter`, `PrintWriter`

`read()` on either returns the next unit as an `int` — a byte 0–255 or a `char` 0–65535 — and **−1 at end of stream**, which is why it returns `int` and not `byte`: −1 must be distinguishable from the byte `0xFF`. A file, a network socket and `System.in` are byte streams; text is a *view* over them that a `Reader` provides once you tell it the charset.

Charsets: where text meets bytes

A `String` is a sequence of UTF-16 code units in memory (compact Latin-1 internally since Java 9, but that is invisible). A file or a socket holds bytes. The mapping is a `Charset`:

```
byte[] utf8 = "héllo".getBytes(StandardCharsets.UTF_8);     // 6 bytes: h=68,
byte[] latin = "héllo".getBytes(StandardCharsets.ISO_8859_1); // 5 bytes: é =
String back = new String(utf8, StandardCharsets.UTF_8);      // "héllo"
String wrong = new String(utf8, StandardCharsets.ISO_8859_1); // "hÃ©llo" — m
```

UTF-8 encodes ASCII in one byte, most European letters in two, CJK in three, emoji in four. **Always name the charset.** Before Java 18 `getBytes()`, `new String(bytes)`, `FileReader` and `FileWriter` used the *platform default* — UTF-8 on Linux, Windows-1252 on many Windows machines — so the same program wrote different bytes on different machines. Java 18 (JEP 400) made UTF-8 the default everywhere, which removes the trap for new code, but code that must run on older JDKs, and any interview answer, still says `StandardCharsets.UTF_8` explicitly.

`s.length()` counts UTF-16 units, `s.getBytes(UTF_8).length` counts bytes, `s.codePointCount(0, s.length())` counts characters as a human would — three different numbers for the same emoji-laden string.

The decorator pattern in practice

```
try (BufferedReader in = new BufferedReader(new InputStreamReader(new FileInpu
    String line;
    while ((line = in.readLine()) ≠ null) process(line);
}
```

Read it inside out: a `FileInputStream` produces bytes; an `InputStreamReader` decodes them as UTF-8 into characters; a `BufferedReader` batches reads into 8 KB chunks and adds `readLine()`. Each layer has the same interface as the one it wraps, so you can add or remove layers freely — that is the pattern. Closing the outermost closes the chain. The symmetrical output stack is `PrintWriter(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(f), UTF_8)))`, and `Files.newBufferedReader(path)` / `Files.newBufferedWriter(path)` build the common stacks in one call (UTF-8 by default).

`System.in` is a raw `InputStream`; `System.out` is a `PrintStream` — a byte stream that knows how to format text, which is why `println` works on it. `new Scanner(System.in)` and `new BufferedReader(new InputStreamReader(System.in))` are the two text views over it you have been using since module 1.

In-memory streams: testing and building

`ByteArrayOutputStream` collects everything written to it into a growing `byte[]` (`toByteArray()`, `toString(UTF_8)`); `ByteArrayInputStream` serves a `byte[]` as a stream. `StringWriter` / `StringReader` are the character twins. They let you test any code written against `InputStream` / `Reader` without a file, and they are how you build a byte image in memory — a serialised object, an encoded message — before sending it anywhere.

Typed binary I/O

`DataOutputStream` / `DataInputStream` write and read primitives in a fixed **big-endian** binary form: `writeInt(258)` emits the four bytes `00 00 01 02`; `writeUTF` writes a length-prefixed modified-UTF-8 string. `ObjectOutputStream` (lesson 4) sits on top for whole object graphs. Reading with the wrong method (`readInt` where `readLong` was written) does not fail — it silently produces garbage, so binary formats need a header and a version.

Closing, flushing and try-with-resources

Every stream holds a native resource — a file descriptor, a socket — and must be closed. `try-with-resources` (`try (var in = …) { }`) closes in reverse order of opening even when an exception escapes, and adds any exception from `close()` as a *suppressed* exception rather than losing the original. Writers and buffered output streams hold data in memory until `flush()` or `close()`: a program that writes a file and exits without closing may write nothing at all. `System.out` flushes on `println` only when it is a console; redirected to a file it buffers, which matters for the output-heavy exercises of the next lesson.

`IOException` is checked

Almost every I/O method throws `IOException`, a checked exception, because the outside world fails in ways the program cannot prevent (disk full, connection reset). Handle it where you can do something (retry, report), otherwise let it propagate with a `throws` clause; wrap it in `UncheckedIOException` when an API that takes a lambda (`Stream.map`) forbids checked exceptions.

Interview angle

"Byte stream versus character stream?" `InputStream` / `OutputStream` move bytes; `Reader` / `Writer` move characters; `InputStreamReader` / `OutputStreamWriter` bridge them with a charset.

"Why does `read()` return `int`?" To return −1 at end of stream without clashing with byte `0xFF`.

"What is the decorator pattern in `java.io`?" Streams wrap streams with the same interface, adding buffering, encoding or typed reads.

"What happens if you forget the charset?" Before Java 18, the platform default — different bytes on different machines; always pass `StandardCharsets.UTF_8`.

"`length()` versus `getBytes().length`?" UTF-16 units versus encoded bytes; they differ for anything outside ASCII.

Key takeaways

Bytes: `InputStream` / `OutputStream`. Characters: `Reader` / `Writer`. `read()` gives −1 at the end.

A charset maps text to bytes; UTF-8 is the answer; name it, even on Java 18+.

Decorate: raw source → `InputStreamReader` → `BufferedReader`; `Files.newBufferedReader` for the common stack.

`ByteArrayOutputStream` / `StringWriter` for in-memory work and tests; `DataOutputStream` for big-endian primitives.

Close with try-with-resources; buffered output is not written until flushed or closed.

EXERCISE 1

Count the bytes

Read **all** of standard input as bytes, decode it as UTF-8 and strip the trailing line break. Print `chars=<length()> codePoints=<codePointCount> utf8=<number of UTF-8 bytes>` and then `hex=<the UTF-8 bytes as two-digit lower-case hex, space-separated>`. The three counts agree for ASCII and drift apart as soon as a character needs more than one byte.

Example: `héllo` →

```
chars=5 codePoints=5 utf8=6
hex=68 c3 a9 6c 6c 6f
```

INPUT 1

```
héllo
```

OUTPUT 1

```
chars=5 codePoints=5 utf8=6
hex=68 c3 a9 6c 6c 6f
```

INPUT 2

```
Hi!
```

OUTPUT 2

```
chars=3 codePoints=3 utf8=3
hex=48 69 21
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.nio.charset.StandardCharsets;

public class Main {
    public static void main(String[] args) throws IOException {
        byte[] raw = System.in.readAllBytes();
        String s = new String(raw, StandardCharsets.UTF_8).stripTrailing();
        byte[] utf8 = s.getBytes(StandardCharsets.UTF_8);
        System.out.println("chars=" + s.length() + " codePoints=" + s.codePointCount
        StringBuilder hex = new StringBuilder();
        for (byte b : utf8) {
            if (hex.length() > 0) hex.append(' ');
            hex.append(String.format("%02x", b & 0xff));
        }
        System.out.println("hex=" + hex);
    }
}
```

EXERCISE 2

Big-endian round trip

Read an integer `n` and `n` integers. Write them with a `DataOutputStream` over a `ByteArrayOutputStream`, then print `bytes=<size of the buffer>` and `hex=<the buffer in two-`

`digit hex, space-separated>`. Finally read the buffer back through a `DataInputStream` over a `ByteArrayInputStream` and print `readBack=<Arrays.toString of the ints>`. Every `int` is four bytes, most significant first.

