Module 4 of 20

Control flow

if/else and guard clauses, every form of switch, the four loops, break/continue/labels, and the loop patterns programs are made of.

Control flow is where a program stops being a list of statements and starts making decisions and repeating work. Java's constructs are C's, tightened: conditions must be booleans, switch grew an arrow form and became an expression, and labels replace goto.

This module covers each construct precisely — the dangling else, fall-through, exhaustiveness, loop variable scope, what break does inside a switch — and then the conventions that make control flow readable: guard clauses, early return, and the dozen loop patterns (accumulate, count, find, all/any, best-so-far, two pointers, sliding window) that most code is assembled from.

By the end you write loops without off-by-one errors, choose switch expressions where they fit, and exit nested loops cleanly.

CONTENTS

1	if, else and the shape of a decision	12 min
2	switch — statements, expressions and arrows	15 min
3	Loops — while, do-while, for and for-each	14 min
4	break, continue, labels and return	11 min
5	Loop patterns every program is made of	14 min
✓	Checkpoint — Control flow	22 min

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

LESSON 1 · 12 MIN

if, else and the shape of a decision

`if` is the first control-flow statement everyone learns and the one whose *style* most affects whether code can be read. This lesson covers the semantics precisely — including the dangling `else` and the boolean rules — and then the conventions that turn a nest of conditions into something a reviewer can follow: guard clauses, early returns, and expressing conditions as boolean values rather than branches.

The statement

```
if (condition) statement
if (condition) statement else statement
```

`condition` must be a `boolean` expression — not an `int`, not an object, not a `String`. `if (count)` is a compile error; write `if (count ≠ 0)`. `if (name)` is an error; `if (name ≠ null)` or `if (!name.isEmpty())`. This is deliberate: C's "any non-zero value is true" hides bugs like `if (x = 5)`, which in Java does not compile because `x = 5` is an `int`. (`if (flag = true)` *does* compile, since the assignment yields a `boolean`, and is a famous bug; write `if (flag)`.)

The body is a single statement, which is why braces exist:

```
if (x > 0)
    System.out.println("positive");
    total += x;             // NOT inside the if — indentation lies; the compi
```

Always use braces, even for one line. Apple's "goto fail" SSL bug was exactly this shape. The only accepted exception in some codebases is a one-line guard `if (x == null) return;` on a single line.

`else if` chains

```
if (score ⩾ 90) {
    grade = 'A';
} else if (score ⩾ 80) {
    grade = 'B';
} else if (score ⩾ 70) {
    grade = 'C';
} else {
    grade = 'F';
}
```

There is no `elif` keyword; `else if` is an `else` whose statement happens to be another `if`. The conditions are tested top to bottom and the **first true one wins**, so order matters: testing `score ⩾ 70` first would never let `A` or `B` happen. When the ranges are exhaustive, end with a plain `else`; when a branch should be impossible, throw (`throw new IllegalStateException("unreachable: " + score)`) rather than silently doing nothing.

The dangling `else`

```
if (a)
    if (b)
        x();
else                // which if does this belong to?
    y();
```

Java binds an `else` to the **nearest** unmatched `if` — so this `else` belongs to `if (b)`, whatever the indentation says. Braces remove the ambiguity, which is the third reason to always use them.

Conditions are values

A condition is just a `boolean` expression, and boolean expressions can be named, returned and stored:

```java
// verbose
if (age ⩾ 18) {
    return true;
} else {
    return false;
}
// direct
return age ⩾ 18;

// verbose
boolean valid;
if (name ≠ null && !name.isEmpty()) valid = true; else valid = false;
// direct
boolean valid = name ≠ null && !name.isEmpty();

// naming a condition
boolean isWeekend = day == SATURDAY || day == SUNDAY;
boolean isHoliday = holidays.contains(date);
if (isWeekend || isHoliday) { … }
```

Comparing a boolean to `true` (`if (flag == true)`) is redundant and, with a `Boolean` wrapper, unboxes and can throw; write `if (flag)` and `if (!flag)`.

Guard clauses and early return

Nested conditions grow rightward until the happy path is buried:

```java
void process(Order order) {
    if (order ≠ null) {
        if (order.isPaid()) {
            if (!order.isShipped()) {
                ship(order);
            }
        }
    }
}
```

Invert each condition, handle the exceptional case, and leave:

```
void process(Order order) {
    if (order == null) return;
    if (!order.isPaid()) return;
    if (order.isShipped()) return;
    ship(order);
}
```

Same behaviour, no nesting; each line states one precondition. Throwing from a guard (`throw new IllegalArgumentException("order is null")`) is the variant for cases that should never happen. Interviewers notice guard clauses; so do code reviewers.

The ternary operator, as a conditional expression

For choosing a *value*, `?:` is often clearer than an `if` that assigns in both branches:

```
String label = count == 1 ? "item" : "items";
int max = a > b ? a : b;          // or Math.max(a, b)
```

Nest at most once, and never use it for side effects (`cond ? doA() : doB();` is legal and unreadable).

Short-circuit ordering for safety

Because `&&` and `||` stop early, the *order* of conditions is a correctness tool:

```
if (s ≠ null && s.length() > 3) …     // safe
if (s.length() > 3 && s ≠ null) …     // NullPointerException when s is null
if (i < arr.length && arr[i] == target) …   // bounds check first
```

Put the cheap, protective check first; the expensive or dangerous one second.

Comparing objects in conditions

`==` on references is identity. `if (name == "admin")` compiles and is wrong; `if ("admin".equals(name))`. For enums, `==` is correct and preferred (Module 10). For wrappers, `equals` or unbox. For `null` checks, only `==` / `≠` work.

Pattern matching for `instanceof` (Java 16+)

```
if (shape instanceof Circle c && c.radius() > 1) {
    System.out.println("big circle " + c.radius());
}
```

The test and the cast happen together; `c` is in scope where the condition is true — including after a guard clause `if (!(obj instanceof String s)) return;` where `s` is in scope for the rest of the method. Module 8 covers it with inheritance.

Style checklist

- Braces always; one statement per line.
- Positive conditions where possible: `if (isReady)` rather than `if (!isNotReady)`.
- Avoid `else` after a `return` / `throw` / `continue` — the `else` is implied.
- Extract complex conditions into well-named booleans or methods (`isEligible(user)`).
- Order `else if` chains from most specific to most general, and make the ranges obviously exhaustive.
- Prefer `switch` (next lesson) when comparing one value against many constants.

Interview angle

- *"`if (x = 5)` — what happens?"* Compile error: an `int` is not a `boolean`. *"`if (flag = true)`?"* Compiles, assigns, always true.
- *"Which `if` does a dangling `else` bind to?"* The nearest one.
- *"Why `"admin".equals(name)` rather than `name.equals("admin")`?"* Null safety.

Key takeaways

- The condition must be a `boolean`; nothing converts to one.
- Braces always — indentation is not structure, and `else` binds to the nearest `if`.
- Conditions are values: return them, name them, combine them.
- Guard clauses flatten nesting; order short-circuit conditions for safety.

EXERCISE 1

Grades with guard clauses

Read an integer `n`, then `n` integer scores. For each score print its letter: `A` for 90–100, `B` for 80–89, `C` for 70–79, `D` for 60–69, `F` for 0–59. A score below 0 or above 100 is `invalid`.

Write a method `static String grade(int score)` that handles the invalid case with a **guard clause** first and then an `else if` chain (or early returns) from the highest band down.

