Module 2 of 20

Types & operators

The eight primitives, variables and constants, conversions, integer overflow, floating point, every operator, and the wrapper classes.

Every value in a Java program is one of eight primitive types or a reference. Most numeric bugs — a sum that goes negative, a division that gives zero, a comparison that is randomly false, an equality test that works for small numbers only — come from not knowing exactly how those types behave.

This module fixes that. You will learn the sizes and ranges, what the compiler converts silently and what needs a cast, what happens on overflow and how to detect it, why floating point cannot hold 0.1 and what to use for money, the full operator table with its precedence traps, and the wrapper classes with the autoboxing bugs that follow them.

By the end you can predict the result of any arithmetic expression, choose the right type for any quantity, and explain the Integer cache in an interview.

CONTENTS

1	The eight primitive types	14 min
2	Variables, constants and var	12 min
3	Conversions, promotion and casting	15 min
4	Integer arithmetic, overflow and the Math class	14 min
5	Floating point — doubles, precision and BigDecimal	15 min
6	Operators and precedence	14 min
7	Wrapper classes and autoboxing	15 min
✓	Checkpoint — Types & operators	25 min

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

LESSON 1 · 14 MIN

The eight primitive types

Java has exactly eight primitive types, and there will never be a ninth. Everything else — every class, array, enum, record, interface — is a reference type. Knowing the eight cold, including their sizes, ranges and literal syntax, is the foundation of every numeric bug you will ever debug.

The table to memorise

Type	Size	Range	Default	Literal
`byte`	8-bit signed	−128 … 127	`0`	`(byte) 7`
`short`	16-bit signed	−32 768 … 32 767	`0`	`(short) 7`
`int`	32-bit signed	−2 147 483 648 … 2 147 483 647	`0`	`7`
`long`	64-bit signed	±9.22 × 10^{18}	`0L`	`7L`
`float`	32-bit IEEE 754	~±3.4 × 10^{38}, ~7 significant digits	`0.0f`	`7.0f`
`double`	64-bit IEEE 754	~±1.8 × 10^{308}, ~15–16 significant digits	`0.0`	`7.0`
`char`	16-bit unsigned	0 … 65 535 (UTF-16 code unit)	`'\u0000'`	`'A'`
`boolean`	1 bit of information (size unspecified)	`true` / `false`	`false`	`true`

Two numbers worth knowing by heart: `Integer.MAX_VALUE` is **2 147 483 647** (about 2.1 billion — a phone number or a file size in bytes can exceed it), and `Long.MAX_VALUE` is **9 223 372 036 854 775 807** (about 9.2 quintillion). All integral types are signed two's complement; there is no

`unsigned int` in Java, though `Integer.toUnsignedString` and friends exist for when you need to *interpret* bits that way.

The defaults matter only for **fields** (and array elements). A local variable has no default: reading it before assignment is a compile error, not a zero.

Integer literals

```
int decimal = 42;
int hex     = 0x2A;         // 42
int binary  = 0b101010;     // 42 (Java 7+)
int octal   = 052;          // 42 — a leading zero means octal; a classic trap
long big    = 3_000_000_000L;   // underscores anywhere between digits
int million = 1_000_000;
```

A plain integer literal is an `int`. `3000000000` without the `L` is a compile error ("integer number too large") because it does not fit in an `int`. The suffix can be lower-case `l` but never write it — `3000000000l` looks like 30000000001.

Underscores are for readability only and are removed by the compiler; they cannot start or end a literal or sit next to the `0x` / `0b` prefix or the `L` suffix.

Floating-point literals

```
double d = 3.14;      // double by default
float  f = 3.14f;     // the f is required; 3.14 alone is a double and will no
double e = 1e-9;      // scientific notation
double h = 0x1.8p1;   // hexadecimal floating point: 1.5 × 2¹ = 3.0 (rare)
```

`float f = 3.14;` fails with "possible lossy conversion from double to float". Use `double` unless you have a measured reason (graphics, huge arrays) to use `float`.

