Module 3 of 20

Strings

Immutability and the pool, comparison, the String API, StringBuilder, formatting and text blocks, Unicode, and regular expressions.

Text is most of what programs handle, and Java's String has a design — immutable, pooled, UTF-16 — whose consequences show up in every interview and every codebase: why == is wrong, why += in a loop is slow, why an emoji has length 2, why split drops trailing empties.

This module goes through the String class methodically: how strings are created and shared, how they are compared and ordered, the twenty methods that do most of the work and their edge cases, StringBuilder for building text, format strings and text blocks for producing it, characters and encodings underneath, and regular expressions for pattern work.

By the end you can manipulate text correctly and efficiently, and answer the string questions interviewers reach for first.

CONTENTS

1	Strings are immutable objects	13 min
2	Comparing strings — equals, compareTo and hashCode	12 min
3	The String API you will actually use	16 min
4	StringBuilder — building strings efficiently	12 min
5	Formatting output and text blocks	13 min
6	Characters, Unicode and encodings	14 min
7	Regular expressions in Java	16 min
✓	Checkpoint — Strings	25 min

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

LESSON 1 · 13 MIN

Strings are immutable objects

`String` is the most used class in Java and the one with the most surprising design: a string, once created, **cannot change**. Every method that seems to modify one — `toUpperCase`, `replace`, `trim`, `+` — returns a new string and leaves the original untouched. Understanding immutability, and the *string pool* that it makes possible, is the key to everything else in this module.

Creating strings

```
String a = "hello";                 // a literal — the normal way
String b = new String("hello");     // a new object — almost never what you w
String c = String.valueOf(42);      // "42"
String d = Integer.toString(42);    // "42"
String e = "" + 42;                 // "42" — works, slightly wasteful
String f = new String(charArray);   // from a char[]
String g = new String(bytes, StandardCharsets.UTF_8);   // decode bytes
String h = String.join(", ", "a", "b", "c");            // "a, b, c"
String i = "ab".repeat(3);          // "ababab" (Java 11+)
```

A `String` is a class in `java.lang` — a reference type. `String s;` as a field is `null`, not `""`. The empty string `""` is a real object with length 0; the two are different, and `s.length()` on `null` throws `NullPointerException`.

Immutability

```
String s = "hello";
s.toUpperCase();                    // returns "HELLO" — and throws it away
System.out.println(s);              // hello

s = s.toUpperCase();                // rebinds s to the new string
System.out.println(s);              // HELLO
```

The first form is one of the most common beginner bugs: calling a method for its effect when it has none. Every `String` method returns a value you must use.

Why immutable? Four reasons the designers gave, all still valid:

Safety — a string passed to a method, stored as a map key, or used as a file name cannot be changed under you. `HashMap` keys must not change after insertion, and strings are the most common key.

Sharing — because nobody can alter a string, the same object can be safely shared everywhere it is equal: the string pool, and `substring` before Java 7 (which shared the backing array).

Thread safety — immutable objects need no locks.

Cached hash code — `hashCode()` is computed once and stored, since the value can never change. Map lookups with string keys are fast for this reason.

The cost is allocation: building a string character by character with `+` creates a new object each time. That is what `StringBuilder` is for (lesson 4).

The string pool

```
String a = "java";
String b = "java";
String c = new String("java");
String d = c.intern();

a == b        // true  — both literals refer to the ONE pooled "java"
a == c        // false — new String made a separate object
a.equals(c)   // true  — same characters
a == d        // true  — intern() returns the pooled instance
```

Every string **literal** in your program is put in the *string pool* (a table inside the JVM) at class-loading time, and identical literals resolve to the same object. That is why `==` sometimes "works" on strings — and why relying on it is a bug: strings built at run time (read from input, concatenated from variables, `new String(...)`) are not pooled.

```
String x = "ja" + "va";              // compile-time constant folding → pooled
String y = "ja"; String z = y + "va";   // run-time concatenation → new object
final String w = "ja"; String v = w + "va";   // w is a constant → folded → v
```

`intern()` returns the pooled instance for any string, creating it if needed. It is used in a few memory-heavy systems to deduplicate; in ordinary code you never call it.

The rule for comparison is therefore simple and absolute: **`equals` for content, `==` only when you mean "the same object"** — which for strings is essentially never. The next lesson is entirely about this.

Length, indexing and the basic accessors

```
String s = "hello";
s.length()          // 5 — a method, not a field (arrays use .length without p
s.charAt(0)         // 'h'
s.charAt(5)         // StringIndexOutOfBoundsException — valid indices are 0..
s.isEmpty()         // false; true for ""
s.isBlank()         // true for "", "  ", "\t\n" (Java 11+)
s.toCharArray()     // a NEW char[] copy — modifying it does not touch s
```

Iterating over characters, the two idioms:

```
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
}
for (char c : s.toCharArray()) { }     // simpler, but copies the array once
```

Concatenation

`+` with a `String` operand converts the other operand with `String.valueOf` and joins. `null` becomes the four characters `"null"` — a common sight in logs. Since Java 9, `+` in a single expression compiles to an efficient `invokedynamic` call, so `a + b + c + d` is fine; **`+=` in a loop** is not, because each iteration copies the whole accumulated string (quadratic time).

```
String csv = "";
for (String item : items) csv += item + ",";     // O(n²) — copies grow every
```

Lesson 4 replaces this with `StringBuilder`; `String.join` and streams' `Collectors.joining` are the other answers.

`String` is `final`

You cannot subclass `String`. Combined with immutability, this guarantees that a `String` reference *always* behaves like a string — no subclass can override `equals` or `hashCode` to lie. This is the same reason wrappers are final.

`null` versus empty

```
String s = null;
s.length();                  // NullPointerException
"".length();                 // 0
s == null                    // the only safe test
"x".equals(s)                // false, no exception — "constant first" idiom
Objects.equals(s, "x")       // null-safe equals
Objects.requireNonNull(s, "name must not be null");
```

Writing `"literal".equals(variable)` rather than `variable.equals("literal")` avoids a `NullPointerException` when the variable is null. It is an idiom you will see everywhere in Java code.

Interview angle

"Why are strings immutable?" Safety as keys, sharing via the pool, thread safety, cached hash.

"How many objects does `new String("hi")` create?" Up to two: the pooled literal `"hi"` (if not already there) and the new object.

"`"a" + "b" == "ab"`?" True — constant folding at compile time puts it in the pool.

"Is `String` thread-safe?" Yes, because it is immutable.

Key takeaways

Strings never change; every method returns a new string — **use the return value**.

Literals live in the pool and share one object; run-time strings do not.

`equals` for content, never `==`.

`length()` is a method; indices run 0…length−1.

`+=` in a loop is quadratic — that is what `StringBuilder` fixes.

EXERCISE 1

Palindromes

Read an integer `n`, then `n` lines. For each line print `yes` if it reads the same forwards and backwards (exact character comparison, case-sensitive) and `no` otherwise. Compare characters with two indices walking inward — do not build a reversed copy.

Input: `n`, then `n` lines (a line may contain spaces). **Output:** `n` lines.

INPUT 1

```
3
level
Java
step on no pets
```

OUTPUT 1

```
yes
no
yes
```

INPUT 2

```
2
Aba
a
```

OUTPUT 2

```
no
yes
```

REFERENCE SOLUTION

```java
import java.io.*;

public class Main {
    static boolean isPalindrome(String s) {
        int lo = 0, hi = s.length() - 1;
        while (lo < hi) {
            if (s.charAt(lo) != s.charAt(hi)) return false;
            lo++;
            hi--;
        }
        return true;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        for (int i = 0; i < n; i++) {
            System.out.println(isPalindrome(br.readLine()) ? "yes" : "no");
        }
    }
}
```

EXERCISE 2

Proof of immutability

Read one word `w` (no spaces). Call `w.toUpperCase()` **without** assigning the result and print `after call: <w>`; then assign `String up = w.toUpperCase()` and print `assigned: <up>`; call `w.concat("!")` unassigned and print `still: <w>`; finally print `trim returns same object: <w == w.trim()>` — `trim` hands back the very same object when there is nothing to remove.

Example: `java` →

```
after call: java
assigned: JAVA
still: java
trim returns same object: true
```

INPUT 1

```
java
```

OUTPUT 1

```
after call: java
assigned: JAVA
still: java
trim returns same object: true
```

INPUT 2

```
Hello
```

OUTPUT 2

```
after call: Hello
assigned: HELLO
still: Hello
trim returns same object: true
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String w = new Scanner(System.in).next();
        w.toUpperCase();
        System.out.println("after call: " + w);
        String up = w.toUpperCase();
        System.out.println("assigned: " + up);
        w.concat("!");
        System.out.println("still: " + w);
        System.out.println("trim returns same object: " + (w == w.trim()));
    }
}
```

QUIZ

1. `String s = "hi"; s.toUpperCase(); System.out.println(s);` prints…

A. `HI`

B. `hi` — the result of `toUpperCase` was discarded

C. `Hi`

D. A compile error

2. `String a = "java"; String b = "java"; a == b` is…

A. `true` — both literals refer to the same pooled object

B. `false` — different variables

C. Undefined

D. A compile error

3. Which is a reason strings are immutable?

A. It makes concatenation faster

B. Safe use as `HashMap` keys and sharing via the pool

C. Strings would otherwise not fit in memory

D. The JVM cannot modify heap objects

4. What does `"x" + null` produce?

A. `NullPointerException`

B. `"x"`

C. `"xnull"`

D. Compile error

5. Why is `s += x` inside a loop of 100 000 iterations slow?

A. Strings are stored on the stack

B. Each `+=` copies the whole accumulated string into a new one — quadratic total work

C. The JIT cannot compile loops with strings

D. It is not slow

ANSWER KEY · STRINGS ARE IMMUTABLE OBJECTS

1. B Strings are immutable; `toUpperCase` returns a new string and leaves `s` alone. `s = s.toUpperCase()` is what was meant.

2. A Identical string literals resolve to one object in the string pool. It is still wrong to rely on — a string read at run time is not pooled.

3. B A key that could change would break hashing; sharing one object for equal literals is only safe if nobody can alter it. Concatenation is actually *slower* because of immutability.

4. C Concatenation converts operands with `String.valueOf`, which turns `null` into the text `"null"`. Calling a method on a null string is what throws.

5. B Immutability means every concatenation allocates a new string of the full length so far. `StringBuilder.append` grows a buffer in place instead.

LESSON 2 · 12 MIN

Comparing strings — equals, compareTo and hashCode

Comparing strings correctly is a rite of passage: every Java programmer has shipped a `==` that worked in tests and failed with real input. This lesson makes the rules precise, and then goes further into ordering (`compareTo`), case-insensitive comparison, and the `hashCode` contract that lets strings work as map keys.