Input: `n`, then `n` integers. **Output:** `n` lines.

INPUT 1

```
6
95 80 79 60 59 101
```

OUTPUT 1

```
A
B
C
D
F
invalid
```

INPUT 2

```
3
0 100 -1
```

OUTPUT 2

```
F
A
invalid
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static String grade(int score) {
        if (score < 0 || score > 100) return "invalid";
        if (score ⩾ 90) return "A";
        if (score ⩾ 80) return "B";
        if (score ⩾ 70) return "C";
        if (score ⩾ 60) return "D";
        return "F";
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            System.out.println(grade(in.nextInt()));
        }
    }
}
```

EXERCISE 2

Ticket pricing

Read an integer `n` and `n` lines `<age> <day> <member>` where `day` is a weekday name and `member` is `y` or `n`. The base price is 12; under-12s pay 6 and people 65 or older pay 8 (check the child rule first). On `Tuesday` every ticket is 2 cheaper (never below 0). Members then get 10% off, rounded to the nearest whole number with `Math.round`. Print one price per line. Write it as a chain of guard clauses, not nested ifs.

Example: `3` then `30 Monday n`, `8 Tuesday y`, `70 Friday y` →

```
12
4
7
```

INPUT 1

```
3
30 Monday n
8 Tuesday y
70 Friday y
```

OUTPUT 1

```
12
4
7
```

INPUT 2

```
2
12 Tuesday n
65 Tuesday y
```

OUTPUT 2

```
10
5
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static long price(int age, String day, boolean member) {
        int base = 12;
        if (age < 12) base = 6;
        else if (age ⩾ 65) base = 8;
        if (day.equals("Tuesday")) base = Math.max(0, base - 2);
        if (!member) return base;
        return Math.round(base * 0.9);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int age = in.nextInt();
            String day = in.next();
            boolean member = in.next().equals("y");
            System.out.println(price(age, day, member));
        }
    }
}
```

QUIZ

1. `int x = 3; if (x) { … }` — what happens?

A. Runs the block because 3 is non-zero

B. Compile error: an `int` is not a `boolean`

C. Runs the block only if x == 1

D. Throws at run time

2. In `if (a) if (b) x(); else y();` — which `if` does the `else` belong to?

A. `if (a)`

B. `if (b)` — the nearest unmatched `if`

C. Both

D. It is a compile error

3. Which is the null-safe way to test `name` against `"admin"`?

A. `name == "admin"`

B. `name.equals("admin")`

C. `"admin".equals(name)`

D. `name.compareTo("admin")`

4. `boolean flag = false; if (flag = true) { … }` — result?

A. Compile error

B. The block always runs: the assignment yields `true`

C. The block never runs

D. `flag` stays false

5. What is a guard clause?

A. An `else` at the end of every chain

B. An early `return` / `throw` at the top of a method that handles an exceptional case so the main logic is not nested

C. A `try` block

D. A `switch` default

ANSWER KEY · IF, ELSE AND THE SHAPE OF A DECISION

1. B Java conditions must be booleans; nothing converts to one implicitly. Write `if (x ≠ 0)`.

2. B The dangling-else rule binds an `else` to the closest `if` without one, regardless of indentation. Braces make the intent unambiguous.

3. C Calling `equals` on the literal never throws when `name` is null; the other forms either compare references or dereference `name`.

4. B An assignment expression has the assigned value, and since it is a `boolean` the `if` accepts it. This is the classic typo for `==`; write `if (flag)`.

5. B Guard clauses invert the condition and leave immediately, keeping the happy path flat and readable.

LESSON 2 · 15 MIN

switch — statements, expressions and arrows

`switch` compares one value against a list of constants. It has existed since Java 1.0 in a form inherited from C — with fall-through, the bug generator — and was reworked in Java 14 into a cleaner **arrow form** and a **switch expression** that yields a value. Modern Java code uses the new forms almost exclusively, but you will read the old one in every codebase, so both are here.

The classic statement (with fall-through)

```
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 6:
    case 7:
        System.out.println("Weekend");
        break;
    default:
        System.out.println("Midweek");
}
```

Execution jumps to the matching `case` label and then **runs downward until a `break`** — through following `case` labels if there is no `break`. That is *fall-through*, and it is what makes `case 6: case 7:` share a body. It is also the source of the classic bug: forget a `break` and the next case's code runs too. Compilers warn about it (`-Xlint:fallthrough`); reviewers reject it unless there is a `// fall through` comment.

The selector can be `byte`, `short`, `char`, `int` (and their wrappers), `String` (Java 7), or an `enum`. Not `long`, not `boolean`, not `double`. Case labels must be **compile-time constants** and unique. `default` can go anywhere but conventionally goes last.

The arrow form (Java 14+)

```
switch (day) {
    case 1 -> System.out.println("Monday");
    case 6, 7 -> System.out.println("Weekend");
    default -> System.out.println("Midweek");
}
```

`->` replaces `:` and there is **no fall-through**: exactly one branch runs. Multiple labels are separated by commas. A branch body is a single expression, a block `{ … }`, or a `throw`. Mixing arrow and colon forms in one `switch` is a compile error.

Switch expressions

A `switch` can produce a value:

```
String kind = switch (day) {
    case 1, 2, 3, 4, 5 -> "weekday";
    case 6, 7 -> "weekend";
    default -> throw new IllegalArgumentException("day " + day);
};

int days = switch (month) {
    case 2 -> leap ? 29 : 28;
    case 4, 6, 9, 11 -> 30;
    default -> {
        log("long month");
        yield 31;            // a block yields its value
    }
};
```

Rules:

The expression must be **exhaustive**: for an `int` or `String` selector that means a `default`; for an enum, covering every constant is enough (a `default` is then optional — and omitting it means adding a constant later is a compile error where you switch, which is usually what you want).

Each branch yields a value of a common type; `yield` is used inside a block. (`yield` is a contextual keyword — `return` would be wrong here.)

It ends with a semicolon, because it is an expression statement.

Colon-form cases can be used in an expression too (`case 1: yield "x";`) but nobody should.

Switch expressions make many `if/else` chains disappear and, because they are expressions, they can initialise a `final` variable in one step.

Strings and enums

```
switch (command.toLowerCase(Locale.ROOT)) {
    case "start", "run" → start();
    case "stop" → stop();
    default → unknown(command);
}

enum Level { LOW, MEDIUM, HIGH }
switch (level) {
    case LOW → …;          // constants are written unqualified: LOW, not Lev
    case MEDIUM, HIGH → …;
}
```

A `String` switch compares with `equals` after a `hashCode` dispatch — case-sensitive, and `null` throws `NullPointerException`. So does a null enum. Guard with an `if` before the switch (or `case null` in Java 21).

Scope inside a classic switch

The whole colon-form `switch` body is **one block**, so a variable declared in one `case` is in scope in later cases — but only *initialised* if that case ran:

```
switch (x) {
    case 1:
        int y = 10;
        break;
    case 2:
        y = 20;          // legal: same scope; y declared above
        System.out.println(y);
        break;
    case 3:
        System.out.println(y);   // compile error: y might not have been init
}
```

Wrap a case's body in braces to give it its own scope. The arrow form has no such issue — each branch is its own scope.

Choosing between `if` and `switch`

Use `switch` when one value is compared against several constants — it is clearer, and the compiler can dispatch with a jump table (`tableswitch`) or binary search (`lookupswitch`) rather than a chain of comparisons. Use `if` for ranges (`score ≥ 90`), for conditions on several variables, or when comparing non-constants. A `switch` on a `boolean` does not exist; that is an `if`.