`char`: a number in disguise

```
char c = 'A';          // 65
char d = 65;           // also 'A' — an int constant that fits
char e = '\u0041';     // Unicode escape, also 'A'
char nl = '\n';        // escapes: \n \t \r \\ \' \" \0 \b \f
int code = c;          // 65, widening
char next = (char) (c + 1);   // 'B' — c + 1 is an int, so the cast is needed
```

`char` is unsigned 16-bit and arithmetic on it yields `int`. `'a' + 1` is `98`, not `'b'`. The classic use in interviews:

```
int digitValue = ch - '0';           // '7' → 7
int letterIndex = ch - 'a';          // 'c' → 2
char upper = (char) (ch - 'a' + 'A');
```

Since Java is UTF-16, a `char` is a *code unit*, not necessarily a whole character — an emoji is two `char`s. Module 3 returns to this.

`boolean`: not a number

Unlike C, `boolean` does not convert to or from integers. `if (x)` where `x` is an `int` is a compile error; write `if (x ≠ 0)`. `true` and `false` are the only values, and `Boolean.parseBoolean("TRUE")` is how you get one from text (anything but a case-insensitive "true" is `false`).

Why primitives exist at all

Java's designers wanted "everything is an object" but measured the cost: an `int` in a register is one machine word and one instruction to add; an `Integer` object is a heap allocation, a header, a pointer indirection, and garbage-collector work. For loops, arrays and arithmetic — the hot paths of every program — primitives are the difference between fast and unusable. The compromise is the eight primitives plus the wrapper classes (`Integer`, `Double`, …) for when an object is required, with **autoboxing** converting between them (a later lesson, and a rich source of bugs).

A consequence you feel constantly: generics cannot hold primitives. `List<int>` is a compile error; it must be `List<Integer>`. `int[]` on the other hand is a real array of primitives, contiguous and fast.

Choosing a type

Counting, indexing, most arithmetic → `int`.

Anything that can reach billions — timestamps in milliseconds, file sizes, money in the smallest unit, products of two `int`s → `long`.

Measurements and science → `double`. **Never money** (Module 2, lesson 5).

`byte` and `short` are for arrays of them (binary data, memory-tight structures), not for "small numbers": arithmetic on them promotes to `int` anyway.

`float` only for graphics APIs or huge datasets where memory halves.

Sizes are guaranteed

In C, `int` is "at least 16 bits" and varies by platform. In Java, `int` is 32 bits everywhere — the same `.class` file computes the same result on every JVM. This is part of the portability promise and why Java arithmetic is *specified* down to overflow behaviour (next lessons).

Interview angle

"What is the size of `boolean`*?"* Unspecified by the language; HotSpot uses one byte in fields and arrays.

"Is `char` *signed?"* No — it is the only unsigned primitive.

"What does `0.1f == 0.1` *give?"* `false`: the `float` is widened to a `double`, and the two roundings of 0.1 differ.

"Why is `int i = 3_000_000_000;` *an error?"* The literal exceeds `Integer.MAX_VALUE`; it needs `L` and a `long`.

Key takeaways

Eight primitives: four integral (`byte`, `short`, `int`, `long`), two floating (`float`, `double`), `char`, `boolean`. Fixed sizes everywhere.

Integer literals are `int` unless suffixed `L`; decimal literals are `double` unless suffixed `f`.

`char` is an unsigned 16-bit number; arithmetic on it gives `int`.

`boolean` never converts to a number. Fields default to zero/false/null; locals have no default.

EXERCISE 1

The smallest type that fits

Read an integer `n`, then `n` whole numbers (each fits in a `long`). For each, print the **smallest** integral type that can hold it: `byte`, `short`, `int` or `long`.

Use the `MIN_VALUE` / `MAX_VALUE` constants of the wrapper classes rather than typing the ranges.