Example: `2` then `258 -1` →

```
bytes=8
hex=00 00 01 02 ff ff ff ff
readBack=[258, -1]
```

INPUT 1

```
2
258 -1
```

OUTPUT 1

```
bytes=8
hex=00 00 01 02 ff ff ff ff
readBack=[258, -1]
```

INPUT 2

```
1
0
```

OUTPUT 2

```
bytes=4
hex=00 00 00 00
readBack=[0]
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] values = new int[n];
        for (int i = 0; i < n; i++) values[i] = in.nextInt();
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        try (DataOutputStream out = new DataOutputStream(buffer)) {
            for (int v : values) out.writeInt(v);
        }
        byte[] bytes = buffer.toByteArray();
        System.out.println("bytes=" + bytes.length);
        StringBuilder hex = new StringBuilder();
        for (byte b : bytes) hex.append(hex.length() > 0 ? " " : "").append(String.f
        System.out.println("hex=" + hex);
        int[] back = new int[n];
        try (DataInputStream din = new DataInputStream(new ByteArrayInputStream(byte
            for (int i = 0; i < n; i++) back[i] = din.readInt();
        }
        System.out.println("readBack=" + Arrays.toString(back));
    }
}
```

QUIZ

1. `InputStream.read()` returns an `int` rather than a `byte` because…

A. Bytes are slow

B. It must return −1 at end of stream, distinct from the byte value 0xFF

C. `byte` cannot be returned

D. Historical accident

2. `new String(bytes)` without a charset on Java 17…

A. Always decodes UTF-8

B. Uses the platform default — different machines may decode differently

C. Throws

D. Decodes Latin-1

3. The correct order for reading a UTF-8 text file line by line is…

A. `BufferedReader` → `FileInputStream` → `InputStreamReader`

B. `FileInputStream` wrapped in `InputStreamReader(UTF_8)` wrapped in `BufferedReader`

C. `FileReader` only

D. `Scanner` over `DataInputStream`

4. Predict: `"€".length()` and `"€".getBytes(UTF_8).length`

A. `1` and `1`

B. `1` and `3`

C. `3` and `3`

D. `2` and `1`

5. A program writes to a `BufferedWriter` and exits without closing it. The file…

A. Is complete

B. May be empty or truncated — buffered data is written on `flush` / `close`

C. Is corrupted

D. Is locked

ANSWER KEY · BYTES AND CHARACTERS — STREAMS, READERS AND ENCODINGS

1. B 0–255 for data, −1 for end; the same convention holds for `Reader.read()`.

2. B Java 18 made UTF-8 the default (JEP 400); explicit `StandardCharsets.UTF_8` is still the habit.

3. B Bytes → decoded characters → buffering with `readLine()`; `Files.newBufferedReader` builds the same stack.

4. B One UTF-16 unit; three UTF-8 bytes. An emoji would be 2 and 4.

5. B Try-with-resources closes (and therefore flushes) even when an exception escapes.

LESSON 2 · 13 MIN

Buffering and fast I/O — the template that passes time limits

A `Scanner` reading a million integers takes several seconds; a `BufferedReader` with a `StringTokenizer` does it in a fraction of one; a hand-rolled byte reader is faster still. `System.out.println` in a loop of a million lines is equally slow. None of this matters for a five-line program and all of it decides whether a correct solution passes a judge's time limit. This lesson explains *why* buffering matters, gives the standard fast-I/O template every competitive Java programmer carries, and covers `PrintWriter`, `printf` and the flush rules that catch people out.

Why unbuffered I/O is slow

Every `read()` on a raw `FileInputStream` or `System.in` can be a **system call** — a trip into the kernel that costs about a microsecond. Reading bytes one at a time is a million system calls for a megabyte. A `BufferedInputStream` / `BufferedReader` asks the kernel for 8 KB at once and serves the next 8 000 reads from memory. The same on output: `BufferedWriter` collects text and writes it in chunks. Buffering is not an optimisation for large files only; it is the difference between I/O that costs microseconds and I/O that costs nothing measurable.

`Scanner` adds a second cost: it tokenises with **regular expressions** and boxes each result (`nextInt` parses through a `String`), which is several hundred nanoseconds per token. Fine for a hundred numbers; catastrophic for a million.

The standard fast template

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in
        StringTokenizer st = new StringTokenizer(in.readLine());
        int n = Integer.parseInt(st.nextToken());
        PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStream
        for (int i = 0; i < n; i++) {
            while (!st.hasMoreTokens()) st = new StringTokenizer(in.readLine()
            int x = Integer.parseInt(st.nextToken());
            out.println(x * 2);
        }
        out.flush();                // NOTHING is printed until this line
    }
}
```

The `while (!st.hasMoreTokens())` refill is the detail people forget: input that says "n integers" may put them on one line or fifty, and a tokenizer that assumes one line per number reads `null` and throws. Wrap that logic in a `next()` helper and you have a reader that does not care about line structure — exactly how `Scanner` behaves, at a tenth of the cost.

```java
static BufferedReader in = new BufferedReader(new InputStreamReader(System.in
static StringTokenizer st;
static String next() throws IOException {
    while (st == null || !st.hasMoreTokens()) st = new StringTokenizer(in.read
    return st.nextToken();
}
static int nextInt() throws IOException { return Integer.parseInt(next()); }
static long nextLong() throws IOException { return Long.parseLong(next()); }
```

`in.readLine()` returns `null` at end of input; a program that reads "until EOF" loops on `while ((line = in.readLine()) ≠ null)`.

Faster still: a byte reader

For the heaviest inputs (10^7 numbers), even `readLine` + `parseInt` allocates a `String` per line. The competitive-programming answer reads raw bytes and parses digits by hand:

```
static DataInputStream din = new DataInputStream(new BufferedInputStream(Syste
static int readInt() throws IOException {
    int ret = 0, b = din.read();
    while (b ⩽ ' ') b = din.read();                    // skip whitespace
    boolean neg = b == '-';
    if (neg) b = din.read();
    while (b ⩾ '0' && b ⩽ '9') { ret = ret * 10 + (b - '0'); b = din.read()
    return neg ? -ret : ret;
}
```

Zero allocation per number. It is ugly, it is what the top of every Java leaderboard uses, and you should be able to write it from memory — while saying in an interview that in application code you would never need it.

Output: `PrintWriter` and `flush`

`System.out` is a `PrintStream` with autoflush on every `println` when attached to a console — a system call per line. Wrapping it in `PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)))` (or `new PrintWriter(System.out)` — buffered by default, no autoflush) turns a million `println`s into a few hundred writes. The price: **you must `flush()` (or `close()`) at the end**, or the output stays in the buffer and the program prints nothing. A judge that shows "no output" for a program that obviously computes the answer is almost always a missing flush.

Alternatively build the whole output in a `StringBuilder` and print it once: `sb.append(x).append('\n')` in the loop, `System.out.print(sb)` at the end. Same effect, one flush, and no `IOException`s to declare.

`printf` / `String.format` are convenient and slow (a format-string parse per call); in a hot loop prefer `append`. When you do format, pass `Locale.ROOT` or `Locale.US` for decimals — a judge running in a German locale prints `3,14` and fails your output.

Reading everything at once

For small-to-medium inputs the simplest fast approach is one read: `String all = new String(System.in.readAllBytes(), UTF_8)` then `all.split("\\s+")` or `all.lines()`. One system call, one allocation of the whole text, and every parsing tool of the strings module available. Watch the empty-input edge case (`split` on `""` yields `[""]`, not `[]`).

Character-at-a-time when the format demands it

Some inputs are grids of characters, or expressions with no whitespace. `in.read()` on a `BufferedReader` returns one `char` as an `int` (−1 at end); `in.readLine().toCharArray()` gives a row of a grid. `Scanner` has no `nextChar` — `next().charAt(0)` is the workaround — another reason the reader wins.

Interview angle

"Why is `Scanner` slow?" Regex tokenising and per-token parsing through strings; a `BufferedReader` + `StringTokenizer` is ~10× faster.

"What does buffering actually save?" System calls: one 8 KB read serves thousands of small reads.

"Why did my program print nothing?" Buffered output never flushed — call `flush()` / `close()` on the `PrintWriter`.

"How do you read until end of input?" `readLine()` returns `null` at EOF; `Scanner.hasNext()` returns `false`.

"Fastest way to read ten million integers in Java?" A byte-level reader over `BufferedInputStream`, parsing digits by hand.

Key takeaways

Unbuffered I/O is a system call per byte; buffer everything, always.

Template: `BufferedReader` + `StringTokenizer` with a refill loop, `PrintWriter` out, `flush()` at the end.

Byte-level parsing for the extreme cases; `readAllBytes` + `split` for the simple ones.

Format with `Locale.ROOT`; avoid `printf` in hot loops; `StringBuilder` then one print is fine.

`readLine()` → `null` at EOF; `read()` → −1.

EXERCISE 1

The fast reader

Read an integer `n` followed by `n` **long** integers that may be spread over any number of lines, using a `BufferedReader` and a `StringTokenizer` with a refill loop — **no** `Scanner`. Write the answer through a `PrintWriter` and flush it: `sum=<sum> min=<smallest> max=<largest>`.

Example (three lines): `5`, `3 9`, `-2 7 1` →