Pattern matching for switch (Java 21)

Java 21 lets a `switch` test *types* and *patterns*, with guards:

```
String describe(Object o) {
    return switch (o) {
        case null -> "null";
        case Integer i when i > 100 -> "big int";
        case Integer i -> "int " + i;
        case String s -> "string of " + s.length();
        default -> "something else";
    };
}
```

And with sealed hierarchies (Module 9) the compiler checks exhaustiveness without a `default`. This is the direction the language is going. The runner behind these exercises is JDK

18, where this form is a preview feature, so the exercises stay with constants; the Modern Java module covers patterns in depth.

Common mistakes

Mistake	Effect
Missing `break` in colon form	Falls through into the next case
Duplicate case labels	Compile error
`case` with a variable	Compile error: needs a constant
Switch on `long` / `double`	Compile error
`null` selector	`NullPointerException`
Non-exhaustive switch expression	Compile error
`return` instead of `yield` in a block	Returns from the method — probably not intended

---

Interview angle

"What is fall-through?" Colon-form cases continue into the next case without `break`.

"Which types can a switch select on?" Integral primitives up to `int`, their wrappers, `char`, `String`, enums (and, from Java 21, any reference type via patterns).

"Difference between switch statement and expression?" The expression yields a value and must be exhaustive.

"What happens with a `null` string in a switch?" `NullPointerException`.

Key takeaways

Colon `case` falls through; arrow `case →` does not. Prefer arrows.

A switch expression yields a value with `→` or `yield`, and must be exhaustive.

Selectors: `int`-sized integrals, `char`, `String`, enums; labels are constants; `null` throws.

Enum switches covering every constant need no `default` — and then fail to compile when a constant is added, on purpose.

Java 21 adds type and guard patterns; the runner here is JDK 18.

EXERCISE 1

Days in a month

Read an integer `n`, then `n` lines each with a month number (1–12) and a year. Print the number of days in that month, using a **switch expression** over the month with a `default` that throws for an invalid month. February has 29 days in a leap year (divisible by 4, except centuries unless divisible by 400) and 28 otherwise.

Input: `n`, then `n` lines `month year`. **Output:** `n` lines.

Example: `2 2024` → `29`; `2 1900` → `28`; `4 2023` → `30`.

INPUT 1

```
3
2 2024
2 1900
4 2023
```

OUTPUT 1

```
29
28
30
```

INPUT 2

```
4
2 2000
2 2100
1 2023
12 1999
```

OUTPUT 2

```
29
28
31
31
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static boolean isLeap(int year) {
        return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
    }

    static int daysIn(int month, int year) {
        return switch (month) {
            case 2 -> isLeap(year) ? 29 : 28;
            case 4, 6, 9, 11 -> 30;
            case 1, 3, 5, 7, 8, 10, 12 -> 31;
            default -> throw new IllegalArgumentException("month " + month);
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            System.out.println(daysIn(in.nextInt(), in.nextInt()));
        }
    }
}
```

EXERCISE 2

A calculator with switch

Read an integer `n` and `n` lines `<a> <op> <b>` with `long` operands. Compute with a `switch` **expression** over the operator: `+`, `-`, `*`, `/` (integer division), `%`, and `^` meaning `a` to the power `b` (`b ≥ 0`, use a loop). Print the result, or `error: division by zero` for `/` and `%` with `b = 0`, or `error: unknown operator <op>`.

Example: `4` then `7 / 2`, `2 ^ 10`, `5 % 0`, `3 ? 4` →

```
3
1024
error: division by zero
error: unknown operator ?
```

INPUT 1

```
4
7 / 2
2 ^ 10
5 % 0
3 ? 4
```

OUTPUT 1

```
3
1024
error: division by zero
error: unknown operator ?
```

INPUT 2

```
3
-7 / 2
-7 % 3
3 ^ 0
```

OUTPUT 2

```
-3
-1
1
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static long power(long a, long b) {
        long r = 1;
        for (long i = 0; i < b; i++) r *= a;
        return r;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long a = in.nextLong();
            String op = in.next();
            long b = in.nextLong();
            if ((op.equals("/") || op.equals("%")) && b == 0) {
                System.out.println("error: division by zero");
                continue;
            }
            String out = switch (op) {
                case "+" -> String.valueOf(a + b);
                case "-" -> String.valueOf(a - b);
                case "*" -> String.valueOf(a * b);
                case "/" -> String.valueOf(a / b);
                case "%" -> String.valueOf(a % b);
                case "^" -> String.valueOf(power(a, b));
                default -> "error: unknown operator " + op;
            };
            System.out.println(out);
        }
    }
}
```

QUIZ

1. In a classic (colon) switch, what happens when a case has no `break`?

A. Compile error

B. Execution falls through into the next case's statements

C. The switch exits

D. The default runs

2. Which selector type is **not** allowed in a Java 17 switch?

A. `String`

B. `char`

C. `long`

D. An enum

3. A switch *expression* over an `int` with no `default` — what happens?

A. It compiles and yields 0 for unmatched values

B. Compile error: a switch expression must be exhaustive

C. Runtime exception

D. It yields `null`

4. Inside a block in a switch-expression branch, how do you produce the branch's value?

A. `return value;`

B. `yield value;`

C. `break value;`

D. `value;`

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

A. Runs the default

B. Throws `NullPointerException`

C. Matches nothing and continues

D. Compile error

ANSWER KEY · SWITCH — STATEMENTS, EXPRESSIONS AND ARROWS

1. B Fall-through is the C legacy: control continues downward until a `break`. Arrow cases (`→`) never fall through.

2. C `long`, `boolean`, `float` and `double` cannot be switched on. Integral types up to `int`, their wrappers, `char`, `String` and enums can.

3. B The compiler requires every possible value to produce a result. For an enum, covering all constants suffices; for `int` or `String`, a `default` is required.

4. B `yield` hands the value out of the block to the switch. `return` would return from the enclosing method.

5. B A `String` switch calls `hashCode()` on the selector first. Only Java 21's `case null` handles it inside the switch.

LESSON 3 · 14 MIN

Loops — while, do-while, for and for-each

Java has four loop forms and they are not interchangeable: each states something different about *what is known before the loop starts*. Choosing the right one is half of writing a clear loop; the other half is getting the boundaries right, which is where the off-by-one errors live. This lesson covers all four, the scope rules, and the boundary habits that eliminate the classic bugs.

`while`: repeat while a condition holds

```
int n = 27, steps = 0;
while (n != 1) {
    n = (n % 2 == 0) ? n / 2 : 3 * n + 1;
    steps++;
}
```

The condition is checked **before** each iteration, so the body may run zero times. Use `while` when you do not know how many iterations there will be — reading until end of input, converging on a value, draining a queue. Something in the body must eventually make the condition false, or you have an infinite loop; `while (true)` with a `break` inside is the accepted way to write "loop until something happens in the middle".

`do … while`: at least once

```
String answer;
do {
    answer = prompt("Continue? (y/n)");
} while (!answer.equals("y") && !answer.equals("n"));
```

The condition is checked **after** the body, so the body runs at least once — right for "ask, then check, repeat if bad". Note the semicolon after the `while`. Rarely needed; when you reach for it, you usually mean "do the thing once, then `while`" and this is exactly that.