`==` compares references

```
String input = new Scanner(System.in).next();   // user typed "yes"
if (input == "yes") { … }                        // FALSE even when they typed
if (input.equals("yes")) { … }                   // correct
```

`==` on two references asks "are these the same object?" — and a string read from input, built by concatenation, or returned by most methods is a fresh object, never the pooled literal. The comparison compiles, runs, and is silently wrong.

`equals`: content, case-sensitive

```
"Java".equals("Java")        // true
"Java".equals("java")        // false
"Java".equals(null)          // false — never throws
"Java".equalsIgnoreCase("JAVA")   // true
```

`String.equals` checks: same object → true; not a `String` → false; different length → false; then compares character by character. It is `O(n)` in the worst case and short-circuits on the first difference. Its `null` handling is why the idiom is `"constant".equals(variable)`, or `Objects.equals(a, b)` when both may be null.

`equalsIgnoreCase` compares after folding case per character (using both `Character.toUpperCase` and `toLowerCase`, which handles most alphabets). For locale-sensitive folding — Turkish dotless i, German ß — use a `Collator`; for programming identifiers and protocol keywords, `equalsIgnoreCase` is right.

`compareTo`: lexicographic ordering

`String implements Comparable<String>`, so strings sort naturally:

```
"apple".compareTo("banana")   // negative (a < b)
"banana".compareTo("apple")   // positive
"apple".compareTo("apple")    // 0
"apple".compareTo("applesauce")   // negative: shorter prefix sorts first (re
"Zebra".compareTo("apple")    // negative! 'Z' (90) < 'a' (97) — uppercase so
"10".compareTo("9")           // negative: '1' < '9' — strings compare charac
```

The rule: compare characters at each index by their UTF-16 code unit values; at the first difference return `this.charAt(i) - other.charAt(i)`; if one is a prefix of the other return the length difference. The sign is all you should rely on — the magnitude is an implementation detail.

Consequences you must know:

Uppercase before lowercase (ASCII order). `compareToIgnoreCase` fixes that for sorting names.

Digits compare as characters: `"10" < "9"`. To sort numerically, parse, or use a comparator on `Integer.parseInt`.

Accented characters sort by code point, which is rarely what a user expects; `java.text.Collator` gives locale-correct ordering.

`Collections.sort(list)` and `Arrays.sort(strings)` use `compareTo`; `list.sort(String.CASE_INSENSITIVE_ORDER)` uses the case-insensitive comparator built into `String`.

`hashCode`: the map-key contract

```
"hello".hashCode()      // 99162322
```

`String.hashCode()` is `s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]`, computed with `int` overflow — specified in the Javadoc, so it is the same on every JVM. It is cached in the object after the first call.

The contract that every `equals` override must obey (Module 8) is visible here: two strings that are `equals` always have the same `hashCode`, which is what lets `HashMap<String, V>` find a

key that was inserted as a different object with the same characters. `"hello".hashCode() == new String("hello").hashCode()` — always.

Because 31 is small and the arithmetic wraps, collisions exist (`"Aa"` and `"BB"` both hash to 2112); `HashMap` handles collisions, so this is trivia, not a problem — until someone sends a million colliding keys as a denial-of-service, which is why Java 8 turned overfull buckets into trees.

`contentEquals`, `regionMatches`, `matches`

`s.contentEquals(sb)` — compare a `String` with a `StringBuilder` or any `CharSequence` without converting.

`s.regionMatches(ignoreCase, offset, other, otherOffset, len)` — compare a substring without allocating one.

`s.matches(regex)` — full-match against a regular expression (lesson 7). `"abc".matches("a.*")` is true; note it matches the *whole* string.

`s.startsWith(prefix)`, `s.endsWith(suffix)`, `s.contains(sub)` — the everyday tests; `contains` is `indexOf(sub) ≥ 0`.

`switch` on strings

```
switch (command) {
    case "start" → start();
    case "stop"  → stop();
    default      → System.out.println("unknown: " + command);
}
```

Since Java 7, `switch` accepts a `String`. It is compiled into a `hashCode` lookup followed by an `equals` check, so it is content-based and case-sensitive, and a `null` selector throws `NullPointerException`.

A comparison checklist

Want	Use
Same characters	`a.equals(b)`
Same characters, any case	`a.equalsIgnoreCase(b)`
Either may be null	`Objects.equals(a, b)`
Ordering	`a.compareTo(b)`, or `String.CASE_INSENSITIVE_ORDER`
Locale-correct ordering	`Collator.getInstance(locale).compare(a, b)`
Same object	`a == b` — and ask yourself why

Interview angle

"`String s = "hi"; String t = new String("hi"); s == t`?" False. *"`s.equals(t)`?"* True. *"`s == t.intern()`?"* True.

"What does `"apple".compareTo("banana")` return?" A negative number (−1: 'a' − 'b'). Only the sign is contractual.

"Why must `equals` and `hashCode` agree?" Hash-based collections find the bucket by `hashCode` and then the entry by `equals`; disagreement makes keys unfindable.

"Is `switch` on a string by reference?" No — by `hashCode` then `equals`.

Key takeaways

`==` compares references and is wrong for strings; `equals` compares content.

`"literal".equals(var)` and `Objects.equals` are null-safe.

`compareTo` is character-by-character by code unit: uppercase first, digits as text.

`hashCode` is specified, cached, and agrees with `equals` — the basis of `HashMap` keys.

`switch` on strings is content-based and throws on `null`.

EXERCISE 1

First and last in order

Read three words (whitespace-separated). Print two lines:

`strict: <first> <last>` — the smallest and largest by `compareTo`.

`ignoreCase: <first> <last>` — the smallest and largest by `compareToIgnoreCase`.

Print the words as they were given (do not change their case).

Example: `banana Apple cherry` →

```
strict: Apple cherry
ignoreCase: Apple cherry
```

and `zebra apple Mango` →

```
strict: Mango zebra
ignoreCase: apple zebra
```

INPUT 1

```
banana Apple cherry
```

OUTPUT 1

```
strict: Apple cherry
ignoreCase: Apple cherry
```

INPUT 2

```
zebra apple Mango
```

OUTPUT 2

```
strict: Mango zebra
ignoreCase: apple zebra
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String a = in.next(), b = in.next(), c = in.next();

        String min = a, max = a;
        if (b.compareTo(min) < 0) min = b;
        if (c.compareTo(min) < 0) min = c;
        if (b.compareTo(max) > 0) max = b;
        if (c.compareTo(max) > 0) max = c;
        System.out.println("strict: " + min + " " + max);

        String imin = a, imax = a;
        if (b.compareToIgnoreCase(imin) < 0) imin = b;
        if (c.compareToIgnoreCase(imin) < 0) imin = c;
        if (b.compareToIgnoreCase(imax) > 0) imax = b;
        if (c.compareToIgnoreCase(imax) > 0) imax = c;
        System.out.println("ignoreCase: " + imin + " " + imax);
    }
}
```

EXERCISE 2

Two orders, one list

Read an integer `n` and `n` words. Print `natural: <words sorted with the natural String order>`, then `ignoreCase: <words sorted with String.CASE_INSENSITIVE_ORDER>` (the sort is stable, so words equal ignoring case keep their input order), then `distinctIgnoreCase: <how many distinct words there are when case is ignored>` using a `TreeSet` built with that comparator. Join words with spaces.

Example: `4` then `banana Apple apple Cherry` →

```
natural: Apple Cherry apple banana
ignoreCase: Apple apple banana Cherry
distinctIgnoreCase: 3
```

INPUT 1

```
4
banana Apple apple Cherry
```

OUTPUT 1

```
natural: Apple Cherry apple banana
ignoreCase: Apple apple banana Cherry
distinctIgnoreCase: 3
```

INPUT 2

```
3
b B a
```

OUTPUT 2

```
natural: B a b
ignoreCase: a b B
distinctIgnoreCase: 2
```

REFERENCE SOLUTION

```
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());
        List<String> natural = new ArrayList<>(words);
        Collections.sort(natural);
        System.out.println("natural: " + String.join(" ", natural));
        List<String> ignore = new ArrayList<>(words);
        ignore.sort(String.CASE_INSENSITIVE_ORDER);
        System.out.println("ignoreCase: " + String.join(" ", ignore));
        Set<String> distinct = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
        distinct.addAll(words);
        System.out.println("distinctIgnoreCase: " + distinct.size());
    }
}
```

QUIZ

1. `String s = new Scanner(System.in).next();` — the user types `yes`. `s == "yes"` is…

A. `true`

B. `false` — the input string is a new object, not the pooled literal

C. `true` only after `trim()`

D. Compile error

2. `"Zebra".compareTo("apple")` is…

A. Positive, because Z comes after a alphabetically

B. Negative, because `'Z'` (90) is less than `'a'` (97)

C. 0

D. An exception

3. Why is `"literal".equals(variable)` preferred over `variable.equals("literal")`?

A. It is faster

B. It cannot throw `NullPointerException` when `variable` is null

C. The compiler requires it

D. It ignores case

4. Two strings are `equals`. What must be true of their `hashCode`s?

A. They may differ

B. They are equal

C. They are equal only if both are interned

D. `hashCode` is not defined for strings

5. `switch (s) { case "a" → … }` with `s == null` does what?

A. Runs the default branch

B. Throws `NullPointerException`

C. Matches `case "null"`

D. Compile error

ANSWER KEY · COMPARING STRINGS — EQUALS, COMPARETO AND HASHCODE

1. B Strings created at run time are not interned. `==` compares references. `s.equals("yes")` is the correct test.

2. B `compareTo` compares UTF-16 code units; all uppercase letters precede all lowercase ones. `compareToIgnoreCase` or `String.CASE_INSENSITIVE_ORDER` gives the human order.

3. B Calling a method on a null reference throws; `equals` on a literal simply returns false for a null argument. `Objects.equals(a, b)` is the symmetric null-safe form.

4. B The `equals` / `hashCode` contract: equal objects must have equal hash codes. `String.hashCode` is computed from the characters, so this holds — and `HashMap` depends on it.

5. B A string `switch` starts by calling `hashCode()` on the selector, which throws for null. Guard with `if (s == null)` first.

LESSON 3 · 16 MIN

The String API you will actually use

`String` has around seventy methods. Perhaps twenty carry ninety percent of real code, and each has a detail — an exclusive end index, a regex where you expected a literal, a trailing-empty-string rule — that separates "I know the method" from "I know what it does". This lesson is those twenty, with the details.