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

Example: `3` then `100 40000 -129` →

```
byte
int
short
```

INPUT 1

```
3
100 40000 -129
```

OUTPUT 1

```
byte
int
short
```

INPUT 2

```
4
-128 127 128 -32769
```

OUTPUT 2

```
byte
byte
short
int
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        String[] tokens = new String(System.in.readAllBytes()).trim().split("\\s+");
        int n = Integer.parseInt(tokens[0]);
        StringBuilder out = new StringBuilder();
        for (int i = 1; i ⩽ n; i++) {
            long v = Long.parseLong(tokens[i]);
            String type;
            if (v ⩾ Byte.MIN_VALUE && v ⩽ Byte.MAX_VALUE) type = "byte";
            else if (v ⩾ Short.MIN_VALUE && v ⩽ Short.MAX_VALUE) type = "short";
            else if (v ⩾ Integer.MIN_VALUE && v ⩽ Integer.MAX_VALUE) type = "int";
            else type = "long";
            out.append(type).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Bits of an int

Read an integer `n` and `n` `int` values. For each print `<x>: bin=<Integer.toBinaryString> hex=<Integer.toHexString> ones=<Integer.bitCount> sign=<negative|non-negative>`. Negative numbers show their two's-complement pattern — thirty-two bits, most of them ones.

Example: `3` then `5 -1 255` →

```
5: bin=101 hex=5 ones=2 sign=non-negative
-1: bin=11111111111111111111111111111111 hex=ffffffff ones=32 sign=negative
255: bin=11111111 hex=ff ones=8 sign=non-negative
```

INPUT 1

```
3
5 -1 255
```

OUTPUT 1

```
5: bin=101 hex=5 ones=2 sign=non-
negative
-1: bin=11111111111111111111111111111111
hex=ffffffff ones=32 sign=negative
255: bin=11111111 hex=ff ones=8
sign=non-negative
```

INPUT 2

```
1
0
```

OUTPUT 2

```
0: bin=0 hex=0 ones=0 sign=non-negative
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            System.out.println(x + ": bin=" + Integer.toBinaryString(x) + " hex=" +
                + " ones=" + Integer.bitCount(x) + " sign=" + (x < 0 ? "negative" :
        }
    }
}
```

QUIZ

1. How many primitive types does Java have?

A. 6

B. 8

C. 9 — including `String`

D. It depends on the JVM

2. What is wrong with `int n = 3000000000;`?

A. Nothing

B. The literal is too large for an `int`; it needs `L` and a `long`

C. Integers must use underscores above a million

D. `int` cannot be negative

3. What is the value of `'a' + 1`?

A. `'b'`

B. `98`

C. `"a1"`

D. A compile error

4. Which primitive is unsigned?

A. `byte`

B. `short`

C. `char`

D. `int`

5. A class declares `int count;` as a field and a method declares `int local;`. Which can be read without being assigned?

A. Both — both default to 0

B. Only `count`, which defaults to 0; reading `local` is a compile error

C. Only `local`

D. Neither

ANSWER KEY · THE EIGHT PRIMITIVE TYPES

1. B Eight: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`. `String` is a class. The count is fixed by the language specification.

2. B An unsuffixed integer literal is an `int`, and 3 000 000 000 exceeds `Integer.MAX_VALUE` (2 147 483 647). `long n = 3_000_000_000L;` compiles.

3. B `char` arithmetic promotes to `int`: `'a'` is 97, so the sum is the `int` 98. To get `'b'` you cast: `(char) ('a' + 1)`.

4. C `char` is a 16-bit unsigned value (0–65535), a UTF-16 code unit. All the other integral types are signed two's complement.

5. B Fields (and array elements) get default values. Locals do not — the compiler's definite-assignment check refuses a read before an assignment on every path.

LESSON 2 · 12 MIN

Variables, constants and var

A variable is a named slot with a declared type. Java has four kinds — local variables, parameters, instance fields and static fields — and they differ in where they live, when they get a default, and how long they last. This lesson also covers `final`, the difference between a constant and a merely-final variable, and the `var` keyword that Java 10 added.

Declaring and initialising