`for`: counted iteration

```
for (int i = 0; i < n; i++) {
    …
}
```

Three parts in the header, all optional: **initialiser** (runs once), **condition** (checked before each iteration), **update** (runs after each iteration). `for (;;)` is an infinite loop. The loop variable is scoped to the loop — `i` does not exist after the closing brace, which is why you can write `for (int i …)` twice in a row.

The idiom for indices is `0` to `< n`, never `⩽ n - 1` or `1 … n` — matching Java's zero-based, half-open convention everywhere. Counting down: `for (int i = n - 1; i ⩾ 0; i--)`. Stepping: `for (int i = 0; i < n; i += 2)`. Two variables: `for (int i = 0, j = n - 1; i < j; i++, j--)` — the comma is allowed only in the init and update parts.

A `for` header can express things a `while` needs three lines for, but the reverse is also true — a `for` whose body modifies `i` is a `while` in disguise and should be written as one.

Enhanced `for` (for-each)

```
for (String name : names) {          // any Iterable, or an array
    System.out.println(name);
}
for (int v : new int[] {3, 1, 2}) { sum += v; }
for (char c : s.toCharArray()) { … }
for (Map.Entry<String, Integer> e : map.entrySet()) { … }
```

Reads every element in order, with no index and no way to go wrong on bounds. Use it whenever you do not need the index. Limitations you must know:

- **No index.** If you need `i`, use a classic `for` (or keep a counter).
- **Assigning to the loop variable does not change the collection.** `for (int v : arr) v = 0;` leaves `arr` untouched — `v` is a copy of each element.
- **You cannot remove from a collection you are iterating** — `list.remove(x)` inside a for-each over `list` throws `ConcurrentModificationException` (usually on the *next* iteration). Use an `Iterator` with `it.remove()`, or `removeIf` (Module 14).

The loop variable is fresh each iteration; it can be declared `final` and captured by a lambda.

Boundaries: the off-by-one family

Almost every loop bug is a boundary:

```
for (int i = 0; i ⩽ arr.length; i++)  arr[i]   // ArrayIndexOutOfBounds at i
for (int i = 1; i < arr.length; i++)   arr[i]   // skips element 0
for (int i = 0; i < s.length() - 1; i++) …      // skips the last char — some
while (i < n) { … }  vs  while (i ⩽ n) { … }   // runs n vs n + 1 times
```

Habits that prevent them:

- Half-open ranges: start inclusive, end exclusive, `< end`. The count is `end - start`.
- Test the loop on the smallest inputs by hand: `n = 0`, `n = 1`, `n = 2`.
- When comparing neighbours (`arr[i]` and `arr[i + 1]`), stop at `i < length - 1`; when the loop touches `arr[i - 1]`, start at `1`.
- Never modify the loop variable inside a `for` body; if you must skip, use `continue`.

Loop variable scope and lifetime

```
for (int i = 0; i < 3; i++) { }
System.out.println(i);          // compile error: i is out of scope

int i;
for (i = 0; i < 3; i++) { }
System.out.println(i);          // 3 — declared outside, survives
```

Declare the variable in the header unless you need its final value afterwards (searching for an index, for instance). A variable declared **inside** the body is created fresh each iteration — a new `int` or a new reference — which matters for lambdas and for objects: `List<int[]> rows` filled with a `new int[3]` created *inside* the loop holds distinct arrays; created *outside*, every entry is the same array.

Nested loops

```
for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
        grid[r][c] = r * cols + c;
    }
}
```

Two loops give `rows × cols` iterations — quadratic when both are `n`. Name the variables for what they index (`r` / `c`, `i` / `j` for generic pairs) and keep the inner body short. Leaving a nested loop early needs a labelled `break` (next lesson) or a flag.

Infinite loops and termination

`while (true)`, `for (;;)` and a `for` whose update never reaches the condition are legitimate when a `break` or `return` inside ends them; a loop with *no* exit is a bug, and one with an exit that is never reached is a bug the tests may not catch. For `double` counters, the loop may never hit the exact stopping value — count with an `int` and compute the double (Module 2).

Performance notes

Hoist invariants: computing `list.size()` in the condition is fine (it is O(1)) but calling a method that scans the collection each iteration is not.

`for (int i = 0; i < arr.length; i++)` over an array is as fast as it gets; the JIT removes bounds checks it can prove safe.

For-each over an `ArrayList` uses an `Iterator` — negligible cost. Over a `LinkedList`, for-each is O(n) while indexed `get(i)` is O(n²) — always for-each there.

Interview angle

"Difference between `while` and `do-while`?" Condition before vs after; `do` runs at least once.

"Can you modify a list inside a for-each?" Not structurally — `ConcurrentModificationException`. Use an iterator's `remove`, `removeIf`, or a classic index loop going backwards.

"What is the scope of a `for` loop variable?" The loop; it is gone after the closing brace.

"How many times does `for (int i = 0; i ⩽ n; i++)` *run?"* `n + 1`.

Key takeaways

`while` for unknown counts (checked first), `do-while` for at-least-once, `for` for counted loops, for-each for "every element".

Half-open ranges (`0` to `< n`) and testing `n = 0, 1, 2` by hand kill off-by-one errors.

The `for` variable lives only in the loop; body variables are fresh each iteration.

For-each gives copies of elements and forbids structural modification.

EXERCISE 1

Collatz steps

Read an integer `n`, then `n` positive integers. For each starting value `x`, repeatedly apply: if `x` is even, `x = x / 2`; otherwise `x = 3x + 1` — until `x` becomes 1. Print `steps=<count> peak=<largest value seen, including the start>`.

Use a `while` loop, and a `long` for `x` — intermediate values can exceed `int`.

Input: `n`, then `n` integers. **Output:** `n` lines.

Example: `27` → `steps=111 peak=9232`; `1` → `steps=0 peak=1`.

INPUT 1

```
3
27 1 6
```

OUTPUT 1

```
steps=111 peak=9232
steps=0 peak=1
steps=8 peak=16
```

INPUT 2

```
2
97 2
```

OUTPUT 2

```
steps=118 peak=9232
steps=1 peak=2
```

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();
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            long x = in.nextLong();
            long peak = x;
            int steps = 0;
            while (x != 1) {
                x = (x % 2 == 0) ? x / 2 : 3 * x + 1;
                if (x > peak) peak = x;
                steps++;
            }
            out.append("steps=").append(steps).append(" peak=").append(peak).append(
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Multiplication grid

Read an integer `n` (`1 ≤ n ≤ 12`). Print the `n × n` multiplication table with every cell right-aligned in 4 characters (`%4d` , no trailing separator), then `diagonal=<sum of i*i for i in 1..n>` and `total=<sum of every cell>` . Nested `for` loops, one `printf` per cell, one `println` per row.

Example: `3` →

```
   1   2   3
   2   4   6
   3   6   9
diagonal=14
total=36
```

INPUT 1

```
3
```

OUTPUT 1

```
   1   2   3
   2   4   6
   3   6   9
diagonal=14
total=36
```

INPUT 2

```
1
```

OUTPUT 2

```
   1
diagonal=1
total=1
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        int n = new Scanner(System.in).nextInt();
        long diagonal = 0, total = 0;
        for (int i = 1; i ⩽ n; i++) {
            for (int j = 1; j ⩽ n; j++) {
                System.out.printf("%4d", i * j);
                total += i * j;
                if (i == j) diagonal += i * j;
            }
            System.out.println();
        }
        System.out.println("diagonal=" + diagonal);
        System.out.println("total=" + total);
    }
}
```

QUIZ

1. How many times does the body of `for (int i = 0; i ⩽ n; i++)` run?

A. `n`

B. `n + 1`

C. `n - 1`

D. Infinite

2. What is true of `do { … } while (cond);`?

A. The body may run zero times

B. The body runs at least once; the condition is checked after it

C. It is identical to `while`

D. It needs no semicolon

3. `for (int v : arr) v = 0;` — effect on `arr`?