Substrings and searching

```
String s = "Hello, World";
s.substring(7)          // "World"          — from index 7 to the end
s.substring(0, 5)       // "Hello"          — begin inclusive, END EXCLUSIVE
s.substring(5, 5)       // ""               — legal: empty
s.substring(7, 5)       // StringIndexOutOfBoundsException — begin > end
s.substring(0, 20)      // exception — end past length
s.indexOf('o')          // 4                — first occurrence
s.indexOf('o', 5)       // 8                — search from index 5
s.indexOf("World")      // 7
s.indexOf("xyz")        // -1               — not found: -1, never an excepti
s.lastIndexOf('o')      // 8
s.contains("lo, W")     // true
s.startsWith("Hell")    // true
s.startsWith("World", 7)   // true — with an offset
s.endsWith("ld")        // true
s.charAt(s.length() - 1)   // 'd' — the last character
```

The half-open interval `[begin, end)` is Java's convention everywhere (arrays, lists, streams), and it makes `end - begin` the length. Pair it with `indexOf`:

```
int at = s.indexOf(',');
String first = s.substring(0, at);        // "Hello"
String rest  = s.substring(at + 1).trim(); // "World"
```

Case, whitespace, and cleaning

```
"  padded \t\n".trim()     // "padded"   — removes chars ≤ U+0020 at both ends
"  padded ".strip()        // "padded"   — Unicode-aware (Java 11+); also stri
"Hello".toUpperCase()      // "HELLO"
"Hello".toLowerCase()      // "hello"
"hello".isBlank()          // false; "   ".isBlank() is true
"ab".repeat(3)             // "ababab"
```

`toUpperCase()` / `toLowerCase()` use the default locale — in Turkish, `"i".toUpperCase()` is `İ`. For programmatic text (keywords, identifiers, file extensions) pass `Locale.ROOT`: `s.toLowerCase(Locale.ROOT)`. Static analysers flag the no-argument form for this reason.

Replacing

```
"a.b.c".replace('.', '-')          // "a-b-c"  — char, all occurrences
"a.b.c".replace(".", "-")          // "a-b-c"  — LITERAL CharSequence, all occ
"a.b.c".replaceAll(".", "-")       // "-----"  — REGEX: . matches any characte
"a.b.c".replaceAll("\\.", "-")     // "a-b-c"  — escaped regex
"a.b.c".replaceFirst("\\.", "-")   // "a-b.c"
```

The trap is in the names: `replace` is literal, `replaceAll` is a regular expression. Use `replace` unless you need a pattern. When you do need a pattern with a literal that has special characters, `Pattern.quote(".")` produces the escaped form.

Splitting and joining

```
"a,b,c".split(",")            // ["a", "b", "c"]
"a,b,,c,,".split(",")         // ["a", "b", "", "c"]   — trailing empties DROP
"a,b,,c,,".split(",", -1)     // ["a", "b", "", "c", "", ""]  — limit -1 keeps
"a,b,c".split(",", 2)         // ["a", "b,c"]          — at most 2 pieces
"one  two   three".split(" ")     // ["one", "", "two", "", "", "three"]   — e
"one  two   three".split("\\s+")  // ["one", "two", "three"]  — runs of whites
"  lead".split("\\s+")            // ["", "lead"]  — a leading empty string IS
"a.b".split(".")              // []  — "." is a regex matching everything; eve
"a.b".split("\\.")            // ["a", "b"]
"".split(",")                 // [""]  — one empty string, not an empty array

String.join("-", "a", "b", "c")           // "a-b-c"
String.join(", ", List.of("x", "y"))      // "x, y"
```

`split` takes a **regex**, drops **trailing** empty strings (unless the limit is negative), and keeps leading ones. The safe form for whitespace-separated input is `line.trim().split("\\s+")`. For a single literal character that is not a regex metacharacter (`,`, `;`, `|` is a metacharacter!, `-`), plain `split(",")` is fine; for `.`, `|`, `+`, `*`, `?`, `(`, `[`, `\`, `^`, `$` escape with `\\`.

Conversion to and from other types

```
String.valueOf(3.5)         // "3.5"   — overloaded for every primitive, and Ob
String.valueOf(chars)       // from char[]
Integer.parseInt("42")      // 42; NumberFormatException for "42 ", "4.2", "",
Integer.parseInt("ff", 16) // 255
Double.parseDouble("1e3")  // 1000.0
Boolean.parseBoolean("yes")   // false — only "true" (any case) is true
s.toCharArray()             // char[] copy
s.chars()                   // IntStream of UTF-16 code units (Module 15)
s.getBytes(StandardCharsets.UTF_8)   // byte[] — always name the charset
```

Comparison helpers (recap)

`equals`, `equalsIgnoreCase`, `compareTo`, `compareToIgnoreCase`, `contentEquals`, `regionMatches`, `matches`. Covered in the previous lesson.

Newer additions worth knowing

Method	Since	What
`isBlank()`, `strip()`, `stripLeading()`, `stripTrailing()`	11	Unicode whitespace handling
`repeat(n)`	11	Repetition
`lines()`	11	Stream of lines, splitting on `\n`, `\r`, `\r\n`
`chars()`, `codePoints()`	9	Streams of characters
`formatted(args…)`	15	`"%d items".formatted(3)` — instance form of `String.format`
`indent(n)`	12	Adds/removes leading spaces on every line
`translateEscapes()`	15	Interprets `\n`, `\t` in the string's content
Text blocks `"""…"""`	15	Multi-line literals (next lessons)

Performance notes

`substring` copies (since Java 7u6). `s.substring(i)` in a loop over `i` is O(n²).

`indexOf` / `contains` are linear; for many searches in a large text, consider one pass with a `Matcher` or your own scan.

`split` compiles a regex on every call for patterns longer than one non-meta character; in a hot loop, precompile with `Pattern.compile(...).split(s)`.

`+` in a single expression is fine; `+=` in a loop is not.

Common exceptions

`StringIndexOutOfBoundsException` — `charAt` / `substring` with a bad index. Message: `begin 0, end 20, length 12`.

`NullPointerException` — calling anything on a `null` reference.

`NumberFormatException` — parsing non-numeric text. Message: `For input string: "4.2"`.

`PatternSyntaxException` — a broken regex in `split` / `replaceAll` / `matches`, e.g. `split("(")`.

Interview angle

- *"`"a,b,,".split(",")` length?"* 2 — trailing empties are dropped.
- *"`replace` vs `replaceAll`?"* Literal vs regex; both replace every occurrence.
- *"Is `substring` O(1)?"* Not since Java 7u6 — it copies.
- *"How do you reverse a string?"* `new StringBuilder(s).reverse().toString()` (next lesson), or a two-pointer loop over a `char[]`.

Key takeaways

- `substring(begin, end)` — end is exclusive; `indexOf` returns −1, never throws.
- `replace` is literal; `replaceAll` / `split` / `matches` take regexes — escape `.` `|` `+` `*` `?`.
- `split` drops trailing empty strings; `trim().split("\\s+")` for whitespace.
- Pass `Locale.ROOT` to case conversions in programmatic text.
- Every method returns a new string — capture the result.

EXERCISE 1

Email parts

Read an integer `n`, then `n` email addresses, one per line. For each, print `user=<user> domain=<domain> tld=<tld>` where `user` is everything before the `@`, `domain` everything after it, and `tld` the part of the domain after its **last** dot. If the line has no `@`, or the domain has no dot, print `invalid`.

Use `indexOf`, `lastIndexOf` and `substring`. Trim each line first.

Example: `ada@mail.example.com` → `user=ada domain=mail.example.com tld=com`

INPUT 1

```
3
ada@mail.example.com
bob@localhost
not-an-email
```

OUTPUT 1

```
user=ada domain=mail.example.com tld=com
invalid
invalid
```

INPUT 2

```
2
  x@y.io
first.last@corp.co.uk
```

OUTPUT 2

```
user=x domain=y.io tld=io
user=first.last domain=corp.co.uk tld=uk
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            String line = br.readLine().trim();
            int at = line.indexOf('@');
            if (at == -1) {
                out.append("invalid\n");
                continue;
            }
            String user = line.substring(0, at);
            String domain = line.substring(at + 1);
            int dot = domain.lastIndexOf('.');
            if (dot == -1) {
                out.append("invalid\n");
                continue;
            }
            String tld = domain.substring(dot + 1);
            out.append("user=").append(user).append(" domain=").append(domain).apper
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Slug maker

Read lines until end of input; each is a title. Turn it into a URL slug: trim, lower-case (`Locale.ROOT`), replace every run of characters that are not `a-z` or `0-9` with a single `-`, and strip any leading or trailing `-`. Print `slug=<slug>` per line, or `slug=(empty)` when nothing is left.

Example input

```
  Hello, World!
Java 17: What's New?
```

→

```
slug=hello-world
slug=java-17-what-s-new
```

INPUT 1

```
  Hello, World!
Java 17: What's New?
```

OUTPUT 1

```
slug=hello-world
slug=java-17-what-s-new
```

INPUT 2

```
---
A
```

OUTPUT 2

```
slug=(empty)
slug=a
```

REFERENCE SOLUTION

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

public class Main {
    static String slug(String title) {
        String s = title.trim().toLowerCase(Locale.ROOT).replaceAll("[^a-z0-9]+", "-
        if (s.startsWith("-")) s = s.substring(1);
        if (s.endsWith("-")) s = s.substring(0, s.length() - 1);
        return s;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
            String s = slug(line);
            System.out.println("slug=" + (s.isEmpty() ? "(empty)" : s));
        }
    }
}
```

QUIZ

1. `"Hello, World".substring(7, 12)` is…

   A. `"World"`

   B. `" World"`

   C. `"Worl"`

   D. An exception

2. `"a,b,,c,,".split(",").length` is…

   A. 6

   B. 4

   C. 5

   D. 3

3. `"a.b.c".replaceAll(".", "-")` produces…

   A. `"a-b-c"`

   B. `"-----"`

   C. `"a.b.c"`

   D. An exception

4. `"abc".indexOf("z")` returns…

   A. 0

   B. -1

   C. `null`

   D. Throws `StringIndexOutOfBoundsException`

5. Why pass `Locale.ROOT` to `toLowerCase()` for keywords and file extensions?

A. It is faster

B. The default locale can change the result (Turkish `I` → dotless `ı`)

C. `toLowerCase()` without a locale is deprecated

D. It handles emoji

ANSWER KEY · THE STRING API YOU WILL ACTUALLY USE

1. A Begin is inclusive (index 7 = 'W'), end is exclusive (index 12 = one past 'd'). The length of the result is `end - begin` = 5.

2. B Trailing empty strings are removed: `["a", "b", "", "c"]`. Pass a negative limit (`split(",", -1)`) to keep them.

3. B `replaceAll` takes a regex, and `.` matches any character. Use `replace(".", "-")` (literal) or escape: `"\\."`.

4. B Searching never throws; a miss is −1. Only indexing (`charAt`, `substring`) with a bad index throws.

5. B Case mapping is locale-sensitive. Programmatic text should be mapped with `Locale.ROOT` so `"TITLE".toLowerCase()` is the same on every machine.