```
sum=18 min=-2 max=9
```

INPUT 1

```
5
3 9
-2 7 1
```

OUTPUT 1

```
sum=18 min=-2 max=9
```

INPUT 2

```
3 10000000000 20000000000 30000000000
```

OUTPUT 2

```
sum=60000000000 min=10000000000
max=30000000000
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static StringTokenizer st;

    static String next() throws IOException {
        while (st == null || !st.hasMoreTokens()) st = new StringTokenizer(in.readLi
        return st.nextToken();
    }

    public static void main(String[] args) throws IOException {
        PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(
        int n = Integer.parseInt(next());
        long sum = 0, min = Long.MAX_VALUE, max = Long.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            long x = Long.parseLong(next());
            sum += x;
            min = Math.min(min, x);
            max = Math.max(max, x);
        }
        out.println("sum=" + sum + " min=" + min + " max=" + max);
        out.flush();
    }
}
```

EXERCISE 2

A buffered report

Read an integer `n` and `n` records `name score`. Through a `PrintWriter` wrapped around a buffered writer on `System.out`, print one CSV line per record, `name,score,grade`, where the grade is `A` for 90 and above, `B` for 75–89, `C` for 50–74, else `F`; then `average=<mean score, two decimals>` (use `Locale.ROOT`). Flush at the end.

Example: `3` then `ann 91`, `bob 75`, `cy 40` →

```
ann,91,A
bob,75,B
cy,40,F
average=68.67
```

INPUT 1

```
3
ann 91
bob 75
cy 40
```

OUTPUT 1

```
ann,91,A
bob,75,B
cy,40,F
average=68.67
```

INPUT 2

```
1
zed 50
```

OUTPUT 2

```
zed,50,C
average=50.00
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static char grade(int s) { return s ⩾ 90 ? 'A' : s ⩾ 75 ? 'B' : s ⩾ 50 ? 'C'

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(
        int n = Integer.parseInt(in.readLine().trim());
        long total = 0;
        for (int i = 0; i < n; i++) {
            StringTokenizer st = new StringTokenizer(in.readLine());
            String name = st.nextToken();
            int score = Integer.parseInt(st.nextToken());
            total += score;
            out.println(name + "," + score + "," + grade(score));
        }
        out.println(String.format(Locale.ROOT, "average=%.2f", n == 0 ? 0.0 : (doubl
        out.flush();
    }
}
```

QUIZ

1. Buffering speeds I/O up mainly by…

A. Compressing data

B. Reducing system calls: one 8 KB read serves thousands of small reads

C. Using more threads

D. Skipping the charset

2. `Scanner` is slower than `BufferedReader` + `StringTokenizer` because…

A. It is unbuffered

B. It tokenises with regular expressions and parses through strings per token

C. It reads bytes

D. It is synchronized

3. A solution using `PrintWriter` prints nothing on the judge. The most likely cause is…

A. Wrong charset

B. A missing `flush()` / `close()` — the output sits in the buffer

C. The `PrintWriter` is deprecated

D. Too much output

4. `BufferedReader.readLine()` at end of input returns…

A. `""`

B. `null`

C. Throws `EOFException`

D. `-1`

5. Formatting a decimal with `String.format("%.2f", x)` and no locale on a German-locale judge prints…

A. `3.14`

B. `3,14` — pass `Locale.ROOT` or `Locale.US`

C. `3.140`

D. An exception

ANSWER KEY · BUFFERING AND FAST I/O — THE TEMPLATE THAT PASSES TIME LIMITS

1. B A kernel round trip costs about a microsecond; unbuffered byte reads pay it every time.

2. B Roughly an order of magnitude on a million integers.

3. B `new PrintWriter(System.out)` has no autoflush by default.

4. B `read()` returns −1; `readLine()` returns `null`; `Scanner.hasNext()` returns `false`.

5. B Number formatting follows the default locale unless you say otherwise.

LESSON 3 · 14 MIN

Files and paths — java.nio.file

The `java.io.File` class you may have met is twenty-five years old and mostly returns `false` when things go wrong. Since Java 7 the file system API is `java.nio.file`: `Path` is an immutable, purely syntactic description of a location; `Files` is a utility class of static operations on paths that throw real exceptions with real messages; and the two together read, write, copy, move, list and walk with one-liners that were fifteen lines before. This lesson covers the API you will use ninety-five percent of the time, the path arithmetic behind `..` and relative paths, and the handful of operations that are still surprising.

`Path`: a location, not a file

```
Path p = Path.of("/srv/app", "logs", "today.log");     // Java 11; Paths.get(
p.getFileName();          // today.log
p.getParent();            // /srv/app/logs
p.getRoot();              // /  (null for a relative path)
p.getNameCount();         // 4
p.isAbsolute();           // true
p.toString();             // /srv/app/logs/today.log — separators are the pla
```

Nothing touches the disk. A `Path` can name a file that does not exist; `Files.exists(p)` is the question, not the constructor. Separators are platform-specific (`/` on Linux and macOS, `\` on Windows) but `Path.of("a/b")` is accepted everywhere.

Path arithmetic

```
Path base = Path.of("/srv/app");
base.resolve("conf/app.yml");                  // /srv/app/conf/app.yml
base.resolve("/etc/passwd");                   // /etc/passwd
base.resolve("../logs/./x.log").normalize();   // /srv/logs/x.log
base.relativize(Path.of("/srv/logs/x.log"));   // ../logs/x.log
base.resolveSibling("backup");                 // /srv/backup
Path.of("a/b/c").subpath(1, 3);                // b/c
Path.of("x.tar.gz").getFileName().toString(); // the extension is yours to sp
```

`normalize()` is syntactic: it does not check that the directories exist or follow symbolic links; `toRealPath()` does both and throws if the file is missing. `resolve` with an absolute path replacing the base is a security classic — never resolve a user-supplied string against a trusted directory without checking that the result still `startsWith` it after normalising.

`Files`: reading and writing

```
String text = Files.readString(p);                                  // Java 11, UTF
List<String> lines = Files.readAllLines(p);                         // whole file i
byte[] bytes = Files.readAllBytes(p);
Files.writeString(p, "hello\n");                                    // create or tr
Files.writeString(p, "more\n", StandardOpenOption.APPEND);
Files.write(p, lines);                                              // one line per
try (Stream<String> s = Files.lines(p)) { s.filter(...).count(); }   // lazy;
try (BufferedReader r = Files.newBufferedReader(p)) { … }
try (BufferedWriter w = Files.newBufferedWriter(p, StandardOpenOption.CREATE,
```

`readString` / `readAllLines` are right for files that fit comfortably in memory (configuration, inputs measured in megabytes). `Files.lines` streams lazily and is the choice for anything large — but it holds a file handle, so it goes in a try-with-resources. `Files.newBufferedReader` builds the decorator stack from lesson 1 with UTF-8 already in it.

`Files`: the file system

```
Files.exists(p); Files.isDirectory(p); Files.isRegularFile(p); Files.isReadabl
Files.size(p);                                          // bytes
Files.getLastModifiedTime(p);                           // FileTime → toInstan
Files.createDirectories(dir);                           // mkdir -p: no error
Files.createFile(p);                                    // FileAlreadyExistsEx
Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
Files.move(src, dst, StandardCopyOption.ATOMIC_MOVE);   // rename; atomic on t
Files.delete(p);                                        // NoSuchFileExceptio
Files.deleteIfExists(p);                                // returns boolean, ne
Path tmp = Files.createTempFile("report-", ".csv");     // in the system temp
```

The exceptions are the improvement: `NoSuchFileException`, `AccessDeniedException`, `DirectoryNotEmptyException`, `FileAlreadyExistsException` — each an `IOException` subclass naming the path. `java.io.File.delete()` just returned `false`.

Listing and walking