```
int count;              // declared, not initialised
int total = 0;          // declared and initialised
int a = 1, b = 2, c;    // several in one statement — legal, discouraged
final double TAX = 0.18;
```

The type comes first, then the name, then optionally `= value`. The type is fixed for the variable's life: `count = "seven";` is a compile error. You can declare a variable anywhere a statement can appear, and the style since Java 1.0 has been to declare each variable as close as possible to its first use, not at the top of the method.

The four kinds

Kind	Declared	Lives	Default value?	Lifetime
Local	inside a method/block	the thread's stack frame	**no** — must assign before read	until the block ends
Parameter	in a method signature	the stack frame	assigned by the caller	the call
Instance field	in a class, no `static`	inside each object on the heap	yes (0 / false / null)	the object's life
Static field	in a class, `static`	once per class, in the method area	yes	the class's life (the JVM's)

```java
public class Counter {
    private static int created = 0;   // static field: one for the class
    private int value;                // instance field: one per Counter, sta

    public void increment(int by) {   // parameter
        int next = value + by;        // local
        value = next;
        created++;
    }
}
```

Definite assignment

The compiler proves that every local variable is assigned on every path before it is read:

```java
int x;
if (flag) x = 1;
System.out.println(x);     // error: variable x might not have been initialize
```

Add an `else` or an initial value. This is not paranoia: uninitialised memory reads are a top source of C bugs, and Java removes the whole class of them at compile time. The rule is called *definite assignment* and it is flow-sensitive — the compiler tracks branches and loops.

`final`: assign exactly once

```java
final int limit = 10;
limit = 11;                // error: cannot assign a value to final variable

final List<String> names = new ArrayList<>();
names.add("Ada");          // fine! the *reference* is final, the object is
```

`final` on a variable means the variable is assigned once. It says **nothing** about the object it refers to. A `final List` can still be added to; a `final int[]` can still have elements changed. Immutability of the *object* is a property of the class (Module 7), not of the variable.

A blank final can be assigned later, once, as long as definite assignment can prove it:

```
final int sign;
if (n < 0) sign = -1; else sign = 1;
```

Marking locals and parameters `final` is a style some teams use; the compiler does not care, except in one place: a local captured by a lambda or anonymous class must be *effectively final* (Module 11).

Constants

A **constant** is a `static final` field with a compile-time constant value:

```
public static final int MAX_RETRIES = 3;
public static final String GREETING = "Hello";
```

Naming: UPPER_SNAKE_CASE. Two technical consequences:

> The compiler **inlines** constant values into every class that uses them. If `Config.MAX_RETRIES` is a constant and you change it in `Config` without recompiling the classes that use it, they keep the old value. This is real and bites in libraries.
>
> A `static final` whose value is not a compile-time constant (`static final List<String> NAMES = List.of(...)`) is not inlined; it is initialised once when the class loads.

`Math.PI`, `Integer.MAX_VALUE`, `Long.MIN_VALUE` are the constants you will use most.

`var`: local type inference (Java 10+)

```
var count = 0;                              // int
var names = new ArrayList<String>();        // ArrayList<String>
var reader = new BufferedReader(new InputStreamReader(System.in));
```

`var` tells the compiler to infer the type from the initialiser. It is **still static typing** — `count = "x"` is still an error — and it applies only to local variables with an initialiser (and `for` loop variables, and `try`-with-resources). Not allowed: fields, parameters, return types, `var x;` with no initialiser, `var x = null;`, or `var` with an array initialiser `{1, 2}`.

Use it when the type is obvious from the right-hand side (`new ArrayList<String>()`) or long and unhelpful. Avoid it when the type carries information: `var result = service.compute();` hides what `result` is. `var` is a *reserved type name*, not a keyword, so old code with a variable named `var` still compiles.

Scope and shadowing

```
int i = 0;
for (int i = 0; i < 3; i++) { }     // error: variable i is already defined
```

A local variable's scope is its block. You cannot declare a local that shadows another local in an enclosing block of the same method. You **can** shadow a field with a local or parameter — and it is the source of one of the most common beginner bugs:

```
public class Point {
    private int x;
    public Point(int x) {
        x = x;          // assigns the parameter to itself; the field stays 0
    }
}
```

`this.x = x;` is the fix. IDEs warn about self-assignment; the compiler does not.

Naming

Lower camel case, descriptive: `elapsedMillis` , `isEmpty` , `userCount` . Single letters are fine for loop indices (`i` , `j`) and mathematical variables (`x` , `n`). Never `$` or `_` alone (`_` is a keyword since Java 9). Constants upper snake. A variable's name should say what it holds; its type says what it can do.

Key takeaways

Locals have no default and must be definitely assigned before use; fields default to zero/false/null.

`final` = assigned once. It freezes the variable, never the object.

Constants (`static final` with a constant value) are inlined by the compiler.

`var` infers a local's type from its initialiser; the type is still fixed and checked.

Shadowing a field with a parameter is legal and a trap; `this.field = param`.

EXERCISE 1

Temperature log

A sensor reports temperatures. Read an integer `n`, then `n` integers (degrees). Print three lines: the highest reading as `max=<v>`, the lowest as `min=<v>`, and how many readings were strictly below freezing as `freezing=<count>`.

Declare the freezing point as a named constant (`static final int FREEZING = 0;`) and use it rather than a bare `0`.

Input: `n` (at least 1), then `n` integers. **Output:** three lines.

INPUT 1

```
5
3 -2 7 0 -5
```

OUTPUT 1

```
max=7
min=-5
freezing=2
```

INPUT 2

```
1
-40
```

OUTPUT 2

```
max=-40
min=-40
freezing=1
```

REFERENCE SOLUTION

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

public class Main {
    static final int FREEZING = 0;

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        int freezing = 0;
        for (int i = 0; i < n; i++) {
            int t = in.nextInt();
            if (t > max) max = t;
            if (t < min) min = t;
            if (t < FREEZING) freezing++;
        }
        System.out.println("max=" + max);
        System.out.println("min=" + min);
        System.out.println("freezing=" + freezing);
    }
}
```

EXERCISE 2

Unit converter with constants

Declare `static final double KM_PER_MILE = 1.609344` and `KG_PER_POUND = 0.45359237`. Read an integer `n` and `n` lines `<value> <unit>` where the unit is `km`, `mi`, `kg` or `lb`, and print the value converted to the other unit of its pair with two decimals: `<value> <unit> = <converted> <other>`. Use `Locale.ROOT` for the formatting.

Example: `2` then `10 km`, `150 lb` →

```
10.00 km = 6.21 mi
150.00 lb = 68.04 kg
```

INPUT 1

```
2
10 km
150 lb
```

OUTPUT 1

```
10.00 km = 6.21 mi
150.00 lb = 68.04 kg
```

INPUT 2

```
2
26.2 mi
1 kg
```

OUTPUT 2

```
26.20 mi = 42.16 km
1.00 kg = 2.20 lb
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static final double KM_PER_MILE = 1.609344;
    static final double KG_PER_POUND = 0.45359237;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            double value = in.nextDouble();
            String unit = in.next();
            double converted;
            String other;
            switch (unit) {
                case "km" -> { converted = value / KM_PER_MILE; other = "mi"; }
                case "mi" -> { converted = value * KM_PER_MILE; other = "km"; }
                case "kg" -> { converted = value / KG_PER_POUND; other = "lb"; }
                default -> { converted = value * KG_PER_POUND; other = "kg"; }
            }
            System.out.println(String.format(Locale.ROOT, "%.2f %s = %.2f %s", value
        }
    }
}
```

QUIZ

1. `final List<String> names = new ArrayList<>(); names.add("Ada");` — what happens?