A. Every element becomes 0

B. None: `v` is a copy of each element

C. Compile error

D. Only the first element changes

4. After `for (int i = 0; i < 3; i++) {}`, what is `i`?

A. 3

B. 2

C. 0

D. Not accessible — `i` is scoped to the loop

5. Removing an element with `list.remove(x)` inside `for (X x : list)` typically causes…

A. Nothing unusual

B. `ConcurrentModificationException`

C. A compile error

D. The loop to restart

ANSWER KEY · LOOPS — WHILE, DO-WHILE, FOR AND FOR-EACH

1. B Indices 0 through n inclusive is n + 1 iterations. The half-open `i < n` runs exactly n times.

2. B The check comes after the body, so one iteration is guaranteed. The trailing semicolon is required.

3. B The enhanced for loop hands you the element's value (or reference), not a slot in the array. To modify, use an index loop.

4. D A variable declared in the `for` header lives only inside the loop. Declare it before the loop if you need its final value.

5. B Structural modification during for-each iteration is detected by the iterator. Use `Iterator.remove`, `removeIf`, or an index loop.

LESSON 4 · 11 MIN

break, continue, labels and return

A loop's condition says when to stop *between* iterations. Real loops often need to stop or skip *in the middle* — a match found, a bad record to ignore, an inner loop that should end the outer one too. Java gives four tools: `break`, `continue`, labelled versions of both, and `return`. Each has a precise meaning, and using the wrong one is a common source of subtle bugs.

`break`: leave the loop now

```
int found = -1;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == target) {
        found = i;
        break;          // stop scanning; the rest of the array is not examine
    }
}
```

`break` exits the **innermost** enclosing loop or `switch` immediately; execution continues with the statement after it. It is the standard way to stop a search on the first hit, and the standard way out of `while (true)`.

Inside a `switch` that is inside a loop, `break` leaves the `switch`, not the loop — a frequent surprise:

```
while (running) {
    switch (cmd) {
        case "quit" -> { running = false; }   // break here would only leave t
        default -> handle(cmd);
    }
}
```

`continue`: skip to the next iteration

```java
for (String line : lines) {
    if (line.isBlank()) continue;      // skip blank lines
    if (line.startsWith("#")) continue; // and comments
    process(line);
}
```

`continue` abandons the rest of the current iteration and goes to the next: in a `for`, the **update** runs and then the condition is checked; in a `while`, the condition is checked. It is the loop equivalent of a guard clause — filters at the top, the real work below, no nesting.

A `continue` in a `while` loop whose increment is at the *bottom* of the body skips the increment and loops forever:

```java
int i = 0;
while (i < n) {
    if (skip(i)) continue;     // i is never incremented → infinite loop
    work(i);
    i++;
}
```

This is one concrete reason to prefer `for` when there is an update step.

Labels: breaking out of nested loops

`break` and `continue` reach only the innermost loop. To leave an outer loop from inside an inner one, label the outer loop:

```
outer:
for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
        if (grid[r][c] == target) {
            System.out.println("found at " + r + "," + c);
            break outer;            // leaves BOTH loops
        }
    }
}

next:
for (int[] row : rows) {
    for (int v : row) {
        if (v < 0) continue next;   // abandon this row, go to the next
    }
    System.out.println("row is all non-negative");
}
```

A label is an identifier followed by a colon, placed before a statement (normally a loop). `break label` exits the labelled statement; `continue label` goes to the next iteration of the labelled loop. Labels are not `goto` — you can only break out of an enclosing labelled statement, never jump into one.

Labels are legitimate and clearer than a `found` flag checked in every loop condition. But if a nested loop needs one, consider extracting the inner search into a **method** and using `return` — usually the cleanest answer.

`return`: leave the method

```
static int indexOf(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) return i;     // leaves the loop AND the method
    }
    return -1;
}
```

`return` inside a loop ends the whole method, loop included. For a search that lives in its own method this beats `break` plus a result variable: no flag, no state, the answer is returned where it is found. The `-1` after the loop is the "not found" path and the compiler insists on it (every path must return).

Extracting loops into small methods that `return` early is one of the highest-leverage refactorings: `findFirstNegative`, `containsDuplicate`, `allPositive` each become a five-line method with an early return instead of a flag.

`break` in `switch` (classic form)

In the colon form, `break` ends the case; without it, execution falls through. In the arrow form there is nothing to break out of, and a `break` inside an arrow-case block that is inside a loop *does* break the loop — since there is no fall-through to stop, the `switch` is not what `break` targets… except it is: a `break` in an arrow `switch` **statement** still targets the switch. Use a labelled break or a flag when you need to leave a loop from inside any switch.

Flags versus early exits

```
boolean found = false;
for (int i = 0; i < n && !found; i++) {
    if (match(i)) found = true;
}
```

The flag-in-the-condition style works but spreads the exit logic across two places. `break` (or `return` from a method) says the same thing in one. Prefer the early exit; use a flag only when you need to *remember* the outcome after the loop and the loop must also finish for other reasons.

Unreachable code

```
for (…) {
    break;
    System.out.println("never");     // compile error: unreachable statement
}
```

Java refuses to compile statements it can prove unreachable — after `break`, `continue`, `return`, `throw`, or inside `while (false)`. (`if (false)` is exempt, for conditional compilation.) A `return` in both branches of an `if/else` followed by more code is the usual way to meet this error.

Interview angle

"How do you exit two nested loops?" A labelled `break`, or move the loops into a method and `return`.

"What does `break` do inside a `switch` inside a loop?" Leaves the switch only.

"Difference between `break` and `continue`?" `break` leaves the loop; `continue` skips to the next iteration.

"Does Java have `goto`?" It is a reserved word but not implemented; labelled break/continue are the structured alternative.

Key takeaways

`break` leaves the innermost loop or switch; `continue` skips to the next iteration (running a `for`'s update).

Labels let `break` / `continue` target an outer loop; extraction into a method with `return` is often cleaner.

`continue` in a `while` with a bottom increment loops forever.

The compiler rejects unreachable statements — a signal that the control flow is not what you think.

EXERCISE 1

Read until END

Read lines until a line that is exactly `END` (or until input runs out). Skip blank lines and lines that start with `#` (after trimming). Print every other line numbered from 1 as `<k>: <line>`, then a final line `skipped=<count of skipped lines>`.

Use `break` for `END` and `continue` for the skipped lines.