```
try (Stream<Path> children = Files.list(dir)) { … }                    // one
try (Stream<Path> all = Files.walk(dir)) {                             // rec
    long javaFiles = all.filter(x → x.toString().endsWith(".java")).count();
}
try (Stream<Path> found = Files.find(dir, 10, (path, attrs) → attrs.size() >
Files.walkFileTree(dir, new SimpleFileVisitor<>() { … });              // the
```

All of these are lazy streams over open directory handles — **close them** with try-with-resources, or a long-running process runs out of file descriptors. Order is not specified; sort if it matters. For "delete a directory tree", walk it, sort in reverse (children before parents), and `deleteIfExists` each.

Attributes, permissions and links

`Files.readAttributes(p, BasicFileAttributes.class)` returns size, times and type in one system call — cheaper than four separate queries. `PosixFileAttributes` and `Files.setPosixFilePermissions` handle Unix modes; `Files.isSymbolicLink`, `Files.readSymbolicLink` and the `LinkOption.NOFOLLOW_LINKS` flag control whether operations follow links. `Files.probeContentType(p)` guesses a MIME type from the name.

The `WatchService`, briefly

`FileSystems.getDefault().newWatchService()` plus `dir.register(watcher, ENTRY_CREATE, ENTRY_MODIFY)` delivers events when a directory changes — the basis of hot-reload and log tailers. Events can coalesce and overflow; production code re-scans on `OVERFLOW`.

Legacy `File`, and when you still meet it

`java.io.File` is not deprecated and many APIs still take one. Convert both ways: `path.toFile()` and `file.toPath()`. Prefer `Path` in new code; the `Files` methods are clearer, faster (fewer system calls) and honest about errors.

Interview angle

"`Path` versus `File`?" `Path` is an immutable syntactic location with real exceptions from `Files`; `File` is the legacy class returning booleans.

"What does `resolve` do with an absolute argument?" Returns the argument — the base is discarded; check `startsWith` after `normalize` for user input.

"`normalize` versus `toRealPath`?" Syntactic `.` / `..` removal versus resolving links against the real file system (and failing if absent).

"Why must `Files.lines` / `Files.walk` be closed?" They hold a file or directory handle open while the stream is alive.

"How do you read a whole file as a string?" `Files.readString(path)` (Java 11), UTF-8 by default.

Key takeaways

`Path` is arithmetic — `resolve`, `normalize`, `relativize`, `getFileName` — and touches no disk.

`Files` does the work: `readString` / `writeString`, `readAllLines`, `lines` (close it), `newBufferedReader/Writer`.

`createDirectories`, `copy` / `move` with options, `delete` / `deleteIfExists`, temp files; exceptions name the path.

`list` / `walk` / `find` are lazy streams over open handles — try-with-resources every time.

Never trust `resolve` on user input without a `startsWith` check on the normalised result.

EXERCISE 1

Path arithmetic

Read an absolute base path and an integer `n`, then `n` paths (relative or absolute). For each print one line: `resolved=<base.resolve(p).normalize()> name=<file name> ext=<text after the last dot of the name, or → rel=<base.relativize(resolved)>`. Nothing here touches the disk — the paths need not exist.

Example: base `/srv/app`, `n = 2`, paths `../logs/./x.log` and `conf/app.yml` →

```
resolved=/srv/logs/x.log name=x.log ext=log rel=../logs/x.log
resolved=/srv/app/conf/app.yml name=app.yml ext=yml rel=conf/app.yml
```

INPUT 1

```
/srv/app 2
../logs/./x.log
conf/app.yml
```

OUTPUT 1

```
resolved=/srv/logs/x.log name=x.log
ext=log rel=../logs/x.log
resolved=/srv/app/conf/app.yml
name=app.yml ext=yml rel=conf/app.yml
```

INPUT 2

```
/home/dev 2
README
/etc/hosts
```

OUTPUT 2

```
resolved=/home/dev/README name=README
ext=- rel=README
resolved=/etc/hosts name=hosts ext=-
rel=../../etc/hosts
```

REFERENCE SOLUTION

```java
import java.nio.file.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Path base = Path.of(in.next());
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            Path p = Path.of(in.next());
            Path resolved = base.resolve(p).normalize();
            String name = resolved.getFileName().toString();
            int dot = name.lastIndexOf('.');
            String ext = dot > 0 && dot < name.length() - 1 ? name.substring(dot + 1
            System.out.println("resolved=" + resolved + " name=" + name + " ext=" +
        }
    }
}
```

EXERCISE 2

Temp file round trip

Read an integer `n` and `n` words. Create a temporary file with `Files.createTempFile`, write the words one per line with `Files.write`, and then — reading the file back, not your list — print `lines=<Files.readAllLines(...).size()>`, `size=<Files.size in bytes>` (every word plus one newline each), `longest=<the first longest line, via Files.lines>`. Finally delete the file and print `deleted=<Files.deleteIfExists result> exists=<Files.exists>`.

Example: `3` then `kiwi fig banana` →

```
lines=3
size=16
longest=banana
deleted=true exists=false
```

INPUT 1

```
3
kiwi fig banana
```

OUTPUT 1

```
lines=3
size=16
longest=banana
deleted=true exists=false
```

INPUT 2

```
1
x
```

OUTPUT 2

```
lines=1
size=2
longest=x
deleted=true exists=false
```

REFERENCE SOLUTION

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) throws IOException {
        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());
        Path tmp = Files.createTempFile("study-", ".txt");
        Files.write(tmp, words);
        System.out.println("lines=" + Files.readAllLines(tmp).size());
        System.out.println("size=" + Files.size(tmp));
        try (Stream<String> lines = Files.lines(tmp)) {
            System.out.println("longest=" + lines.max(Comparator.comparingInt(String
        }
        boolean deleted = Files.deleteIfExists(tmp);
        System.out.println("deleted=" + deleted + " exists=" + Files.exists(tmp));
    }
}
```

QUIZ

1. `Path.of("/srv/app").resolve("/etc/passwd")` returns…

A. `/srv/app/etc/passwd`

B. `/etc/passwd` — an absolute argument replaces the base

C. An exception

D. `/srv/etc/passwd`

2. `normalize()` differs from `toRealPath()` in that it…

A. Touches the disk

B. Is purely syntactic — removes `.` and `..` without checking existence or links

C. Follows symlinks

D. Makes the path absolute

3. Which `Files` methods return streams that must be closed?

A. `readAllLines`, `readString`

B. `lines`, `list`, `walk`, `find`

C. `exists`, `size`

D. `writeString`

4. `Files.createDirectories(p)` when `p` already exists…

A. Throws `FileAlreadyExistsException`

B. Succeeds silently, like `mkdir -p`

C. Deletes and recreates

D. Returns `false`

5. `java.io.File.delete()` on a missing file…

A. Throws `NoSuchFileException`

B. Returns `false` with no explanation — one reason to prefer `Files.delete`

C. Creates the file

D. Blocks

ANSWER KEY · FILES AND PATHS — JAVA.NIO.FILE

1. B Always `normalize()` and check `startsWith(base)` when the argument comes from a user.

2. B `toRealPath` resolves against the real file system and throws if the file is missing.

3. B They hold a file or directory handle open while the stream lives — try-with-resources.

4. B `createDirectory` (singular) throws; `createFile` throws when the file exists.

5. B `Files.delete` throws a named exception; `Files.deleteIfExists` returns a boolean deliberately.

LESSON 4 · 14 MIN

Serialization, CSV and the formats you actually use

Sooner or later an object has to leave the JVM — into a file, across a socket, into a cache — and come back. Java shipped its own answer in 1997: mark a class `Serializable` and `ObjectOutputStream` writes the whole object graph. It still works, it is still on interview lists, and the platform's own architects now advise against it. This lesson explains how it works and what its knobs (`serialVersionUID`, `transient`, `readObject`) do, why it is a security problem, and then the formats real systems use — JSON, CSV, `Properties` — including the CSV parser that everyone gets wrong on the first try.

Java serialization: the mechanism

```
class Session implements Serializable {
    private static final long serialVersionUID = 1L;
    private final String user;
    private transient String token;          // not written; null after deseri
    …
}

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("s.b
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("s.bin")
```

`Serializable` is a **marker interface** — no methods; it grants permission. `writeObject` walks the object graph reflectively, writing the class name, the `serialVersionUID`, and every non-`transient`, non-`static` field, recursively; shared references are written once and restored as shared. Every class in the graph must be `Serializable`, or `NotSerializableException` names the one that is not. `readObject` recreates the objects **without calling any constructor** of the serialisable classes (the first non-serialisable superclass's no-arg constructor does run) — invariants a constructor enforced are simply not re-checked.

> `serialVersionUID` is the version stamp. If you omit it the compiler computes one from the class's shape, so adding a method changes it and old data fails with `InvalidClassException`. Declare it explicitly; bump it when the format really changes.
>
> `transient` excludes a field: caches, derived values, secrets, anything not serialisable (`Thread`, a socket). It comes back as its default (`null`, `0`, `false`), so restore it in

`readObject`.

`private void writeObject(ObjectOutputStream)` / `readObject(ObjectInputStream)` hooks customise the format and are where you validate and rebuild transient state — call `defaultWriteObject()` / `defaultReadObject()` first.

`readResolve()` replaces the deserialised object with another — the way a serialisable singleton stays single.

Records serialise by their components and are reconstructed through the canonical constructor, so their invariants *are* re-checked — one of several reasons records are safer here.

`Externalizable` is the fully manual variant: you write and read every byte yourself and a public no-arg constructor is called.