A. Compile error: `names` is final

B. Runtime exception

C. It compiles and adds "Ada": `final` fixes the reference, not the object

D. It compiles but the add is ignored

2. Which declaration is **not** allowed?

A. `var x = 10;`

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

C. `var x;`

D. `for (var i = 0; i < 3; i++) {}`

3. In `class Point { int x; Point(int x) { x = x; } }`, what does the constructor do?

A. Sets the field to the parameter

B. Assigns the parameter to itself; the field stays 0

C. Compile error: duplicate variable

D. Throws at run time

4. Why can changing a `public static final int LIMIT = 10;` in a library break code compiled against the old value?

A. Static fields are cached by the JVM

B. Compile-time constants are inlined into the classes that use them

C. Final fields cannot be changed at all

D. It cannot break anything

5. Where does a local variable live?

A. On the heap, inside the object

B. In the method area

C. In the current thread's stack frame

D. In a static table

ANSWER KEY · VARIABLES, CONSTANTS AND VAR

1. C `final` means the variable is assigned once. The `ArrayList` it points at is still mutable. Object immutability is a property of the class, not of the variable holding it.

2. C `var` infers from the initialiser, so it needs one. `var x;` and `var x = null;` are both rejected. Fields and parameters cannot use `var` either.

3. B The parameter shadows the field, so `x = x` is a self-assignment. The fix is `this.x = x`. The compiler allows it; a good IDE warns.

4. B A `static final` with a constant initialiser is a compile-time constant; `javac` copies its value into every class that reads it. Those classes keep the old value until recompiled.

5. C Locals and parameters live in the frame pushed for the method call and vanish when it returns. Objects they refer to live on the heap; the *reference* is on the stack.

LESSON 3 · 15 MIN

Conversions, promotion and casting

Java moves values between numeric types constantly — `int` into `long`, `char` into `int`, `double` into `int` — and it has precise rules for when that happens silently, when it needs a cast, and what is lost. Getting these rules right explains half of the "why is my answer wrong" questions in numeric code.

Widening: automatic, safe

A value may be converted *implicitly* to a type that can hold everything it can:

```
byte → short → int → long → float → double
        char → int
```

```
int i = 'A';          // 65
long l = i;           // fine
double d = l;         // fine
float f = l;          // allowed — but see below
```

Widening never throws and never needs a cast. One subtlety: `int → float` and `long → float` / `long → double` are widening by the rules but can **lose precision**, because a `float` has only 24 bits of mantissa and a `double` 53. `float f = 16_777_217;` stores 16 777 216. The language calls it widening because the *magnitude* is preserved.

Narrowing: explicit, lossy

Going the other way requires a cast, and the cast tells the compiler "I accept the loss":

```
int i = (int) 3.99;          // 3 — truncation toward zero, not rounding
int j = (int) -3.99;         // -3
byte b = (byte) 200;         // -56 — the low 8 bits, reinterpreted as signed
char c = (char) 65601;       // 'A' — 65601 mod 65536 = 65
int k = (int) 1e20;          // Integer.MAX_VALUE — saturates, does not wrap
long m = (long) Double.NaN;  // 0
```

The rules, precisely:

Floating → integral: truncate toward zero; if the result is out of range, clamp to `MIN_VALUE` / `MAX_VALUE`; `NaN` becomes 0.

Integral → smaller integral: keep the low-order bits and reinterpret. `(byte) 200` is `-56` because `200 = 0b11001000` and the top bit is now the sign.

`double` → `float`: round to the nearest representable float.

`Math.round(3.99)` gives `4L` (a `long`, note); `(int) Math.round(x)` is the usual "round to int". `Math.floor`, `Math.ceil` return `double`.

Compile-time constants get a pass

```
byte b = 100;          // fine: 100 is a constant that fits in a byte
byte c = 200;          // error: possible lossy conversion from int to byte
int x = 100;
byte d = x;            // error: x is a variable, even though its value fits
final int y = 100;
byte e = y;            // fine: y is a constant expression
```

The compiler allows an `int` **constant expression** to narrow to `byte`, `short` or `char` when it can see the value fits. A variable, even a `final` one assigned at run time, gets no such pass.

Binary numeric promotion

When an operator has two numeric operands, both are converted to a common type before the operation:

If either is `double`, both become `double`.