Example input:

```
alpha
# comment

beta
END
gamma
```

Output:

```
1: alpha
2: beta
skipped=2
```

INPUT 1

```
alpha
# comment

beta
END
gamma
```

OUTPUT 1

```
1: alpha
2: beta
skipped=2
```

INPUT 2

```
one
two
```

OUTPUT 2

```
1: one
2: two
skipped=0
```

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));
        StringBuilder out = new StringBuilder();
        int k = 0, skipped = 0;
        String line;
        while ((line = br.readLine()) != null) {
            String t = line.trim();
            if (t.equals("END")) break;
            if (t.isEmpty() || t.startsWith("#")) {
                skipped++;
                continue;
            }
            k++;
            out.append(k).append(": ").append(t).append('\n');
        }
        out.append("skipped=").append(skipped).append('\n');
        System.out.print(out);
    }
}
```

EXERCISE 2

First repeat in a grid

Read `r` and `c` and an `r × c` grid of integers. Scanning row by row, left to right, find the **first cell whose value has already appeared** and print `repeat <value> at (<row>,<col>)` (0-based), leaving both loops with a **labelled** `break` the moment it is found; print `none` if every value is distinct. Then print `scanned=<how many cells were examined>`.

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

```
repeat 2 at (1,1)
scanned=5
```

INPUT 1

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

OUTPUT 1

```
repeat 2 at (1,1)
scanned=5
```

INPUT 2

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

OUTPUT 2

```
none
scanned=4
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt();
        int[][] g = new int[r][c];
        for (int i = 0; i < r; i++) for (int j = 0; j < c; j++) g[i][j] = in.nextInt
        Set<Integer> seen = new HashSet<>();
        int scanned = 0;
        String found = "none";
        search:
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                scanned++;
                if (!seen.add(g[i][j])) {
                    found = "repeat " + g[i][j] + " at (" + i + "," + j + ")";
                    break search;
                }
            }
        }
        System.out.println(found);
        System.out.println("scanned=" + scanned);
    }
}
```

QUIZ

1. A `break` inside a `switch` that is inside a `while` loop...

A. Exits the loop

B. Exits the switch only

C. Exits both

D. Is a compile error

2. `continue` inside a `for` loop causes…

A. The loop to exit

B. The rest of the body to be skipped, then the update and condition run

C. The condition to be skipped

D. The update to be skipped

3. How do you exit an outer loop from inside an inner loop?

A. `break break;`

B. A labelled `break outer;`

C. `exit(outer)`

D. It is impossible without a flag

4. `for (…) { return x; System.out.println("done"); }` — the compiler says…

A. Nothing

B. Warning only

C. Error: unreachable statement

D. Error: missing return

5. Which is the cleanest way to search for a value in a nested structure?

A. A `found` flag tested in every loop condition

B. Nested loops in a method that `return`s on the first hit

C. `System.exit` on the first hit

D. Catching an exception

ANSWER KEY · BREAK, CONTINUE, LABELS AND RETURN

1. B `break` targets the innermost enclosing switch or loop — here the switch. Use a labelled break or a flag to leave the loop.

2. B In a `for`, `continue` jumps to the update expression, then re-checks the condition. In a `while` it goes straight to the condition — which is why a `while` with a bottom increment can loop forever.

3. B Label the outer loop (`outer:`) and write `break outer;`. Moving the loops into a method and using `return` is the other clean option.

4. C Java rejects statements it can prove unreachable, such as anything after an unconditional `return`, `break`, `continue` or `throw`.

5. B A method that returns the answer where it is found needs no flag and no labels, and its "not found" value is the final `return`.

LESSON 5 · 14 MIN

Loop patterns every program is made of

Most loops are one of a dozen patterns: accumulate, count, find, check all or any, track a best, walk two things at once, generate pairs, read until a sentinel. Recognising the pattern lets you write the loop without thinking about it — and lets an interviewer see that you have. This lesson names them, shows the canonical form of each, and points out the boundary and initial-value traps in each.

Accumulate

```
long sum = 0;
for (int v : values) sum += v;

double product = 1.0;
for (double f : factors) product *= f;

StringBuilder sb = new StringBuilder();
for (String s : parts) sb.append(s);
```

The trap is the **identity** value: `0` for sums, `1` for products, an empty builder for concatenation. And the type: a sum of `int`s needs a `long` accumulator when it can overflow (Module 2).

Count

```
int evens = 0;
for (int v : values) if (v % 2 == 0) evens++;
```

A count is an accumulate of `1`s under a condition. A count over characters, words, or matches is the same shape.

Find (first match)

```java
int index = -1;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == target) { index = i; break; }
}
```

Or, better, as a method with `return i;` and `return -1;` after the loop. The "not found" value must be one that cannot be a real answer: `-1` for indices, `null` for objects (or `Optional`, Module 15).

All / any / none

```java
boolean allPositive = true;
for (int v : values) if (v <= 0) { allPositive = false; break; }

boolean anyNegative = false;
for (int v : values) if (v < 0) { anyNegative = true; break; }
```

Note the initial values: **all** starts `true` and looks for a counterexample; **any** starts `false` and looks for a witness. Both stop early. On an empty input, `all` is vacuously true and `any` is false — the mathematically right answers, and sometimes a surprise. As methods, these are `return false;` inside the loop and `return true;` after (for all), and the reverse for any.

Best so far (max / min / argmax)

```java
int max = Integer.MIN_VALUE;
int maxIndex = -1;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] > max) { max = arr[i]; maxIndex = i; }
}
```

Two ways to seed: with the type's extreme (`Integer.MIN_VALUE` for a max) or with the **first element** (`max = arr[0]`, loop from 1 — but then an empty array must be handled first). Seeding with `0` is the classic bug: an all-negative array reports a max of 0. `>` versus `>=` decides whether ties keep the first or the last occurrence — say which you want.

Two pointers

```
int lo = 0, hi = arr.length - 1;
while (lo < hi) {
    if (arr[lo] + arr[hi] == target) return true;
    if (arr[lo] + arr[hi] < target) lo++; else hi--;
}
```

Two indices moving toward each other (palindromes, pair sums in a sorted array, reversing in place) or in the same direction at different speeds (removing duplicates, sliding windows). The loop condition is `lo < hi` for "distinct pair", `lo ⩽ hi` for "may meet".

Sliding window

```
int best = 0, windowSum = 0;
for (int i = 0; i < arr.length; i++) {
    windowSum += arr[i];
    if (i ⩾ k) windowSum -= arr[i - k];       // drop the element that left t
    if (i ⩾ k - 1) best = Math.max(best, windowSum);
}
```

A fixed-size window over an array in one pass: add the incoming element, subtract the outgoing one. The two `if`s handle the warm-up while the window is still filling.

Pairs and combinations

```
for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)     // j starts at i + 1: each unordered p
        …
```

`j = 0` gives ordered pairs including self-pairs; `j = i` includes `(i, i)`; `j = i + 1` gives each pair once. Getting this bound wrong double-counts or misses the diagonal.

Sentinel and end-of-input

```
String line;
while ((line = br.readLine()) ≠ null) { … }         // stream ends with null

int x;
while ((x = in.nextInt()) ≠ 0) { … }                 // input ends with a 0 s

while (in.hasNextInt()) { process(in.nextInt()); }    // no sentinel: test ava
```

The assignment-in-condition idiom `(x = read()) ≠ END` reads and tests in one step; the parentheses around the assignment are mandatory because `≠` binds tighter than `=`.

Parallel iteration

```
for (int i = 0; i < Math.min(a.length, b.length); i++) {
    use(a[i], b[i]);
}
```

Two sequences in step: index both with one variable, and decide what happens when the lengths differ (stop at the shorter, or check they match up front).

Running state

```
int prev = arr[0];
for (int i = 1; i < arr.length; i++) {
    if (arr[i] < prev) return false;     // not sorted
    prev = arr[i];
}
```

Comparing each element with the previous one — sortedness, run detection, deltas. Start at index 1 and seed `prev` from index 0, after handling the empty case.

Do-something-n-times