Why "do not use it" is the current advice

Deserialising **untrusted bytes** lets the sender choose which classes are instantiated and how their fields are set. Chains of ordinary library classes ("gadget chains") have been found that execute arbitrary code during `readObject`; several major breaches rode on this. Mitigations exist — `ObjectInputFilter` (Java 9) whitelists classes; never deserialise from the network without one — but the structural problem remains: the format couples wire data to class internals, breaks on refactors, is Java-only, and is slow and verbose. Brian Goetz's summary is the one to quote: *serialization was a mistake; use a data format you control.* For interviews: know the mechanism, know the knobs, and know that the recommendation is JSON/Protobuf with explicit mapping.

JSON, the default today

Use a library — Jackson (the de facto standard), Gson, or JSON-B — never a hand-written parser. The mapping is by property name, so fields can be renamed with annotations, unknown fields ignored, and versions evolve gracefully. Records map cleanly since Jackson 2.12. Two habits: make DTOs (the classes you serialise) separate from domain classes, and configure the mapper once (`FAIL_ON_UNKNOWN_PROPERTIES` off, dates as ISO-8601 strings). Binary alternatives when size and speed matter: Protocol Buffers, Avro, CBOR — all schema-driven and cross-language.

CSV: deceptively simple

`line.split(",")` is a CSV parser for exactly as long as no field contains a comma. The real rules (RFC 4180): fields containing commas, quotes or line breaks are wrapped in double quotes; a quote inside a quoted field is doubled (`"He said ""hi"""`). A correct field splitter is a small state machine:

```
static List<String> parseCsvLine(String line) {
    List<String> fields = new ArrayList<>();
    StringBuilder cur = new StringBuilder();
    boolean quoted = false;
    for (int i = 0; i < line.length(); i++) {
        char c = line.charAt(i);
        if (quoted) {
            if (c == '"') {
                if (i + 1 < line.length() && line.charAt(i + 1) == '"') { cur
                else quoted = false;
            } else cur.append(c);
        } else if (c == '"') quoted = true;
        else if (c == ',') { fields.add(cur.toString()); cur.setLength(0); }
        else cur.append(c);
    }
    fields.add(cur.toString());
    return fields;
}
```

Fields spanning lines need the reader to keep consuming until the quotes balance. Writing is the mirror: quote a field if it contains a comma, quote, or newline, doubling internal quotes. For anything serious use Apache Commons CSV or OpenCSV — but be able to write the loop above, because "parse this CSV" is a common interview exercise and `split(",")` is the wrong answer they are waiting for.

`Properties` and other text formats

`java.util.Properties` reads `key=value` files (`props.load(reader)`, `props.getProperty("port", "8080")`) — configuration, resource bundles, `System.getProperties()`. It is a `Hashtable<Object,Object>`, ancient and thread-safe by accident; fine for what it does. YAML and TOML need libraries (SnakeYAML, tomlj). XML has

JAXB (removed from the JDK in 11; add the dependency) and the built-in DOM/SAX/StAX parsers — StAX for large documents.

Choosing a format

Need	Use
Interop, APIs, config	JSON (Jackson)
Compact, fast, schema-evolved	Protobuf / Avro
Spreadsheets, bulk exports	CSV (library)
Java-only, same version both ends, trusted source, short-lived	Java serialization, with an `ObjectInputFilter` — and still think twice
Key/value config	`Properties`

Interview angle

"How does Java serialization work?" `Serializable` marker + `ObjectOutputStream` writes the graph reflectively; `readObject` rebuilds it without constructors.

"What is `serialVersionUID`*?"* The class's format version; mismatch → `InvalidClassException`; declare it explicitly.

"What does `transient` *do?"* Skips the field; it is null/zero after deserialisation.

"Why is deserialising untrusted data dangerous?" The sender chooses classes and field values; gadget chains give remote code execution. Filter, or use JSON.

"How do you parse CSV with quoted commas?" A state machine tracking whether you are inside quotes; doubled quotes are escapes; not `split`.

Key takeaways

Marker interface, reflective graph write, no constructors on read; `serialVersionUID`, `transient`, `readObject` / `writeObject`, `readResolve` are the knobs.

Untrusted deserialisation is a remote-code-execution risk; filter with `ObjectInputFilter` or, better, do not.

JSON with Jackson for almost everything; Protobuf/Avro for compact schema-driven data.

CSV needs a quote-aware state machine; `split(",")` is wrong.

Records serialise through their constructor and keep their invariants.

EXERCISE 1

Serialize in memory

Read `user token x y`. Define `class Session implements Serializable` with a `String user` and a `transient String token`, and `record Point(int x, int y) implements Serializable`. Write both objects to an `ObjectOutputStream` over a `ByteArrayOutputStream`, read them back through an `ObjectInputStream`, and print `session=<user>/<token>` (the token comes back as `null`), `point=<the record's toString>`, `equalPoint=<restored.equals(original)>` and `samePoint=<restored == original>`.

Example: `ann secret 3 4` →

```
session=ann/null
point=Point[x=3, y=4]
equalPoint=true
samePoint=false
```

INPUT 1

```
ann secret 3 4
```

OUTPUT 1

```
session=ann/null
point=Point[x=3, y=4]
equalPoint=true
samePoint=false
```

INPUT 2

```
bob t -1 0
```

OUTPUT 2

```
session=bob/null
point=Point[x=-1, y=0]
equalPoint=true
samePoint=false
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

class Session implements Serializable {
    private static final long serialVersionUID = 1L;
    private final String user;
    private transient String token;
    Session(String user, String token) { this.user = user; this.token = token; }
    String user() { return user; }
    String token() { return token; }
}

record Point(int x, int y) implements Serializable { }

public class Main {
    public static void main(String[] args) throws IOException, ClassNotFoundExceptio
        Scanner in = new Scanner(System.in);
        String user = in.next(), token = in.next();
        int x = in.nextInt(), y = in.nextInt();
        Session session = new Session(user, token);
        Point point = new Point(x, y);

        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        try (ObjectOutputStream out = new ObjectOutputStream(buffer)) {
            out.writeObject(session);
            out.writeObject(point);
        }
        try (ObjectInputStream oin = new ObjectInputStream(new ByteArrayInputStream(
            Session s2 = (Session) oin.readObject();
            Point p2 = (Point) oin.readObject();
            System.out.println("session=" + s2.user() + "/" + s2.token());
            System.out.println("point=" + p2);
            System.out.println("equalPoint=" + p2.equals(point));
            System.out.println("samePoint=" + (p2 == point));
        }
    }
}
```

EXERCISE 2

CSV fields, with quotes

Read CSV lines until end of input and split each into fields by the RFC 4180 rules: fields are separated by commas; a field wrapped in double quotes may contain commas; a doubled quote inside a quoted field is one literal quote. Print each line as `<field count>: [<fields joined by |>]`. Do **not** use `split`.

Example input

```
a,b,c
"x, y",z
"He said ""hi""",1
a,,b
```

→

```
3: [a|b|c]
2: [x, y|z]
2: [He said "hi"|1]
3: [a||b]
```

INPUT 1

```
a,b,c
"x, y",z
"He said ""hi""",1
a,,b
```

OUTPUT 1

```
3: [a|b|c]
2: [x, y|z]
2: [He said "hi"|1]
3: [a||b]
```

INPUT 2

```
solo
""
```

OUTPUT 2

```
1: [solo]
1: []
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

public class Main {
    static List<String> parse(String line) {
        List<String> fields = new ArrayList<>();
        StringBuilder cur = new StringBuilder();
        boolean quoted = false;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (quoted) {
                if (c == '"') {
                    if (i + 1 < line.length() && line.charAt(i + 1) == '"') { cur.ap
                    else quoted = false;
                } else cur.append(c);
            } else if (c == '"') quoted = true;
            else if (c == ',') { fields.add(cur.toString()); cur.setLength(0); }
            else cur.append(c);
        }
        fields.add(cur.toString());
        return fields;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
            if (line.isEmpty()) continue;
            List<String> f = parse(line);
            System.out.println(f.size() + ": [" + String.join("|", f) + "]");
        }
    }
}
```