LESSON 4 · 12 MIN

StringBuilder — building strings efficiently

A `String` cannot change, so building one incrementally means copying: `s += "x"` allocates a new string holding all of `s` plus one character, every time. For ten pieces nobody notices; for a hundred thousand it is the difference between milliseconds and minutes. `StringBuilder` is the mutable buffer that fixes this — and the class every "reverse a string" and "build the output" answer uses.

The problem, measured

```
String s = "";
for (int i = 0; i < 100_000; i++) {
    s += i;              // copies i digits' worth of characters each time
}
```

Iteration *k* copies a string of length proportional to *k*, so the total work is 1 + 2 + … + n ≈ n²/2. At n = 100 000 that is billions of character copies and gigabytes of garbage. The same with a builder:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100_000; i++) {
    sb.append(i);        // appends into a growable char array
}
String s = sb.toString();
```

Linear time. The builder keeps a backing array with spare capacity and doubles it when full (amortised O(1) per append).

The API

```
StringBuilder sb = new StringBuilder();           // capacity 16
StringBuilder sb = new StringBuilder(1024);       // pre-sized when you know r
StringBuilder sb = new StringBuilder("seed");

sb.append("text");            // overloaded: String, char, int, long, double,
sb.append(42).append(' ').append(3.5);   // returns this — chain freely
sb.insert(0, "start ");       // insert at an index
sb.replace(0, 5, "begin");    // replace [begin, end) with a string
sb.delete(0, 6);              // remove [begin, end)
sb.deleteCharAt(sb.length() - 1);   // drop the last char — e.g. a trailing co
sb.setCharAt(0, 'B');         // mutate one char
sb.setLength(0);              // clear (keeps capacity) — reuse the builder
sb.setLength(3);              // truncate
sb.reverse();                 // in place
sb.length(); sb.charAt(i); sb.indexOf("x"); sb.substring(1, 3);   // like Stri
sb.toString();                // the String — copies the used part of the buff
sb.capacity();                // current buffer size (rarely needed)
```

`append(char)` versus `append(String)`: `sb.append('a')` and `sb.append("a")` both work; the `char` form avoids creating a string. `sb.append(c + 1)` where `c` is a `char` appends a *number* (promotion!) — write `sb.append((char) (c + 1))`.

The idioms

Build output once, print once (from Module 1):

```
StringBuilder out = new StringBuilder();
for (int i = 0; i < n; i++) out.append(values[i]).append('\n');
System.out.print(out);
```

Join with a separator, without a trailing one:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < items.length; i++) {
    if (i > 0) sb.append(", ");
    sb.append(items[i]);
}
// or: String.join(", ", items)  /  StringJoiner
```

Reverse:

```
String reversed = new StringBuilder(s).reverse().toString();
```

`reverse()` handles surrogate pairs correctly (an emoji stays an emoji), which a hand-written `char` loop does not.

Palindrome check without allocating: two indices walking inward on the string itself — no builder needed. Reach for a builder when you *produce* text, not when you *inspect* it.

Build a string with a computed length — `repeat` and `String.valueOf(char[])` are alternatives:

```
String dashes = "-".repeat(40);
char[] row = new char[n]; Arrays.fill(row, '*'); String stars = new String(row
```

`StringBuilder` vs `StringBuffer`

`StringBuffer` is the Java 1.0 original with every method `synchronized`. `StringBuilder` (Java 5) is the same API without the locks and is what you use. The synchronisation in `StringBuffer` is nearly useless anyway — thread safety of individual appends does not make a sequence of appends atomic. The interview answer: "same API; `StringBuffer` is synchronised and slower; use `StringBuilder` unless a builder is genuinely shared across threads, which it should not be."

`StringJoiner` and `String.join`

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
sj.add("a").add("b");
sj.toString();                          // "[a, b]"
sj.setEmptyValue("(none)");             // what toString gives with no elements
String.join("/", parts);                // a StringJoiner underneath
list.stream().collect(Collectors.joining(", "));   // the stream form (Module
```

When `+` is fine

The compiler turns `"x=" + x + ", y=" + y` into a single efficient concatenation (an `invokedynamic` call since Java 9; a `StringBuilder` chain before). A builder is not faster for

one expression — it is faster for *loops* and for *conditional* building. Rules of thumb: `+` for a single expression; `StringBuilder` when appending in a loop or across several statements; `String.join` / `Collectors.joining` for joining a collection.

Equality on builders

```
StringBuilder a = new StringBuilder("x"), b = new StringBuilder("x");
a.equals(b)                  // false — StringBuilder does NOT override equals
a.toString().equals(b.toString())   // true
a.compareTo(b)               // 0 — Comparable since Java 11
"x".contentEquals(a)         // true
```

A `StringBuilder` as a map key or in a set is a bug: identity-based, and mutable.

Capacity and memory

The builder grows by `(old + 1) * 2` when full. If you know the final size — building a fixed-width table, encoding a known-length message — pass it to the constructor and avoid the regrowth copies. After `toString()`, the builder still holds its buffer; drop the reference or `setLength(0)` to reuse.

Interview angle

"Why is `+=` in a loop slow?" Each concatenation copies the whole accumulated string: O(n²) total.

"`StringBuilder` vs `StringBuffer`?" Same API; buffer is synchronised; builder is faster and the default.

"How do you reverse a string?" `new StringBuilder(s).reverse().toString()`; mention the two-pointer `char[]` alternative and surrogate pairs if pressed.

"Does `StringBuilder.equals` compare contents?" No — identity. Use `compareTo` or `toString().equals`.

Key takeaways

`String` concatenation in a loop is quadratic; `StringBuilder.append` is amortised linear.

`append` returns `this` — chain; `setLength(0)` clears; `reverse()` is surrogate-safe.

`+` in a single expression is already efficient — the builder is for loops and conditional assembly.

`StringBuffer` is the synchronised, slower ancestor; `StringBuilder` is the one to use.

Builders compare by identity; convert to `String` for `equals` and as map keys.

EXERCISE 1

Run-length encoding

Read one line and print its run-length encoding: each maximal run of the same character becomes the character followed by the run length. Build the result with a `StringBuilder`.

Input: one non-empty line of letters. **Output:** one line.

Example: `aaabccdddd` → `a3b1c2d4`

INPUT 1

```
aaabccdddd
```

OUTPUT 1

```
a3b1c2d4
```

INPUT 2

```
abc
```

OUTPUT 2

```
a1b1c1
```

REFERENCE SOLUTION

```java
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s = br.readLine();
        StringBuilder sb = new StringBuilder();
        int i = 0;
        while (i < s.length()) {
            char c = s.charAt(i);
            int j = i;
            while (j < s.length() && s.charAt(j) == c) j++;
            sb.append(c).append(j - i);
            i = j;
        }
        System.out.println(sb);
    }
}
```

EXERCISE 2

Devowel, then mirror

Read one line of words separated by single spaces. Print two lines: first the sentence with every vowel (`aeiouAEIOU`) removed — build it with a `StringBuilder` , appending only consonants; second the original sentence with **each word reversed in place** (word order unchanged) using `new StringBuilder(word).reverse()` . Words that lose all their letters disappear from the first line (collapse the spaces).

Example: `Java is fun` →

```
Jv s fn
avaJ si nuf
```

INPUT 1

```
Java is fun
```

OUTPUT 1

```
Jv s fn
avaJ si nuf
```

INPUT 2

```
a e i
```

OUTPUT 2

```

a e i
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String line = new Scanner(System.in).nextLine();
        StringBuilder noVowels = new StringBuilder();
        StringBuilder mirrored = new StringBuilder();
        for (String word : line.trim().split(" ")) {
            StringBuilder w = new StringBuilder();
            for (char c : word.toCharArray()) {
                if ("aeiouAEIOU".indexOf(c) < 0) w.append(c);
            }
            if (w.length() > 0) {
                if (noVowels.length() > 0) noVowels.append(' ');
                noVowels.append(w);
            }
            if (mirrored.length() > 0) mirrored.append(' ');
            mirrored.append(new StringBuilder(word).reverse());
        }
        System.out.println(noVowels);
        System.out.println(mirrored);
    }
}
```

QUIZ

1. What is the main difference between `StringBuilder` and `StringBuffer`?

A. `StringBuffer` is immutable

B. `StringBuffer`'s methods are synchronised; `StringBuilder`'s are not and it is faster

C. `StringBuilder` cannot be reversed

D. `StringBuffer` is newer

2. `StringBuilder a = new StringBuilder("x"), b = new StringBuilder("x"); a.equals(b)` is…

A. `true`

B. `false` — `StringBuilder` does not override `equals`

C. A compile error

D. `true` after `toString()` is called on both automatically

3. `sb.append('a' + 1)` appends…

A. `"b"`

B. `"98"`

C. `"a1"`

D. Nothing — compile error

4. How do you clear a `StringBuilder` for reuse?

A. `sb.clear()`

B. `sb.setLength(0)`

C. `sb = null`

D. `sb.delete()`

5. When is plain `+` concatenation perfectly fine?

A. Never — always use a builder

B. In a single expression such as `"x=" + x + ", y=" + y`

C. Only for two operands

D. Only with literals

ANSWER KEY · STRINGBUILDER — BUILDING STRINGS EFFICIENTLY

1. B Same API, different locking. `StringBuffer` (Java 1.0) synchronises every call; `StringBuilder` (Java 5) is the default choice.

2. B `StringBuilder` inherits `Object.equals` (identity). Compare with `a.toString().equals(b.toString())` or `a.compareTo(b) == 0`.

3. B `'a' + 1` is an `int` (98), so the `append(int)` overload runs. Cast: `sb.append((char)('a' + 1))`.

4. B `setLength(0)` empties it and keeps the buffer. There is no `clear()`; `delete(0, sb.length())` also works but is longer.

5. B The compiler turns one concatenation expression into one efficient operation. The builder matters for loops and multi-statement assembly.

LESSON 5 · 13 MIN

Formatting output and text blocks

Getting text to look right — two decimals, aligned columns, a leading zero, a thousands separator, a multi-line template — is a solved problem in Java, but the solution is a mini-language you have to learn once. This lesson covers `String.format` / `printf` thoroughly, then the text blocks that Java 15 added for multi-line strings.

`String.format` and `printf`

```
String s = String.format("%s has %d items costing %.2f", name, count, total);
System.out.printf("%s has %d items costing %.2f%n", name, count, total);
String t = "%s has %d items".formatted(name, count);    // Java 15+, same thi
```

All three use the same format string. `printf` prints it; `format` / `formatted` return it. Each `%` specifier consumes the next argument (or an indexed one).

The specifier