```
for (int i = 0; i < n; i++) sb.append('*');
```

When the index is unused, some codebases write `for (int i = 0; i < n; i++)` anyway; `"*".repeat(n)` and `IntStream.range(0, n)` are the modern alternatives for the common cases.

Choosing the shape

Need	Shape
A total, a product, a concatenation	accumulate with the right identity
How many satisfy X	count
The first that satisfies X	find with early exit, return -1/null
Do all / does any satisfy X	all (start true) / any (start false), early exit
The largest/smallest and where	best-so-far seeded from the first element or the extreme
A pair with a property in sorted data	two pointers
Best fixed-length run	sliding window
Every pair	nested with `j = i + 1`
Read until done	sentinel / `null` / `hasNext`

Almost every "medium" interview problem is two of these composed. Streams (Module 15) give one-line versions of accumulate, count, find, all/any and max — knowing the loop forms is what lets you read and debug the stream ones.

Key takeaways

- Seed accumulators with the identity, maxima from the first element or the extreme — never 0.
- `all` starts true and hunts for a counterexample; `any` starts false and hunts for a witness; both exit early.

Pairs: `j = i + 1`. Neighbours: start at 1 with `prev`. Windows: add incoming, drop outgoing.

`(x = read()) ≠ END` needs its parentheses.

EXERCISE 1

Four patterns, one pass each

Read integers `n` and `k` (1 ≤ k ≤ n), then `n` integers. Print four lines:

`max=<largest value> at=<index of its FIRST occurrence>`

`run=<length of the longest non-decreasing run>` (consecutive elements where each is ≥ the previous)

`window=<largest sum of k consecutive elements>`

`allPositive=<true|false>`

Seed the max from the first element, not from 0.

Example: `6 2` then `3 -1 4 4 5 2` →

```
max=5 at=4
run=4
window=9
allPositive=false
```

INPUT 1

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

OUTPUT 1

```
max=5 at=4
run=4
window=9
allPositive=false
```

INPUT 2

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

OUTPUT 2

```
max=5 at=4
run=5
window=15
allPositive=true
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        String[] t = new String(System.in.readAllBytes()).trim().split("\\s+");
        int n = Integer.parseInt(t[0]);
        int k = Integer.parseInt(t[1]);
        long[] a = new long[n];
        for (int i = 0; i < n; i++) a[i] = Long.parseLong(t[2 + i]);

        long max = a[0];
        int at = 0;
        for (int i = 1; i < n; i++) {
            if (a[i] > max) { max = a[i]; at = i; }
        }

        int run = 1, best = 1;
        for (int i = 1; i < n; i++) {
            run = a[i] ⩾ a[i - 1] ? run + 1 : 1;
            if (run > best) best = run;
        }

        long window = 0, sum = 0;
        for (int i = 0; i < n; i++) {
            sum += a[i];
            if (i ⩾ k) sum -= a[i - k];
            if (i ⩵ k - 1 || (i ⩾ k && sum > window)) window = sum;
        }

        boolean allPositive = true;
        for (int i = 0; i < n; i++) {
            if (a[i] ⩽ 0) { allPositive = false; break; }
        }

        System.out.println("max=" + max + " at=" + at);
        System.out.println("run=" + best);
        System.out.println("window=" + window);
        System.out.println("allPositive=" + allPositive);
    }
}
```

EXERCISE 2

Runs, searches and thresholds

Read an integer `n`, `n` integers, and a threshold `t`. Print three lines, each one loop pattern:

`run: value <v> length <k>` — the longest run of equal adjacent values (the first such run on ties).

`firstNegative: <index or -1>` — a search with early exit.

`untilOver: <k>` — how many elements, from the start, are needed before the running sum exceeds `t` (`n` if it never does).

Example: `7` then `3 3 -1 5 5 5 2` then `10` →

```
run: value 5 length 3
firstNegative: 2
untilOver: 5
```

INPUT 1

```
7
3 3 -1 5 5 5 2
10
```

OUTPUT 1

```
run: value 5 length 3
firstNegative: 2
untilOver: 5
```

INPUT 2

```
3
1 1 1
100
```

OUTPUT 2

```
run: value 1 length 3
firstNegative: -1
untilOver: 3
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

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

        int bestValue = a[0], bestLen = 1, curLen = 1;
        for (int i = 1; i < n; i++) {
            curLen = a[i] == a[i - 1] ? curLen + 1 : 1;
            if (curLen > bestLen) { bestLen = curLen; bestValue = a[i]; }
        }
        System.out.println("run: value " + bestValue + " length " + bestLen);

        int firstNegative = -1;
        for (int i = 0; i < n; i++) {
            if (a[i] < 0) { firstNegative = i; break; }
        }
        System.out.println("firstNegative: " + firstNegative);

        long sum = 0;
        int k = 0;
        while (k < n && sum ⩽ t) sum += a[k++];
        System.out.println("untilOver: " + (sum > t ? k : n));
    }
}
```

QUIZ

1. Why must a maximum not be seeded with 0?

A. 0 is not an `int`

B. An input of all negative numbers would report 0 as the maximum

C. The loop would not terminate

D. It must be seeded with `null`

2. An `allPositive` loop over an **empty** array yields…

A. `false`

B. `true` — no counterexample exists

C. An exception

D. Undefined

3. To visit each unordered pair `(i, j)` exactly once, the inner loop starts at…

A. `j = 0`

B. `j = i`

C. `j = i + 1`

D. `j = n - 1`

4. Why are the parentheses in `while ((line = br.readLine()) ≠ null)` required?

A. They are optional

B. `≠` binds tighter than `=`; without them `line` would be assigned a boolean, a type error

C. For readability only

D. `readLine` needs them

5. In a fixed-size sliding window of width k, moving one step right means…

A. Re-summing all k elements

B. Adding the incoming element and subtracting the one that left

C. Sorting the window

D. Resetting the sum

ANSWER KEY · LOOP PATTERNS EVERY PROGRAM IS MADE OF

1. B Seed from the first element or from `Integer.MIN_VALUE` so the first comparison always replaces the seed.

2. B `all` starts true and looks for a counterexample; with no elements there is none. `any` over an empty array is false for the same reason.

3. C `j = i + 1` skips self-pairs and each pair's mirror image. `j = i` includes the diagonal; `j = 0` visits ordered pairs.

4. B Without parentheses the expression parses as `line = (br.readLine() ≠ null)`, assigning a `boolean` to a `String`.

5. B That is what makes the window O(n) overall instead of O(n·k).

CHECKPOINT · 22 MIN

Checkpoint — Control flow

This checkpoint covers `if` / `else`, `switch` in its classic, arrow and expression forms, the four loops, `break` / `continue` /labels/ `return`, and the standard loop patterns.

How it works. Twelve questions and two programs; 70% on the questions and both programs accepted clears the module. Retake as often as you like.

Before you start, make sure you can answer:

- Which `if` a dangling `else` belongs to.
- What fall-through is and which `switch` form has it.
- What a switch expression must be, and the keyword that yields a value from a block.
- The difference between `while` and `do-while`; how many times `i ⩽ n` runs.
- What `break` does inside a `switch` inside a loop.
- How `all` and `any` loops are seeded and why.

The programs are pattern exercises: a grid search that needs the right exit, and a classifier that wants a switch expression.

EXERCISE 1

Grid search

Read integers `R`, `C` and `target`, then an `R × C` grid of integers (row by row). Scanning row-major (row 0 left to right, then row 1, …), print `found <row> <col>` for the **first** cell equal to `target`, or `not found`. Leave both loops as soon as the cell is found — with a labelled `break`, or by putting the loops in a method that returns.

Example: `2 3 7` then `1 2 3` / `7 8 7` → `found 1 0`.

INPUT 1