QUIZ

1. `Serializable` is…

A. An abstract class

B. A marker interface with no methods

C. An annotation

D. A `final` class

2. A `transient` field after deserialisation holds…

A. Its serialised value

B. Its default value (`null` , `0` , `false`)

C. A copy of the original

D. An exception

3. Omitting `serialVersionUID` …

A. Is recommended

B. Lets the compiler derive one from the class shape, so unrelated edits break old data with `InvalidClassException`

C. Disables serialization

D. Makes the class immutable

4. Deserialising bytes from an untrusted client is dangerous because…

A. It is slow

B. The sender chooses the classes and field values; gadget chains can execute code — filter or use JSON

C. It uses reflection

D. It needs a constructor

5. `"a,\"b,c\",d".split(",")` produces…

A. 3 fields

B. 4 pieces — `split` knows nothing about quotes

C. 2 fields

D. An exception

6. Records are safer to deserialise than classes because…

A. They cannot be serialised

B. They are reconstructed through the canonical constructor, so invariants are re-checked

C. They have no fields

D. They are `transient`

ANSWER KEY · SERIALIZATION, CSV AND THE FORMATS YOU ACTUALLY USE

1. B It grants permission; `ObjectOutputStream` does the work reflectively.

2. B Restore it in a custom `readObject` if it is needed.

3. B Declare it explicitly and bump it on real format changes.

4. B `ObjectInputFilter` whitelists classes; the better answer is a data format you control.

5. B A quote-aware state machine (or a CSV library) is the only correct parser.

6. B Ordinary serialisable classes are rebuilt without any constructor running.

LESSON 5 · 15 MIN

java.time — dates, times, zones and durations

Dates are where careful programmers become humble. Months are zero-based in the old API, a day is not always 24 hours, "next month" from January 31 has no obvious answer, and the same instant is Tuesday in Tokyo and Monday in Toronto. `java.time` (Java 8, from Joda-Time's author) gets this right by refusing to blur the distinctions: an **instant** on the timeline, a **local** date or time with no zone, a **zoned** date-time, and two kinds of amount — `Duration` (seconds) and `Period` (years/months/days). Every type is immutable and thread-safe. This lesson is the map of those types, the arithmetic and formatting you do daily, and the legacy classes you must recognise and avoid.

The types, and which question each answers

Type	Answers	Example
`Instant`	*when*, on the machine timeline (UTC, nanoseconds since 1970)	`2024-03-01T04:45:00Z` — timestamps, logs, `System.currentTimeMillis()`
`LocalDate`	a calendar date, no time, no zone	`2024-03-01` — birthdays, due dates
`LocalTime`	a wall-clock time, no date, no zone	`09:30` — opening hours
`LocalDateTime`	date + time, **no zone** — ambiguous as an instant	`2024-03-01T09:30` — a meeting "at 9:30" before you know where
`ZonedDateTime`	date + time + zone rules	`2024-03-01T09:30+05:30[Asia/Kolkata]` — the meeting, scheduled
`OffsetDateTime`	date + time + fixed offset (no DST rules)	`2024-03-01T09:30+05:30` — what databases and APIs exchange
`ZoneId` / `ZoneOffset`	a region's rules / a fixed offset	`Asia/Kolkata`, `+05:30`
`Duration`	an amount of time in seconds/nanos	`PT2H30M` — timeouts, elapsed time
`Period`	an amount of calendar time in years/months/days	`P1Y2M3D` — "in 3 months"
`Year`, `YearMonth`, `MonthDay`, `DayOfWeek`, `Month`	the partials	`2024-03`, `--12-25`, `FRIDAY`

The discipline: use the *narrowest* type that answers the question. A birthday is a `LocalDate`; storing it as a `Date` at midnight in the server's zone is how people are born a day early in another country.

Creating and parsing

```
LocalDate d = LocalDate.of(2024, 3, 1);                  // months are 1-based,
LocalDate parsed = LocalDate.parse("2024-03-01");         // ISO-8601 by defau
LocalTime t = LocalTime.parse("09:30");
LocalDateTime ldt = LocalDateTime.of(d, t);
ZonedDateTime z = ldt.atZone(ZoneId.of("Asia/Kolkata"));
Instant now = Instant.now();                               // or Instant.ofEpoch
LocalDate today = LocalDate.now(ZoneId.of("Europe/Berlin"));   // "today" depe
```

Every `now()` takes an optional `Clock`; pass a fixed one in tests (`Clock.fixed(instant, zone)`) so code that depends on "now" becomes testable. Parsing a bad string throws `DateTimeParseException` with the index of the problem.

Arithmetic

```
d.plusDays(30); d.minusWeeks(2); d.plusMonths(1);       // 2024-01-31 plusMont
d.withDayOfMonth(1); d.with(TemporalAdjusters.lastDayOfMonth()); d.with(Tempor
d.getDayOfWeek(); d.getDayOfYear(); d.isLeapYear(); d.lengthOfMonth();
d.isBefore(other); d.isAfter(other); d.compareTo(other);
ChronoUnit.DAYS.between(d1, d2);                          // signed; d2 - d1
Period.between(d1, d2);                                   // P0Y1M3D style: yea
Duration.between(t1, t2);                                 // for times/instants
Duration.ofMinutes(90).toHours();                         // 1 — truncates; tol
```

Month arithmetic **clamps** rather than overflows: March 31 minus a month is February 29 (or 28), and `plusMonths(1).plusMonths(1)` can differ from `plusMonths(2)` for that reason. `Period` and `Duration` are different things: adding `Period.ofDays(1)` to a `ZonedDateTime` across a DST change moves the wall clock by one day (23 or 25 hours); adding `Duration.ofHours(24)` moves exactly 24 hours and lands at a different wall-clock time. Choose the one that matches the sentence a human would say.

Zones and conversions

```
ZonedDateTime kolkata = ZonedDateTime.of(LocalDateTime.of(2024, 3, 1, 9, 30),
ZonedDateTime newYork = kolkata.withZoneSameInstant(ZoneId.of("America/New_Yor
ZonedDateTime sameWall = kolkata.withZoneSameLocal(ZoneId.of("America/New_York
Instant instant = kolkata.toInstant();
LocalDate localDate = instant.atZone(ZoneId.of("UTC")).toLocalDate();
```

`withZoneSameInstant` is the conversion you almost always mean. Zone rules (including DST history and future changes) come from the JDK's copy of the IANA tz database, updated with the JDK — a fixed date in the past converts deterministically; a date years ahead may shift if a government changes its rules. Use region ids (`Asia/Kolkata`), not abbreviations (`IST` is India, Ireland and Israel).

Formatting

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("EEE, dd MMM yyyy HH:mm", Lo
f.format(kolkata);                                    // "Fri, 01 Mar 2024 (
DateTimeFormatter.ISO_LOCAL_DATE.format(d);            // 2024-03-01
LocalDate.parse("01/03/2024", DateTimeFormatter.ofPattern("dd/MM/yyyy"));
```

Pattern letters: `y` year, `M` month (`MM` 03, `MMM` Mar, `MMMM` March), `d` day, `E` weekday, `H` hour 0–23, `h` hour 1–12 with `a` AM/PM, `m` minute, `s` second, `S` fraction, `z` / `Z` / `X` zone and offset forms. Always pass a `Locale` when the output contains names — the judge, the server and the user may not agree on the default. `DateTimeFormatter` is immutable and thread-safe, unlike `SimpleDateFormat`, which was neither and caused a generation of production bugs when shared as a static field.

The legacy classes: recognise, convert, avoid

`java.util.Date` is an instant with a misleading name and mutable state; `Calendar` has zero-based months and mutable everything; `SimpleDateFormat` is not thread-safe; `java.sql.Date` extends `Date` and throws on `getHours`. Convert at the boundary — `date.toInstant()`, `Date.from(instant)`, `Timestamp.toLocalDateTime()` — and keep `java.time` inside. JDBC 4.2 maps `LocalDate` / `LocalDateTime` / `OffsetDateTime` directly, so even the database boundary no longer needs `Date`.

Interview angle

> "*`LocalDateTime` versus `ZonedDateTime` versus `Instant`?*" Wall-clock without zone (ambiguous as a moment); wall-clock with zone rules; a point on the UTC timeline.
>
> "*`Period` versus `Duration`?*" Calendar amounts (years, months, days) versus exact seconds; they differ across DST.

"What does `Jan 31 plusMonths(1)` give?" `Feb 28/29` — clamped to the last valid day.

"Why replace `SimpleDateFormat`?" Not thread-safe and mutable; `DateTimeFormatter` is immutable and shareable.

"How do you make time-dependent code testable?" Inject a `Clock`; call `now(clock)`.

Key takeaways

Instant for *when*, Local* for calendar/wall-clock without zone, Zoned/Offset when the zone is known; narrowest type that fits.

`plus` / `minus` / `with` / `TemporalAdjusters`; `ChronoUnit.between` for counts; month arithmetic clamps.

`Period` (calendar) ≠ `Duration` (exact); `withZoneSameInstant` converts; use IANA region ids.

`DateTimeFormatter` with an explicit `Locale`; ISO formats are the defaults for `parse` / `toString`.

Legacy `Date` / `Calendar` / `SimpleDateFormat`: convert at the boundary and never let them in.

EXERCISE 1

Date arithmetic

Read two ISO dates `a` and `b` (`yyyy-MM-dd`). Print:

`days=<ChronoUnit.DAYS.between(a, b)>`

`period=<Period.between(a, b) as "Yy Mm Dd">`

`dow=<a's day of week>`

`leap=<a.isLeapYear()>`

`nextMonth=<a.plusMonths(1)>` — note the clamping

`lastDay=<a with the last day of its month>`

Example: `2024-01-31 2024-03-01` →