```
%[argument_index$][flags][width][.precision]conversion
```

Conversion	Meaning	Example
`%d`	integer (`int` , `long` , `BigInteger`)	`42`
`%s`	string — anything, via `String.valueOf`	`hello`
`%f`	fixed-point decimal, default 6 places	`3.141593`
`%.2f`	two decimal places, rounded HALF_UP	`3.14`
`%e` / `%E`	scientific	`3.141593e+00`
`%g`	general (shortest of `%e` / `%f`)	
`%c`	character	`A`
`%b`	boolean (`null` → `false` , other objects → `true`)	
`%x` / `%X`	hex	`ff` / `FF`
`%o`	octal	
`%n`	platform newline	
`%%`	a literal percent	
`%tF` , `%tT` , `%tY` …	date/time (use `java.time` formatting instead)	

Width, alignment, padding

```
String.format("[%5d]", 42)      // "[   42]"   right-aligned, width 5
String.format("[%-5d]", 42)     // "[42   ]"   left-aligned (flag -)
String.format("[%05d]", 42)     // "[00042]"   zero-padded (flag 0)
String.format("[%+d]", 42)      // "[+42]"     always show sign
String.format("[%,d]", 1234567) // "[1,234,567]"  grouping separator (locale!)
String.format("[%10.3f]", Math.PI)   // "[     3.142]"
String.format("[%-10s]", "left")     // "[left      ]"
String.format("[%10s]", "right")     // "[     right]"
String.format("[%.3s]", "truncate")  // "[tru]"   precision on %s truncates
String.format("%2$s %1$s", "world", "hello")   // "hello world"   explicit in
String.format("%08.2f", 3.14159)     // "00003.14"
```

A table of aligned columns is width specifiers on each field. `%-12s%8d%10.2f%n` is a typical row.

Rounding

`%.2f` rounds with `RoundingMode.HALF_UP` applied to the *shortest decimal representation* of the double — the digits `Double.toString` would print — not to the exact binary value. `String.format("%.2f", 2.675)` therefore gives `2.68` even though the stored double is 2.67499999…; `%.2f` of `1.005` gives `1.01`. (C's `printf` rounds the binary value and prints `2.67`.) For money, format a `BigDecimal` (which `%f` accepts exactly), or round yourself with `setScale` and print the result.

Locale

`%,d` and `%f` are locale-sensitive: on a German machine `%,d` inserts `.` and `%.2f` uses `,` as the decimal point. Tests and file formats want stability: `String.format(Locale.ROOT, "%.2f", x)`. The judge in this course runs with the ROOT/US conventions, so `%.2f` prints a dot.

Errors

A specifier without a matching argument throws `MissingFormatArgumentException`; a mismatch (`%d` with a `String`) throws `IllegalFormatConversionException`. These are run-time errors — the compiler does not check format strings — so test any format you write.

`printf` vs `println` vs string concatenation

`println("x=" + x)` is simplest for debugging. `printf` when you need alignment or decimals. `String.format` when you need the text (a message, a file name). For very hot output paths, `StringBuilder` with manual formatting beats `printf`, which parses the pattern on every call.

`DecimalFormat` and `NumberFormat`

For user-facing numbers with currency symbols and locale rules:

```
NumberFormat.getCurrencyInstance(Locale.US).format(1234.5)      // "$1,234.50"
NumberFormat.getPercentInstance().format(0.256)                 // "26%"
new DecimalFormat("#,##0.00").format(1234.5)                    // "1,234.50"
new DecimalFormat("0.###").format(1.5)                          // "1.5" — up
```

`DecimalFormat` patterns: `0` = required digit, `#` = optional digit, `,` = grouping, `.` = decimal point. `HALF_EVEN` rounding by default — different from `%f`'s `HALF_UP`.

Text blocks (Java 15+)

Multi-line strings used to be concatenation soup. A text block starts with `"""` followed by a newline and ends with `"""`:

```
String json = """
    {
      "name": "Ada",
      "role": "engineer"
    }
    """;
```

Rules that matter:

Indentation is stripped by the amount common to all lines *and the closing delimiter*. Move the closing `"""` left to keep more indentation; put it on the last content line to drop the trailing newline.

Each line ends with `\n` regardless of the source file's line endings.

Trailing spaces on each line are removed (use `\s` to keep one).

`"` inside needs no escaping; `"""` inside does (`\"""`).

A `\` at the very end of a line joins it with the next (no newline) — for long single-line strings.

`\n`, `\t` and other escapes still work; `\s` is a space that survives stripping.

The block is an ordinary `String` — `.formatted(...)`, `.strip()`, `.lines()` all apply.

```
String sql = """
    SELECT id, name
    FROM users
    WHERE role = '%s'
    """.formatted(role);      // and yes, that is an injection risk — use para

String oneLine = """
    a very long string \
    continued here""";        // "a very long string continued here"
```

Text blocks exist for SQL, JSON, HTML and test fixtures. They are a literal, not a template engine — no variables inside; use `formatted` or a builder.

Escapes, recapped

`\n` newline, `\t` tab, `\r` carriage return, `\\` backslash, `\"` quote, `\'` (needed only in `char` literals), `\0` NUL, `\uXXXX` Unicode (processed *before* parsing, anywhere in the source: a Unicode escape for a double quote — `\u` followed by `0022` — ends the string literal early, and one for a line feed — `\u` followed by `000a` — inside a comment starts a new line of code; never write either), `\s` space (Java 15+). There is no `\e` and no raw-string prefix; a regex in a string is `"\\d+"`.

Interview angle

"*`String.format("%5.1f", 3.14159)`*?" `" 3.1"` — width 5 includes the whole thing.

"How do you print with a thousands separator?" `%,d`, or `NumberFormat`.

"Are format strings checked at compile time?" No; mismatches throw at run time.

"What does a text block do with indentation?" Strips the common indentation, measured against the least-indented line including the closing delimiter.

Key takeaways

`%[flags][width][.precision]conversion` — `%d %s %.2f %n` cover most needs; `-` left-aligns, `0` zero-pads, `,` groups.

`%.2f` rounds HALF_UP on the shortest decimal representation (2.675 → `2.68`); use `Locale.ROOT` for stable decimal points.

Format strings are checked at run time — test them.

Text blocks strip common indentation and normalise line endings; `formatted` fills them.

`DecimalFormat` / `NumberFormat` for currency and locale-correct display.

EXERCISE 1

Receipt

Read an integer `n`, then `n` lines each with an item name (no spaces), a quantity and a unit price (`coffee 2 3.50`). Print one line per item formatted as `%-10s%4d%10.2f` — name left-aligned in 10, quantity right-aligned in 4, line total (quantity × price) in 10 with two decimals — followed by a final line `%-14s%10.2f` with the word `TOTAL` and the sum of the line totals.

Example: `2` then `coffee 2 3.50` and `bagel 1 2.25` →

```
coffee       2      7.00
bagel        1      2.25
TOTAL               9.25
```

INPUT 1

```
2
coffee 2 3.50
bagel 1 2.25
```

OUTPUT 1

```
coffee       2      7.00
bagel        1      2.25
TOTAL               9.25
```

INPUT 2

```
1
widget 100 0.10
```

OUTPUT 2

```
widget     100     10.00
TOTAL              10.00
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        StringBuilder out = new StringBuilder();
        double total = 0;
        for (int i = 0; i < n; i++) {
            String[] p = br.readLine().trim().split("\\s+");
            String name = p[0];
            int qty = Integer.parseInt(p[1]);
            double price = Double.parseDouble(p[2]);
            double line = qty * price;
            total += line;
            out.append(String.format("%-10s%4d%10.2f%n", name, qty, line));
        }
        out.append(String.format("%-14s%10.2f%n", "TOTAL", total));
        System.out.print(out);
    }
}
```

EXERCISE 2

Aligned ledger

Read an integer `n` and `n` lines `<label> <amount>` (amount an integer, possibly negative). Print each as `%-12s%,12d` — label left-aligned in 12, amount right-aligned in 12 with thousands separators — then a rule of 24 dashes and a `TOTAL` row in the same format. Use `Locale.ROOT` so the separator is a comma.

Example: `2` then `rent -12500`, `salary 1250000` →

```
rent             -12,500
salary         1,250,000
------------------------
TOTAL          1,237,500
```

INPUT 1

```
2
rent -12500
salary 1250000
```

OUTPUT 1

```
rent             -12,500
salary         1,250,000
------------------------
TOTAL          1,237,500
```

INPUT 2

```
1
coffee 350
```

OUTPUT 2

```
coffee               350
------------------------
TOTAL                350
```

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();
        long total = 0;
        for (int i = 0; i < n; i++) {
            String label = in.next();
            long amount = in.nextLong();
            total += amount;
            System.out.println(String.format(Locale.ROOT, "%-12s%,12d", label, amour
        }
        System.out.println("-".repeat(24));
        System.out.println(String.format(Locale.ROOT, "%-12s%,12d", "TOTAL", total))
    }
}
```

QUIZ

1. `String.format("[%-6s]", "ab")` gives…

A. `[ ab]`

B. `[ab ]`

C. `[ab]`

D. `[000ab]`

2. `String.format("%,d", 1234567)` in the ROOT locale gives…

A. `1234567`

B. `1,234,567`

C. `1.234.567`

D. An exception

3. Are format strings checked by the compiler?

A. Yes

B. No — a mismatch such as `%d` with a `String` throws at run time

C. Only in `printf`

D. Only with `-Xlint`

4. In a text block, what determines how much indentation is removed?

A. The first line only

B. The least-indented content line **or** the closing `"""`, whichever is further left

C. Nothing is removed

D. Four spaces always

5. `String.format("%.2f", 2.675)` prints `2.68`, although the stored double is 2.67499999.... Why?

A. `%f` always rounds up

B. Java's formatter rounds the shortest decimal representation (`2.675`) HALF_UP, not the exact binary value

C. The JIT corrects the value

D. A JDK bug

ANSWER KEY · FORMATTING OUTPUT AND TEXT BLOCKS

1. B Width 6 with the `-` flag left-aligns and pads on the right. Without the flag it would be right-aligned.

2. B The `,` flag inserts grouping separators appropriate to the locale — commas for ROOT/US, dots for German.

3. B `String.format` parses the pattern at run time; `IllegalFormatConversionException` and `MissingFormatArgumentException` are the results of a mismatch.

4. B The common leading whitespace of all lines, including the position of the closing delimiter, is stripped. Moving the closing `"""` left keeps more indentation.

5. B Unlike C's printf, Java's `%f` works from the digits `Double.toString` would produce and rounds those HALF_UP. `Math.round(2.675 * 100)` sees the binary value and gives 267.