```
2 3 7
1 2 3
7 8 7
```

OUTPUT 1

```
found 1 0
```

INPUT 2

```
2 2 9
1 2
3 4
```

OUTPUT 2

```
not found
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt(), target = in.nextInt();
        int[][] grid = new int[r][c];
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++)
                grid[i][j] = in.nextInt();

        int foundRow = -1, foundCol = -1;
        search:
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                if (grid[i][j] == target) {
                    foundRow = i;
                    foundCol = j;
                    break search;
                }
            }
        }
        System.out.println(foundRow == -1 ? "not found" : "found " + foundRow + " "
    }
}
```

EXERCISE 2

Opening hours

A shop is open Monday–Friday from 9 to 17 (an hour `h` counts as open when 9 ≤ h < 17), Saturday from 10 to 14, and closed on Sunday. Read an integer `n`, then `n` lines each with a day name (any letter case) and an hour (0–23). Print `open`, `closed`, or `unknown day` for a name that is not a weekday.

Use a **switch expression** on the lower-cased day name; use blocks with `yield` where a branch needs an `if`.

Example: `monday 9` → `open`; `Saturday 14` → `closed`; `funday 12` → `unknown day`.

INPUT 1

```
4
monday 9
Saturday 14
funday 12
SUNDAY 11
```

OUTPUT 1

```
open
closed
unknown day
closed
```

INPUT 2

```
3
friday 16
friday 17
saturday 10
```

OUTPUT 2

```
open
closed
open
```

REFERENCE SOLUTION

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

public class Main {
    static String status(String day, int h) {
        return switch (day.toLowerCase(Locale.ROOT)) {
            case "monday", "tuesday", "wednesday", "thursday", "friday" -> (h ⩾ 9 &
            case "saturday" -> {
                boolean open = h ⩾ 10 && h < 14;
                yield open ? "open" : "closed";
            }
            case "sunday" -> "closed";
            default -> "unknown day";
        };
    }

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            String day = in.next();
            int h = in.nextInt();
            out.append(status(day, h)).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 3

Roman numerals, both ways

Read an integer `n` and `n` tokens. A token made of digits (1–3999) is converted **to** a Roman numeral; a token made of the letters `IVXLCDM` is converted **from** Roman to a number (subtractive pairs like `IV` and `XC` apply). Print `<token> → <result>`.

Example: `4` then `1994 MCMXCIV 3999 IV` →

```
1994 → MCMXCIV
MCMXCIV → 1994
3999 → MMMCMXCIX
IV → 4
```

INPUT 1

```
4
1994 MCMXCIV 3999 IV
```

OUTPUT 1

```
1994 → MCMXCIV
MCMXCIV → 1994
3999 → MMMCMXCIX
IV → 4
```

INPUT 2

```
3
1 I 58
```

OUTPUT 2

```
1 → I
I → 1
58 → LVIII
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static final int[] VALUES = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4,
    static final String[] SYMBOLS = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X'

    static String toRoman(int n) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < VALUES.length; i++) {
            while (n >= VALUES[i]) {
                sb.append(SYMBOLS[i]);
                n -= VALUES[i];
            }
        }
        return sb.toString();
    }

    static int value(char c) {
        switch (c) {
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            default: return 1000;
        }
    }

    static int fromRoman(String s) {
        int total = 0;
        for (int i = 0; i < s.length(); i++) {
            int v = value(s.charAt(i));
            if (i + 1 < s.length() && v < value(s.charAt(i + 1))) total -= v;
            else total += v;
        }
        return total;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String token = in.next();
            boolean numeric = Character.isDigit(token.charAt(0));
            System.out.println(token + " → " + (numeric ? toRoman(Integer.parseInt(
        }
```

```
    }
}
```

CHECKPOINT QUESTIONS

1. Which of these compiles?

A. `if (count) …`

B. `if (name) …`

C. `if (list.isEmpty()) …`

D. `if (x = 5) …`

2. Guard clauses are used to…

A. Catch exceptions

B. Handle exceptional cases first with early returns, flattening the main path

C. Lock resources

D. Replace loops

3. Which switch form has no fall-through?

A. The colon form

B. The arrow form

C. Both

D. Neither

4. A switch expression over an enum that lists **every** constant and has no `default` …

A. Is a compile error

B. Compiles; adding a new constant later makes it a compile error, which is useful

C. Compiles and yields `null` for new constants

D. Needs `yield`

5. `while (i < n)` with `i` incremented at the *bottom* of the body and a `continue` in the middle can…

A. Skip the increment and loop forever

B. Increment twice

C. Exit early

D. Not compile

6. How many iterations does `for (int i = 10; i > 0; i -= 3)` perform?

A. 3

B. 4

C. 5

D. Infinite

7. `for (String s : list) { list.add("x"); }` — most likely result?

A. Infinite growth

B. `ConcurrentModificationException`

C. Compile error

D. Adds one element

8. What does `break` do in an arrow-form switch **statement** inside a loop?

A. Leaves the loop

B. Leaves the switch statement

C. Is a compile error

D. Restarts the switch

9. An `anyNegative` search should be seeded with…

A. `true`

B. `false`, and set to `true` on the first negative (then break)

C. `null`

D. The first element

10. Comparing each element with its predecessor, the loop should…

A. Start at 0 and read `arr[i - 1]`

B. Start at 1 after seeding `prev` from `arr[0]`

C. Start at `n`

D. Use a `do-while`

11. `int x = 5; String r = switch (x) { case 5 → "five"; };` — result?

A. `r` is "five"

B. Compile error: not exhaustive

C. `r` is null

D. Runtime error

12. The compiler reports "unreachable statement" for code placed…

A. After an `if (false)`

B. Directly after `break`, `continue`, `return` or `throw` in the same block

C. Inside a `default` branch

D. Before a label

ANSWER KEY · CHECKPOINT — CONTROL FLOW

1. C Only a boolean expression is accepted. `isEmpty()` returns one; the others are `int`, `String`, and an `int` assignment.

2. B Each guard states a precondition and leaves; what remains is the normal case with no nesting.

3. B `case … →` runs exactly one branch. The colon form continues into the next case unless it hits `break`.

4. B Exhaustiveness over an enum is satisfied by covering all constants. Omitting `default` makes the compiler flag every switch when a constant is added.

5. A `continue` jumps to the condition, bypassing the increment. A `for` loop's update runs on `continue`, which avoids this.

6. B i takes 10, 7, 4, 1 — four values above 0 — then becomes −2 and stops.

7. B Adding during for-each iteration is a structural modification; the iterator detects it on the next step.

8. B The switch is still the innermost breakable statement. To leave the loop, use a label or a flag.

9. B `any` looks for a witness. It starts false and becomes true at the first match; over an empty input it stays false.

10. B Reading `arr[i - 1]` at `i = 0` is out of bounds. Handle an empty array, seed from index 0, iterate from 1.

11. B A switch expression over an `int` must have a `default`; the compiler cannot know 5 is the only value.

12. B Statements the compiler can prove will never execute are errors. `if (false)` is deliberately exempt for conditional compilation.