```
days=30
period=0y 1m 1d
dow=WEDNESDAY
leap=true
nextMonth=2024-02-29
lastDay=2024-01-31
```

INPUT 1

```
2024-01-31 2024-03-01
```

OUTPUT 1

```
days=30
period=0y 1m 1d
dow=WEDNESDAY
leap=true
nextMonth=2024-02-29
lastDay=2024-01-31
```

INPUT 2

```
2023-12-25 2023-12-20
```

OUTPUT 2

```
days=-5
period=0y 0m -5d
dow=MONDAY
leap=false
nextMonth=2024-01-25
lastDay=2023-12-31
```

REFERENCE SOLUTION

```java
import java.time.*;
import java.time.temporal.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        LocalDate a = LocalDate.parse(in.next());
        LocalDate b = LocalDate.parse(in.next());
        Period p = Period.between(a, b);
        System.out.println("days=" + ChronoUnit.DAYS.between(a, b));
        System.out.println("period=" + p.getYears() + "y " + p.getMonths() + "m " +
        System.out.println("dow=" + a.getDayOfWeek());
        System.out.println("leap=" + a.isLeapYear());
        System.out.println("nextMonth=" + a.plusMonths(1));
        System.out.println("lastDay=" + a.with(TemporalAdjusters.lastDayOfMonth()));
    }
}
```

EXERCISE 2

Zones and durations

Read a local date-time (`yyyy-MM-ddTHH:mm`), a source zone id and a target zone id, then two times `HH:mm`. Print:

- `source=<ZonedDateTime of the date-time in the source zone>`
- `target=<the same instant in the target zone>` (`withZoneSameInstant`)
- `instant=<the Instant>`
- `formatted=<the target date-time as "EEE, dd MMM yyyy HH:mm" in Locale.ENGLISH>`
- `duration=<Duration.between(t1, t2)>` — it may be negative

Example: `2024-03-01T09:30 Asia/Kolkata America/New_York` then `08:15 10:45` →

```
source=2024-03-01T09:30+05:30[Asia/Kolkata]
target=2024-02-29T23:00-05:00[America/New_York]
instant=2024-03-01T04:00:00Z
formatted=Thu, 29 Feb 2024 23:00
duration=PT2H30M
```

INPUT 1

```
2024-03-01T09:30 Asia/Kolkata
America/New_York
08:15 10:45
```

OUTPUT 1

```
source=2024-03-
01T09:30+05:30[Asia/Kolkata]
target=2024-02-29T23:00-
05:00[America/New_York]
instant=2024-03-01T04:00:00Z
formatted=Thu, 29 Feb 2024 23:00
duration=PT2H30M
```

INPUT 2

```
2024-07-04T12:00 Europe/London UTC
23:00 01:00
```

OUTPUT 2

```
source=2024-07-
04T12:00+01:00[Europe/London]
target=2024-07-04T11:00Z[UTC]
instant=2024-07-04T11:00:00Z
formatted=Thu, 04 Jul 2024 11:00
duration=PT-22H
```

REFERENCE SOLUTION

```
import java.time.*;
import java.time.format.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        LocalDateTime ldt = LocalDateTime.parse(in.next());
        ZoneId from = ZoneId.of(in.next()), to = ZoneId.of(in.next());
        LocalTime t1 = LocalTime.parse(in.next()), t2 = LocalTime.parse(in.next());
        ZonedDateTime source = ldt.atZone(from);
        ZonedDateTime target = source.withZoneSameInstant(to);
        System.out.println("source=" + source);
        System.out.println("target=" + target);
        System.out.println("instant=" + source.toInstant());
        System.out.println("formatted=" + DateTimeFormatter.ofPattern("EEE, dd MMM y
        System.out.println("duration=" + Duration.between(t1, t2));
    }
}
```

QUIZ

1. A person's birthday should be stored as…

A. `Instant`

B. `LocalDate` — a calendar date with no time or zone

C. `ZonedDateTime`

D. `Duration`

2. `LocalDate.of(2024, 1, 31).plusMonths(1)` is…

A. `2024-03-02`

B. `2024-02-29` — clamped to the last valid day

C. An exception

D. `2024-02-31`

3. `Period` versus `Duration`:

A. Synonyms

B. Calendar amounts (years/months/days) versus exact seconds — they differ across DST

C. `Duration` is for dates

D. `Period` is in milliseconds

4. To convert a `ZonedDateTime` to the same moment in another zone use…

A. `withZoneSameLocal`

B. `withZoneSameInstant`

C. `atZone`

D. `toLocalDateTime`

5. `SimpleDateFormat` as a shared `static` field was a classic bug because…

A. It was slow

B. It is mutable and not thread-safe; `DateTimeFormatter` is immutable and shareable

C. It ignored locales

D. It is deprecated

6. Predict: `Duration.between(LocalTime.of(23, 0), LocalTime.of(1, 0))`

A. `PT2H`

B. `PT-22H` — same-day arithmetic, so it goes backwards

C. `PT22H`

D. An exception

ANSWER KEY · JAVA.TIME — DATES, TIMES, ZONES AND DURATIONS

1. B The narrowest type that answers the question; a midnight `Date` shifts by a day across zones.

2. B Month arithmetic clamps rather than overflows.

3. B `Period.ofDays(1)` across a DST change is 23 or 25 hours; `Duration.ofHours(24)` is exactly 24.

4. B `withZoneSameLocal` keeps the wall-clock time and changes the moment.

5. B Concurrent `format` calls corrupted each other's output.

6. B `LocalTime` has no date; crossing midnight needs `LocalDateTime` or a manual `plusDays`.

CHECKPOINT · 28 MIN

Checkpoint — I/O, files and time

This checkpoint covers byte and character streams with charsets and the decorator stack, buffering and the fast-I/O template, `Path` arithmetic and the `Files` API, Java serialization and its knobs, CSV parsing with quotes, and `java.time` — the types, arithmetic, zones and formatting.

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 `read()` returns `int`, and what `length()` versus `getBytes(UTF_8).length` count.
- The order of the decorator stack from a raw `InputStream` to `readLine()`.
- Why `Scanner` is slow, what buffering saves, and why a program with a `PrintWriter` printed nothing.
- What `resolve` does with an absolute argument, and `normalize` versus `toRealPath`.
- Which `Files` methods return lazy streams that must be closed.
- What `serialVersionUID` and `transient` do, and why untrusted deserialisation is dangerous.
- Why `split(",")` is not a CSV parser.
- `LocalDateTime` versus `ZonedDateTime` versus `Instant`; `Period` versus `Duration`; what `Jan 31 plusMonths(1)` gives.

The programs are a log analyser that reads until end of input and measures the span with `java.time`, a hex dump of raw bytes, and a CSV-to-fixed-width report using the quote-aware parser.

EXERCISE 1

Log analyser

Read log lines until end of input with a `BufferedReader`. Each line is `<ISO date-time> <LEVEL> <message…>`, for example `2024-03-01T10:15:00 ERROR disk full`. Skip blank lines. Print:

```
lines=<count>
INFO=<n> WARN=<n> ERROR=<n> OTHER=<n>
first=<first timestamp> last=<last timestamp>
span=<minutes between first and last>m
firstError=<message of the first ERROR line, or →
```

Use `LocalDateTime.parse` and `Duration.between` for the span. Assume at least one line.

INPUT 1

```
2024-03-01T10:15:00 INFO started
2024-03-01T10:20:30 WARN slow query
2024-03-01T11:05:00 ERROR disk full
2024-03-01T11:06:00 DEBUG retry
```

OUTPUT 1

```
lines=4
INFO=1 WARN=1 ERROR=1 OTHER=1
first=2024-03-01T10:15 last=2024-03-
01T11:06
span=51m
firstError=disk full
```

INPUT 2