LESSON 6 · 14 MIN

Characters, Unicode and encodings

A Java `char` is not a character. It is a 16-bit UTF-16 *code unit*, and a `String` is a sequence of those units. For English text the distinction never shows; for an emoji, a Chinese name, or a file read with the wrong encoding, it decides whether your code is correct. This lesson gives you the model: code points versus code units, the `Character` utilities, and how bytes become strings.

Code points and code units

Unicode assigns every character a number, its **code point**, from U+0000 to U+10FFFF — about 1.1 million possible, 150 000 assigned. UTF-16 stores code points up to U+FFFF (the *Basic Multilingual Plane*) in one 16-bit unit and the rest in **two** units called a *surrogate pair*.

```
String s = "A€😜";
s.length()              // 4 — 'A' (1 unit), '€' (1 unit, U+20AC), '😜' (2 un
s.codePointCount(0, s.length())   // 3 — the number of characters a human woul
s.charAt(2)             // '\uD83D' — the HIGH surrogate, half an emoji
s.codePointAt(2)        // 128512 (0×1F600) — the whole emoji
s.substring(2, 3)       // a broken string containing a lone surrogate
new StringBuilder(s).reverse()    // "😜€A" — reverse() keeps the pair togeth
```

Rules:

- `length()`, `charAt`, `substring`, `indexOf` work in **code units**. They are correct for BMP text and *can split a surrogate pair* for text beyond it.
- `codePoints()` (an `IntStream`), `codePointAt`, `codePointCount`, `offsetByCodePoints` work in **code points**.
- Emoji, many mathematical symbols, historic scripts and some CJK characters live outside the BMP. If your input might contain them, iterate by code point.

```
for (int i = 0; i < s.length(); ) {
    int cp = s.codePointAt(i);
    // … handle cp …
    i += Character.charCount(cp);      // 1 or 2
}
s.codePoints().forEach(cp → …);        // the stream form
```

Even a code point is not always a "character": `é` can be one code point (U+00E9) or two (`e` + combining acute U+0301). `java.text.Normalizer.normalize(s, Normalizer.Form.NFC)` composes such sequences; do this before comparing user-entered text.

The `Character` class

Static utilities for classification and conversion — the `char` versions and the `int` (code point) versions:

```
Character.isDigit('7')          // true — also Arabic-Indic digits etc.; for A
Character.isLetter('é')         // true
Character.isLetterOrDigit(c)
Character.isWhitespace(' ')     // true for space, tab, newline…
Character.isUpperCase('A'), Character.isLowerCase('a')
Character.toUpperCase('a')      // 'A'
Character.toLowerCase('A')
Character.getNumericValue('7')  // 7; also works for 'a' → 10 (hex-style), and
Character.digit('f', 16)        // 15; -1 if not a digit in that radix
Character.forDigit(11, 16)      // 'b'
Character.isSurrogate(c), Character.isHighSurrogate(c), Character.isLowSurroga
Character.charCount(cp)         // 1 or 2
Character.toChars(cp)           // char[] for a code point
Character.MAX_VALUE             // '\uFFFF'
```

`Character.isDigit` is Unicode-aware, which is usually right for users and wrong for parsers: `Integer.parseInt` accepts only what `Character.digit` accepts, which includes non-ASCII digits. For strict ASCII checks compare ranges.

The `c - '0'` trick from Module 2 is safe only for ASCII digits; `Character.getNumericValue` is the general form.

Encodings: bytes ↔ strings

A `String` is characters; a file, a socket, or an HTTP body is **bytes**. An *encoding* (charset) maps between them:

Charset	Bytes per char	Notes
UTF-8	1–4	ASCII-compatible; the web and Linux default; **use this**
UTF-16	2 or 4	Java's in-memory form; BOM issues in files
ISO-8859-1 (Latin-1)	1	Western Europe; every byte is a valid char
US-ASCII	1	7-bit; anything else becomes `?`
Windows-1252	1	The Windows Western default; superset of Latin-1

```
byte[] bytes = "héllo".getBytes(StandardCharsets.UTF_8);   // 6 bytes: é is 2
String back = new String(bytes, StandardCharsets.UTF_8);   // "héllo"
String wrong = new String(bytes, StandardCharsets.ISO_8859_1);   // "hÃ©llo" -
```

Always name the charset. `getBytes()` and `new String(bytes)` with no argument use the *platform default*, which was Windows-1252 on Windows and UTF-8 on Linux — the classic "works on my laptop, garbage on the server". Since **Java 18** (JEP 400) the default is UTF-8 everywhere, but the no-argument forms remain a smell because code is compiled for many versions.

The same applies to readers and writers: `new InputStreamReader(in, StandardCharsets.UTF_8)`, `Files.readString(path)` (UTF-8 by default), `Files.newBufferedReader(path, charset)`.

`String.length()` is **not** the byte length. A 10-character string can be 10, 20 or 40 bytes. Database column limits, HTTP `Content-Length` and buffer sizes are in bytes; compute them from `getBytes(charset).length`.

Escapes for characters

```
char tab = '\t';
char quote = '\'';
char unicode = '\u00E9';        // é
char emojiHigh = '\uD83D';      // half of 😜 — a char cannot hold the whole
String emoji = "\uD83D\uDE00";  // both halves; or just paste 😜 into a UTF-8
int cp = 0x1F600; String e = new String(Character.toChars(cp));
String e2 = Character.toString(cp);   // Java 11+
```

`\uXXXX` escapes are processed by the compiler *before* tokenising, anywhere in the source — including comments. `// \u000a int x;` puts `int x;` on a new line after the comment, outside it. A curiosity, but a real one.

`char` arithmetic, revisited

`char` is an unsigned 16-bit number: `'a' + 1` is the `int` 98; `(char) ('a' + 1)` is `'b'`; `'z' - 'a'` is 25; `(char) (c ^ 32)` flips ASCII case. Comparisons `c ⩾ 'a' && c ⩽ 'z'` are numeric. A `char` in a `switch` is fine. A `char` added to a `String` concatenates; a `char` added to an `int` is arithmetic — `"" + 'a' + 1` is `"a1"`, `'a' + 1 + ""` is `"98"`.

Sorting and comparing non-ASCII text

`compareTo` orders by code unit, so `"Zebra" < "apple"` and `"é" > "z"`. For human ordering use `java.text.Collator`:

```
Collator c = Collator.getInstance(Locale.FRENCH);
list.sort(c);       // é sorts with e, case handled sensibly
```

`equalsIgnoreCase` and `toLowerCase(Locale.ROOT)` are fine for identifiers and keywords; for names, normalise (NFC) first and use a collator.

Interview angle

"What is the length of a string containing one emoji?" 2 — a surrogate pair. `codePointCount` gives 1.

"Difference between `char` and code point?" A `char` is a UTF-16 code unit; a code point is the Unicode character number and may need two units.

"Why must you specify a charset?" The default varied by platform before Java 18; bytes decoded with the wrong charset are corrupted.

"How many bytes is `"é"` in UTF-8?" 2. In UTF-16, 2. In Latin-1, 1.

Key takeaways

`char` = UTF-16 code unit; characters beyond U+FFFF take two. Use `codePoints()` when input may contain them.

`Character.isDigit` / `isLetter` are Unicode-aware; use range checks for strict ASCII.

Always pass a `Charset` (`StandardCharsets.UTF_8`) when converting bytes ↔ text.

`length()` counts units, not bytes or user-perceived characters.

`Collator` for human sorting; `Normalizer` before comparing accented input.

EXERCISE 1

Count what is really there

Read one line and print four counts on one line: `units=<length()> codepoints=<codePointCount> letters=<count of code points for which Character.isLetter is true> digits=<count of code points for which Character.isDigit is true>`.

Iterate by **code point**, not by `char`, so that a character outside the Basic Multilingual Plane (an emoji, for instance) is counted once.

Input: one line (UTF-8). **Output:** one line.

Example: `abc 123` → `units=7 codepoints=7 letters=3 digits=3`

INPUT 1

```
abc 123
```

OUTPUT 1

```
units=7 codepoints=7 letters=3 digits=3
```

INPUT 2

```
café é
```

OUTPUT 2

```
units=6 codepoints=6 letters=5 digits=0
```

INPUT 3

```
hi 😜 5
```

OUTPUT 3

```
units=7 codepoints=6 letters=2 digits=1
```

REFERENCE SOLUTION

```java
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s = br.readLine();
        int letters = 0, digits = 0;
        for (int i = 0; i < s.length(); ) {
            int cp = s.codePointAt(i);
            if (Character.isLetter(cp)) letters++;
            if (Character.isDigit(cp)) digits++;
            i += Character.charCount(cp);
        }
        System.out.println("units=" + s.length() + " codepoints=" + s.codePointCount
            + " letters=" + letters + " digits=" + digits);
    }
}
```

EXERCISE 2

Character census

Read one line. Using the `Character` methods, count upper-case letters, lower-case letters, digits, whitespace and everything else, and print `upper=<n> lower=<n> digits=<n> spaces=<n> other=<n>`. Then print `chars=<s.length()> codePoints=<s.codePointCount(0, s.length())>` — they differ when a character needs two `char`s.

Example: `Hello World 42!` →

```
upper=2 lower=8 digits=2 spaces=2 other=1
chars=15 codePoints=15
```

INPUT 1

```
Hello World 42!
```

OUTPUT 1

```
upper=2 lower=8 digits=2 spaces=2
other=1
chars=15 codePoints=15
```

INPUT 2

```
héllo Wörld
```

OUTPUT 2

```
upper=1 lower=9 digits=0 spaces=1
other=0
chars=11 codePoints=11
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s = in.hasNextLine() ? in.nextLine() : "";
        int upper = 0, lower = 0, digits = 0, spaces = 0, other = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isUpperCase(c)) upper++;
            else if (Character.isLowerCase(c)) lower++;
            else if (Character.isDigit(c)) digits++;
            else if (Character.isWhitespace(c)) spaces++;
            else other++;
        }
        System.out.println("upper=" + upper + " lower=" + lower + " digits=" + digit
        System.out.println("chars=" + s.length() + " codePoints=" + s.codePointCount
    }
}
```

QUIZ

1. `"😜".length()` is…

A. 1

B. 2 — the emoji is a surrogate pair of two UTF-16 units

C. 4

D. Depends on the platform

2. Which call is the correct way to turn bytes into text?

A. `new String(bytes)`

B. `new String(bytes, StandardCharsets.UTF_8)`

C. `bytes.toString()`

D. `String.valueOf(bytes)`

3. How many bytes is `"é"` in UTF-8?

A. 1

B. 2

C. 3

D. 4

4. `Character.isDigit(c)` is true for…

A. Only `'0'`–`'9'`

B. Any Unicode decimal digit, including non-ASCII ones

C. Digits and letters

D. Only if `c` is an `int`

5. `new StringBuilder("a😜").reverse()` gives…

A. A string with the emoji broken into two halves

B. `"😜a"` — `reverse` keeps surrogate pairs intact

C. `"a😜"` unchanged

D. An exception

ANSWER KEY · CHARACTERS, UNICODE AND ENCODINGS

1. B U+1F600 lies outside the Basic Multilingual Plane and takes two `char`s. `codePointCount` reports 1.

2. B Always name the charset. `bytes.toString()` gives `[B@hash`; the no-argument constructor used the platform default before Java 18.