```
2024-01-01T00:00:00 INFO boot
```

OUTPUT 2

```
lines=1
INFO=1 WARN=0 ERROR=0 OTHER=0
first=2024-01-01T00:00 last=2024-01-
01T00:00
span=0m
firstError=-
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.time.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int lines = 0, info = 0, warn = 0, error = 0, other = 0;
        LocalDateTime first = null, last = null;
        String firstError = null;
        while ((line = in.readLine()) != null) {
            if (line.isBlank()) continue;
            String[] parts = line.trim().split("\\s+", 3);
            LocalDateTime ts = LocalDateTime.parse(parts[0]);
            String level = parts.length > 1 ? parts[1] : "";
            String message = parts.length > 2 ? parts[2] : "";
            lines++;
            if (first == null) first = ts;
            last = ts;
            switch (level) {
                case "INFO" -> info++;
                case "WARN" -> warn++;
                case "ERROR" -> { error++; if (firstError == null) firstError = mess
                default -> other++;
            }
        }
        System.out.println("lines=" + lines);
        System.out.println("INFO=" + info + " WARN=" + warn + " ERROR=" + error + "
        System.out.println("first=" + first + " last=" + last);
        System.out.println("span=" + Duration.between(first, last).toMinutes() + "m'
        System.out.println("firstError=" + (firstError == null ? "-" : firstError));
    }
}
```

EXERCISE 2

Hex dump

Read all of standard input as raw bytes and print a hex dump, eight bytes per row: an 8-digit lower-case hex offset, two spaces, the bytes as two-digit hex separated by single spaces **padded to 23 characters**, two spaces, then the printable ASCII (`0x20` – `0x7e`) of the row between `|` bars with every other byte shown as `.`. Finally print `total=<byte count>`.

Example (input `Hello, World!` followed by a newline — 14 bytes) →

```
00000000  48 65 6c 6c 6f 2c 20 57  |Hello, W|
00000008  6f 72 6c 64 21 0a        |orld!.|
total=14
```

INPUT 1

```
Hello, World!
```

OUTPUT 1

```
00000000  48 65 6c 6c 6f 2c 20 57
|Hello, W|
00000008  6f 72 6c 64 21 0a
|orld!.|
total=14
```

INPUT 2

```
ABCDEFGH
```

OUTPUT 2

```
00000000  41 42 43 44 45 46 47 48
|ABCDEFGH|
total=8
```

REFERENCE SOLUTION

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = System.in.readAllBytes();
        for (int off = 0; off < data.length; off += 8) {
            StringBuilder hex = new StringBuilder();
            StringBuilder ascii = new StringBuilder();
            for (int i = off; i < Math.min(off + 8, data.length); i++) {
                int b = data[i] & 0xff;
                if (hex.length() > 0) hex.append(' ');
                hex.append(String.format("%02x", b));
                ascii.append(b ⩾ 0x20 && b ⩽ 0x7e ? (char) b : '.');
            }
            while (hex.length() < 23) hex.append(' ');
            System.out.println(String.format("%08x", off) + "  " + hex + "  |" + asc
        }
        System.out.println("total=" + data.length);
    }
}
```

EXERCISE 3

CSV to a fixed-width report

Read CSV lines `name,qty,price` until end of input; the name may be quoted and contain commas (RFC 4180 rules, doubled quotes for a literal quote). Print one line per record formatted `%-14s%5d%10.2f` (name left-aligned in 14, quantity right-aligned in 5, line total `qty × price` in 10 with two decimals), then a final line in the same format with the name `TOTAL`, the total quantity and the grand total. Use `Locale.ROOT`.

Example input

```
Widget,2,3.5
"Gear, large",1,10
```

→

```
Widget            2      7.00
Gear, large       1     10.00
TOTAL             3     17.00
```

INPUT 1

```
Widget,2,3.5
"Gear, large",1,10
```

OUTPUT 1

```
Widget            2      7.00
Gear, large       1     10.00
TOTAL             3     17.00
```

INPUT 2

```
"Bolt ""M8""",10,0.25
```

OUTPUT 2

```
Bolt "M8"        10      2.50
TOTAL            10      2.50
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static List<String> parse(String line) {
        List<String> fields = new ArrayList<>();
        StringBuilder cur = new StringBuilder();
        boolean quoted = false;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (quoted) {
                if (c == '"') {
                    if (i + 1 < line.length() && line.charAt(i + 1) == '"') { cur.ap
                    else quoted = false;
                } else cur.append(c);
            } else if (c == '"') quoted = true;
            else if (c == ',') { fields.add(cur.toString()); cur.setLength(0); }
            else cur.append(c);
        }
        fields.add(cur.toString());
        return fields;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        long totalQty = 0;
        double grand = 0;
        while ((line = in.readLine()) != null) {
            if (line.isBlank()) continue;
            List<String> f = parse(line);
            int qty = Integer.parseInt(f.get(1).trim());
            double price = Double.parseDouble(f.get(2).trim());
            double total = qty * price;
            totalQty += qty;
            grand += total;
            System.out.println(String.format(Locale.ROOT, "%-14s%5d%10.2f", f.get(0)
        }
        System.out.println(String.format(Locale.ROOT, "%-14s%5d%10.2f", "TOTAL", tot
    }
}
```

CHECKPOINT QUESTIONS

1. `Reader` / `Writer` differ from `InputStream` / `OutputStream` in that they…

A. Are faster

B. Move characters, decoded through a charset, rather than raw bytes

C. Are unbuffered

D. Cannot be closed

2. Predict: `"日本".length()` and its UTF-8 byte count

A. `2` and `2`

B. `2` and `6`

C. `6` and `6`

D. `4` and `6`

3. The single biggest cost of unbuffered I/O is…

A. Charset decoding

B. A system call per small read or write

C. Object allocation

D. Thread switching

4. Reading ten million integers fastest in Java uses…

A. `Scanner.nextInt`

B. A byte-level reader over `BufferedInputStream`, parsing digits by hand

C. `Files.readAllLines`

D. `String.split` per line

5. `base.resolve(userInput).normalize().startsWith(base)` is checked to prevent…

A. Slow paths

B. Path traversal — `..` or an absolute path escaping the trusted directory

C. Symlinks

D. Long file names

6. `Files.lines(path)` should be used…

A. Without closing

B. Inside try-with-resources — it holds the file open while the stream lives

C. Only for small files

D. Never

7. `Files.move(src, dst, ATOMIC_MOVE)` is atomic…

A. Always

B. On the same file system (a rename); across file systems it may fail

C. Never

D. Only for directories

8. During Java deserialisation of an ordinary `Serializable` class…

A. The no-arg constructor runs

B. No constructor of the serialisable class runs — fields are set directly

C. All constructors run

D. `readObject` is forbidden

9. A quote inside a quoted CSV field is written as…

A. `\"`

B. `""` — a doubled quote

C. `'`

D. It is not allowed

10. `Instant` represents…

A. A calendar date

B. A point on the UTC timeline

C. A wall-clock time in a zone

D. An amount of time

11. `LocalDateTime` is ambiguous as a moment because…

A. It has no seconds

B. It carries no zone or offset — 09:30 where?

C. It is mutable

D. It uses zero-based months

12. `ChronoUnit.DAYS.between(later, earlier)` returns…

A. A positive count

B. A negative count — it is signed, `earlier - later`

C. Zero

D. An exception

13. Formatting `EEE, dd MMM yyyy` without a `Locale` …

A. Always prints English

B. Prints names in the default locale — pass `Locale.ENGLISH` for stable output

C. Throws

D. Prints numbers only

14. `"IST"` as a zone id is a bad idea because…

A. It is too short

B. It is ambiguous (India, Ireland, Israel); use IANA region ids like `Asia/Kolkata`

C. It is deprecated

D. It has no DST

15. Making code that calls `LocalDate.now()` testable is done by…

A. Mocking the JVM clock

B. Injecting a `Clock` and calling `now(clock)`; tests use `Clock.fixed`

C. Sleeping

D. Using `Date`

ANSWER KEY · CHECKPOINT — I/O, FILES AND TIME

1. B `InputStreamReader` / `OutputStreamWriter` are the bridges.

2. B Two UTF-16 units, three bytes each in UTF-8.

3. B An 8 KB buffer amortises it across thousands of operations.

4. B Zero allocation per number; `BufferedReader` + `StringTokenizer` is the everyday compromise.

5. B An absolute argument replaces the base entirely; `..` climbs out of it.

6. B Same for `list`, `walk` and `find`.

7. B Cross-device moves copy then delete.

8. B Invariants are not re-checked — one reason records (canonical constructor) are safer.

9. B RFC 4180; a parser must treat `""` inside quotes as one literal quote.

10. B Timestamps and logs; convert to a zone only for display.

11. B Attach a zone (`atZone`) to get a `ZonedDateTime` before comparing with an `Instant`.

12. B Order the arguments, or take `Math.abs`.

13. B The judge's locale is not yours.

14. B Region ids carry the full rule history.

15. B Every `now()` overload accepts a `Clock` for this reason.