3. B U+00E9 encodes as two bytes (C3 A9) in UTF-8. ASCII is one byte; CJK characters are three; emoji are four.

4. B The `Character` methods are Unicode-aware. For a strict ASCII check compare ranges: `c ⩾ '0' && c ⩽ '9'`.

5. B `StringBuilder.reverse` is specified to treat surrogate pairs as single characters. A hand-written `char` swap would corrupt them.

LESSON 7 · 16 MIN

Regular expressions in Java

Regular expressions describe patterns in text: "one or more digits", "an email-shaped thing", "a word at the start of a line". Java's `java.util.regex` is a full-featured engine hiding behind a few `String` methods and two classes, `Pattern` and `Matcher`. This lesson covers the syntax you need, the API around it, and the three mistakes — unescaped metacharacters, `matches` versus `find`, and recompiling in loops — that account for most regex bugs.

Where regexes appear

```
s.matches("\\d+")                // whole string is one or more digits?
s.replaceAll("\\s+", " ")        // collapse whitespace
s.replaceFirst("^0+", "")        // strip leading zeros
s.split("[,;]\\s*")              // split on comma or semicolon plus optional s
Pattern.compile("…").matcher(s) // the full API
```

The pattern is written inside a Java string, so every regex backslash is doubled: the regex `\d+` is the Java literal `"\\d+"`.

Syntax essentials

Pattern	Matches
`.`	any character except newline
`\d` `\w` `\s`	digit, word char `[A-Za-z0-9_]`, whitespace
`\D` `\W` `\S`	the complements
`[abc]` `[a-z]` `[^0-9]`	character class, range, negated class
`^` `$`	start / end of input (or line, with `MULTILINE`)
`\b`	word boundary
`x*` `x+` `x?`	zero or more, one or more, zero or one
`x{3}` `x{2,5}` `x{2,}`	exact, range, at-least
`x*?` `x+?`	**lazy** versions — as few as possible
`a|b`	alternation
`(…)`	capturing group
`(?:…)`	non-capturing group
`(?<name>…)`	named group
`(?i)`	case-insensitive flag inline
`\.` `\(` `\\`	escaped literals

Metacharacters that must be escaped when meant literally: `. ^ $ * + ? ( ) [ ] { } | \`. Inside a character class only `\`, `]`, `^` (first) and `-` (not at an edge) are special.

`Pattern` and `Matcher`

```java
import java.util.regex.*;

Pattern p = Pattern.compile("(\\w+)@(\\w+)\\.com");
Matcher m = p.matcher("mail ada@example.com or bob@test.com");

while (m.find()) {                          // next match anywhere in the input
    m.group()                               // "ada@example.com" — the whole mat
    m.group(1)                              // "ada"
    m.group(2)                              // "example"
    m.start(), m.end()                      // indices of the match
}

m.reset();
m.matches()      // does the ENTIRE input match? false here
m.lookingAt()    // does the input START with a match? false here
```

The three verbs:

- `matches()` — the whole input must match. `"abc123".matches("\\d+")` is false.
- `find()` — search for the next occurrence anywhere. Call repeatedly.
- `lookingAt()` — anchored at the start, not the end.

`String.matches` is the `matches()` verb; there is no `String.find` — use a `Matcher`.

Groups are numbered by their opening parenthesis, left to right, starting at 1; `group(0)` is the whole match. Named groups: `m.group("user")`. A group that did not participate returns `null`.

Replacing with groups

```java
"2026-09-16".replaceAll("(\\d{4})-(\\d{2})-(\\d{2})", "$3/$2/$1")   // "16/09/
"price: 10".replaceAll("\\d+", "<$0>")                              // "price:
```

In the replacement string `$1`, `$0` and `${name}` refer to groups, and `\` and `$` must be escaped (`Matcher.quoteReplacement(s)` does it) — a replacement containing a literal `$` is a common source of `IllegalArgumentException: Illegal group reference`.

For computed replacements:

```
Matcher m = Pattern.compile("\\d+").matcher(text);
String doubled = m.replaceAll(r → String.valueOf(Integer.parseInt(r.group())
```

Greedy versus lazy

```
"<b>bold</b> and <i>it</i>".replaceAll("<.*>", "")      // "" — greedy .* ate e
"<b>bold</b> and <i>it</i>".replaceAll("<.*?>", "")     // "bold and it" — lazy
"<b>bold</b>".replaceAll("<[^>]*>", "")                 // same, and faster: a
```

Quantifiers are greedy by default; `?` after one makes it lazy. A negated character class (`[^>]*`) is often clearer and never backtracks past the delimiter.

Performance and safety

Compile once. `Pattern.compile` parses the regex; `String.matches` / `replaceAll` / `split` compile on every call. In a loop, hoist `static final Pattern P = Pattern.compile(...)`. `Pattern` is immutable and thread-safe; `Matcher` is not — create one per use.

Catastrophic backtracking. Nested quantifiers like `(a+)+$` on a non-matching input take exponential time. Prefer possessive quantifiers (`a++`), atomic groups, or a simpler pattern. Never run a user-supplied regex on user-supplied input without a timeout.

`Pattern.quote(literal)` escapes a string for use inside a pattern; `Matcher.quoteReplacement` for the replacement side.

Flags

```
Pattern.compile("^error", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE)
```

`CASE_INSENSITIVE` (inline `(?i)`), `MULTILINE` — `^` / `$` match at line boundaries (`(?m)`), `DOTALL` — `.` matches newlines too (`(?s)`), `UNICODE_CASE`, `COMMENTS` — whitespace and `#` comments allowed in the pattern (`(?x)`).

Patterns worth memorising

```
"\\d+"                          // integer
"-?\\d+(\\.\\d+)?"              // signed decimal
"[A-Za-z_][A-Za-z0-9_]*"        // Java-ish identifier
"\\s+"                          // whitespace run (for split)
"[^\\p{L}\\p{N}]+"              // anything not a letter or number (Unicode)
"(?i)^(yes|y|true)$"            // affirmative answer
"^[\\w.+-]+@[\\w-]+\\.[\\w.-]+$"   // pragmatic email shape (not RFC-complete
"\\b\\w+\\b"                    // words
```

`\p{L}` (letter), `\p{N}` (number), `\p{Lu}` (uppercase letter) are Unicode categories and are preferable to `[A-Za-z]` when input may be non-English.

When not to use a regex

A regex is a tool for *patterns*. For "does it contain this literal", `contains` / `indexOf` is faster and clearer. For parsing nested structures (JSON, HTML, expressions) a regex is the wrong tool — the language is not regular. For simple splitting on one character, `split(",")` or `indexOf` loops. If a regex is longer than the line it lives on, break it into named pieces or write a small parser.

Interview angle

- *"`"a.b".split(".")` returns?"* An empty array — `.` matches every character, all pieces are empty and trailing empties are dropped.
- *"`matches` vs `find`?"* Whole input vs anywhere.
- *"Why compile the pattern outside the loop?"* Compilation is the expensive part; `Pattern` is thread-safe and reusable.
- *"What is greedy matching?"* Quantifiers take as much as possible and backtrack; `?` after a quantifier makes it lazy.

Key takeaways

Double every backslash in the Java string; escape `. ( ) [ ] { } | + * ? ^ $` when literal.

`matches` needs the whole string; `find` searches; groups are 1-based, `$1` in replacements.

Greedy by default; lazy with `?`; negated classes beat `.*?` for delimiters.

Hoist `Pattern.compile` out of loops; `Matcher` is per-use.

`split`, `replaceAll`, `matches` take regexes; `replace` does not.

EXERCISE 1

Numbers in the noise

Read one line and find every integer in it (an optional minus sign followed by one or more digits) using a compiled `Pattern` and `Matcher.find()`. Print the numbers space-separated on the first line (or `none` if there are none), and `sum=<total>` on the second, using a `long` for the total.

Example: `Order 66 shipped -3 items, ref 2024` →

```
66 -3 2024
sum=2087
```

INPUT 1

```
Order 66 shipped -3 items, ref 2024
```

OUTPUT 1

```
66 -3 2024
sum=2087
```

INPUT 2

```
no digits at all
```

OUTPUT 2

```
none
sum=0
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String line = br.readLine();
        Pattern p = Pattern.compile("-?\\d+");
        Matcher m = p.matcher(line);
        StringBuilder found = new StringBuilder();
        long sum = 0;
        while (m.find()) {
            if (found.length() > 0) found.append(' ');
            found.append(m.group());
            sum += Long.parseLong(m.group());
        }
        System.out.println(found.length() == 0 ? "none" : found.toString());
        System.out.println("sum=" + sum);
    }
}
```

EXERCISE 2

Dates and masks

Read lines until end of input. For every date of the form `yyyy-mm-dd` found in a line (pattern `(\d{4})-(\d{2})-(\d{2})` with three groups) print `date: y=<yyyy> m=<mm> d=<dd>`. Then print the line with every run of digits replaced by a single `#` (`replaceAll("\\d+", "#")`), prefixed `masked: `.

Example input

```
Paid 2024-03-01 and 2024-04-15, invoice 88
```

→

```
date: y=2024 m=03 d=01
date: y=2024 m=04 d=15
masked: Paid #-#-# and #-#-#, invoice #
```

INPUT 1

```
Paid 2024-03-01 and 2024-04-15, invoice
88
```

OUTPUT 1

```
date: y=2024 m=03 d=01
date: y=2024 m=04 d=15
masked: Paid #-#-# and #-#-#, invoice #
```

INPUT 2

```
no dates here
```

OUTPUT 2

```
masked: no dates here
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        Pattern date = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) ≠ null) {
            Matcher m = date.matcher(line);
            while (m.find()) {
                System.out.println("date: y=" + m.group(1) + " m=" + m.group(2) + "
            }
            System.out.println("masked: " + line.replaceAll("\\d+", "#"));
        }
    }
}
```

QUIZ

1. `"abc123".matches("\\d+")` is…

A. `true`

B. `false` — `matches` requires the whole string to match

C. `true` because it contains digits

D. A compile error

2. Which pattern removes HTML tags from `<b>bold</b> text` without swallowing the text between tags?

A. `<.*>`

B. `<.*?>` or `<[^>]*>`

C. `<.+>`

D. `<>`

3. Why hoist `Pattern.compile(...)` out of a loop?

A. `Pattern` objects are single-use

B. Compilation is the expensive part; a `Pattern` is immutable and reusable

C. The loop would not compile otherwise

D. It changes the match results

4. In `replaceAll`, what does `$1` in the replacement string mean?

A. A literal dollar and 1

B. The text captured by the first group

C. The first match

D. An error

5. `"a|b|c".split("|")` gives…

A. `["a", "b", "c"]`

B. `["a", "|", "b", "|", "c"]` — `|` is alternation with an empty pattern, so it splits between every character

C. `["a|b|c"]`

D. An exception

ANSWER KEY · REGULAR EXPRESSIONS IN JAVA

1. B `matches` is anchored at both ends. To search for digits anywhere, use a `Matcher` and `find()`.

2. B Greedy `.*` runs to the last `>`. Lazy `.*?` or a negated class `[^>]*` stops at the first one.

3. B `String.matches` / `split` / `replaceAll` recompile every call. A `static final Pattern` is parsed once; `Matcher` is created per input.

4. B `$n` refers to capturing group n; `$0` is the whole match. A literal `$` must be escaped (`Matcher.quoteReplacement`).

5. B `|` is a regex metacharacter meaning "empty or empty", which matches at every position. Escape it: `split("\\|")`.

CHECKPOINT · 25 MIN

Checkpoint — Strings

This checkpoint covers immutability and the string pool, comparison, the core API, `StringBuilder`, formatting and text blocks, characters and encodings, and regular expressions.

How it works. Fifteen questions and two programs; 70% on the questions and both programs accepted clears the module. Retake as often as you like — the best score counts.

Before you start, check these are automatic:

- What `s.toUpperCase();` on its own line does (nothing you can see).
- When `==` on two strings is true, and why you still use `equals`.
- The result of `"a,b,,".split(",")` and of `"a.b".split(".")`.
- `substring(begin, end)` — which end is exclusive.
- Why `+=` in a loop is slow and what to use instead.
- `%5d`, `%-5s`, `%05d`, `%.2f`, `%,d`.
- The length of a string holding one emoji.
- `replace` versus `replaceAll`; `matches` versus `find`.

The programs are string manipulation classics — one is about building output efficiently and correctly, the other about parsing structured text with the right tool.

EXERCISE 1

Caesar shift

Read an integer `k` (which may be negative or larger than 26) on the first line, then a line of text. Print the text with every letter shifted `k` places forward in the alphabet, wrapping around, preserving case; leave non-letters unchanged.

Use `char` arithmetic and `Math.floorMod` so negative shifts work. Build the output with a `StringBuilder`.

Example: `3` then `Hello, World!` → `Khoor, Zruog!`

INPUT 1

```
3
Hello, World!
```

OUTPUT 1

```
Khoor, Zruog!
```

INPUT 2

```
-3
Khoor, Zruog!
```

OUTPUT 2

```
Hello, World!
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int k = Integer.parseInt(br.readLine().trim());
        String text = br.readLine();
        int shift = Math.floorMod(k, 26);
        StringBuilder sb = new StringBuilder(text.length());
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            if (c >= 'a' && c <= 'z') {
                sb.append((char) ('a' + (c - 'a' + shift) % 26));
            } else if (c >= 'A' && c <= 'Z') {
                sb.append((char) ('A' + (c - 'A' + shift) % 26));
            } else {
                sb.append(c);
            }
        }
        System.out.println(sb);
    }
}
```

EXERCISE 2

Title case and acronym

Read one line containing words separated by any amount of whitespace. Print two lines:

> The words in **Title Case** (first letter uppercase, the rest lowercase), joined by single spaces.
>
> `acronym=<X>` where `X` is the uppercase first letter of every word.

Split with a regular expression, trim first, and use `Character.toUpperCase` / `toLowerCase`.

Example: `the QUICK brown fox` →

```
The Quick Brown Fox
acronym=TQBF
```

INPUT 1

```
  the QUICK   brown fox
```

OUTPUT 1

```
The Quick Brown Fox
acronym=TQBF
```

INPUT 2

```
java
```

OUTPUT 2

```
Java
acronym=J
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String line = br.readLine().trim();
        String[] words = line.isEmpty() ? new String[0] : line.split("\\s+");
        StringBuilder title = new StringBuilder();
        StringBuilder acronym = new StringBuilder();
        for (int i = 0; i < words.length; i++) {
            String w = words[i];
            char first = Character.toUpperCase(w.charAt(0));
            if (i > 0) title.append(' ');
            title.append(first).append(w.substring(1).toLowerCase());
            acronym.append(first);
        }
        System.out.println(title);
        System.out.println("acronym=" + acronym);
    }
}
```

EXERCISE 3

Word wrap

Read an integer width `w` on the first line, then text on the remaining lines. Re-flow the words (split on any whitespace) greedily into lines of at most `w` characters, one space between words; a word longer than `w` goes on a line of its own. Print the lines, then `lines=<count>`.

Example: `12` then `the quick brown fox jumps over the lazy dog` →

```
the quick
brown fox
jumps over
the lazy dog
lines=4
```

INPUT 1

```
12
the quick brown fox jumps over the lazy
dog
```

OUTPUT 1

```
the quick
brown fox
jumps over
the lazy dog
lines=4
```

INPUT 2

```
5
abcdefgh ij
```

OUTPUT 2

```
abcdefgh
ij
lines=2
```

REFERENCE SOLUTION

```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));
        int w = Integer.parseInt(in.readLine().trim());
        StringBuilder text = new StringBuilder();
        String line;
        while ((line = in.readLine()) != null) text.append(line).append(' ');
        String[] words = text.toString().trim().split("\\s+");
        StringBuilder current = new StringBuilder();
        int lines = 0;
        for (String word : words) {
            if (word.isEmpty()) continue;
            if (current.length() > 0 && current.length() + 1 + word.length() > w) {
                System.out.println(current);
                lines++;
                current.setLength(0);
            }
            if (current.length() > 0) current.append(' ');
            current.append(word);
        }
        if (current.length() > 0) {
            System.out.println(current);
            lines++;
        }
        System.out.println("lines=" + lines);
    }
}
```

CHECKPOINT QUESTIONS

1. Which statement about `String` is false?

A. It is immutable

B. It is `final`

C. Its `hashCode` is cached after the first call

D. Its `length()` returns the number of Unicode characters

2. `String a = "ja" + "va"; a == "java"` is…

A. `true` — the compiler folds the constant and pools it

B. `false`

C. `true` only with `intern()`

D. Undefined

3. How should two possibly-null strings be compared for equality?

A. `a == b`

B. `a.equals(b)`

C. `Objects.equals(a, b)`

D. `a.compareTo(b) == 0`

4. `"10".compareTo("9")` is…

A. Positive

B. Negative — `'1'` is less than `'9'`

C. 0

D. An exception

5. `"banana".indexOf("an", 2)` returns…

A. 1

B. 3

C. -1

D. 2

6. `"a.b".split(".")` returns…

A. `["a", "b"]`

B. An empty array

C. `["a.b"]`

D. `["", "", ""]`

7. Which builds a large string in a loop efficiently?

A. `s += piece`

B. `s = s.concat(piece)`

C. `sb.append(piece)` on a `StringBuilder`, then `toString()`

D. `String.format("%s%s", s, piece)`

8. `new StringBuilder("abc").reverse().toString()` gives…

A. `"cba"`

B. `"abc"`

C. `"StringBuilder@…"`

D. A compile error

9. `String.format("%05.1f", 3.14159)` gives…

A. `003.1`

B. `3.1`

C. `3.14`

D. `00003.1`

10. What does a text block do with the line terminators in the source file?

A. Keeps them as written (`\r\n` on Windows)

B. Normalises every line ending to `\n`

C. Removes them

D. Converts them to spaces

11. Reading a UTF-8 file with `new String(bytes, StandardCharsets.ISO_8859_1)` produces…

A. The correct text

B. Mojibake — multi-byte characters become two or more wrong characters

C. An exception

D. The correct text for ASCII only, with an exception otherwise

12. `"Hello World".replaceAll("o", "0")` and `.replace("o", "0")` differ in…

A. Nothing here — both replace every `o`; `replaceAll` treats its pattern as a regex

B. `replace` changes only the first

C. `replaceAll` changes only the first

D. `replace` is deprecated

13. In a regex, what does `\b` match?

A. A backspace

B. A word boundary — the position between a word character and a non-word character

C. A blank line

D. A literal `b`

14. `Character.getNumericValue('7')` returns…

A. `'7'`

B. 7

C. 55

D. -1

15. Why is `"x".equals(sb)` (where `sb` is a `StringBuilder` holding `x`) false?

A. `StringBuilder` contents are private

B. `String.equals` returns false for any non-`String` argument

C. It is true

D. `sb` is null

ANSWER KEY · CHECKPOINT — STRINGS

1. D `length()` counts UTF-16 code units; a supplementary character counts as two. The other three statements are true.

2. A Concatenation of compile-time constants is folded into one literal, which is interned. With a non-final variable operand it would be a run-time object and `false`.

3. C `Objects.equals` handles null on either side. `a.equals(b)` throws when `a` is null; `==` compares references.

4. B Strings compare character by character. For numeric ordering parse first or use a comparator on the parsed value.

5. B The search starts at index 2 (`"nana"`), and the next `"an"` begins at index 3.

6. B `.` matches every character, so every piece is empty and trailing empties are dropped — nothing is left. Escape it: `"\\."`.

7. C Only the builder appends in place. The other three create a new full-length string on every iteration.

8. A `reverse` mutates the builder in place and returns it; `toString` produces the string. This is the idiomatic reversal.

9. A Width 5 total, one decimal, zero-padded: `3.1` is three characters, padded to `003.1`.

10. B Text blocks always produce `\n` line endings regardless of how the file was saved, so the string is the same on every platform.

11. B Latin-1 maps every byte to a character, so the two bytes of `é` become `Ã©`. Wrong charset never throws; it silently corrupts.

12. A Both replace all occurrences. The difference is that `replaceAll`'s first argument is a regular expression, which matters for characters like `.`.

13. B `\bcat\b` matches `cat` but not `concatenate`. Inside a character class `[\b]` it is a backspace.

14. B It returns the digit's value as an `int`. `'7' - '0'` gives the same for ASCII digits; `(int) '7'` would be the code 55.

15. B `String.equals` checks `instanceof String` first. Use `"x".contentEquals(sb)` to compare with any `CharSequence`.