Else if either is `float`, both become `float`.

Else if either is `long`, both become `long`.

Else both become `int` — even if both were `byte`, `short` or `char`.

That last rule is the one that surprises:

```
byte a = 10, b = 20;
byte c = a + b;          // error: a + b is an int
byte d = (byte) (a + b); // fine

char ch = 'a';
ch = ch + 1;             // error: int cannot be assigned to char
ch++;                    // fine — see compound assignment below
```

There is no `byte + byte` in the JVM instruction set; `iadd` works on `int`s. So arithmetic on the small types always yields `int`.

```
int i = 7 / 2;           // 3 — both int, integer division
double d = 7 / 2;        // 3.0 — division happens in int, THEN widened
double e = 7 / 2.0;      // 3.5 — promotion to double before dividing
double f = (double) 7 / 2;   // 3.5 — cast binds tighter than /
double g = (double) (7 / 2); // 3.0 — cast applied to the int result
```

The second and last lines are the classic mistakes. Convert *before* the division.

Compound assignment hides a cast

`x += y` is defined as `x = (T)(x + y)` where `T` is the type of `x`. The implicit cast is why:

```
byte b = 10;
b += 5;          // fine — really b = (byte)(b + 5)
b = b + 5;       // error

int i = 10;
i += 3.7;        // fine — i = (int)(i + 3.7) = 13. Silent truncation!
i *= 1.5;        // 19 — also silent
```

The same applies to `++` and `--` on `byte`, `short` and `char`. Be aware that `+=` with a `double` on the right silently truncates; it is legal and occasionally intended, but usually a bug.

Overflow is not a conversion error

```
int big = Integer.MAX_VALUE;
int wrapped = big + 1;                // -2147483648, no exception
long correct = (long) big + 1;        // 2147483648
long stillWrong = big + 1L;           // 2147483648 — the 1L promotes big to lo
long alsoWrong = (long) (big + 1);    // -2147483648 — overflow happened in int
```

Overflow is the next lesson, but note now the pattern: the *placement* of the cast decides whether the arithmetic happens in `int` or `long`.

Strings are not converted

There is no implicit conversion between numbers and `String` — except in `+`, where any operand being a `String` turns the whole thing into concatenation:

```
String s = "Total: " + 1 + 2;   // "Total: 12" — left to right, string first
String t = "Total: " + (1 + 2); // "Total: 3"
String u = 1 + 2 + " items";    // "3 items" — int + int first, then string
```

To go from text to a number, call the parser: `Integer.parseInt("42")`, `Double.parseDouble("3.5")`, `Long.parseLong("9000000000")`. Each throws `NumberFormatException` on bad input (`parseInt("4.0")`, `parseInt(" 42")`, `parseInt("")` all throw). To go from a number to text: `String.valueOf(42)`, `Integer.toString(42)`, or `"" + 42`.

Reference casts

Casting also exists for reference types (`(Dog) animal`) and means something completely different: a run-time check that the object really is that type, throwing `ClassCastException` if not. Nothing is converted — the same object is viewed through a narrower or wider type. Module 8 covers it; do not confuse the two uses of the same syntax.

Interview angle

"What does `(int) 3.7` give?" 3. *"And `Math.round(3.7)`?"* 4, as a `long`.

"Why does `byte b = a + b` fail?" Binary numeric promotion: the sum is an `int`.

"`int x = 10; x += 2.5;` — *compiles?*" Yes; `x` becomes 12 via the implicit cast in `+=`.

"`(byte) 130`?" −126.

Key takeaways

Widening is implicit (byte→short→int→long→float→double, char→int); narrowing needs a cast and truncates or wraps.

Arithmetic on `byte` / `short` / `char` produces an `int`; `int` with `long` / `float` / `double` promotes to the wider.

Integer division happens *before* any widening: cast an operand, not the result.

`+=` hides a cast to the left-hand type, silently truncating.

Strings never convert implicitly except through `+`; use `parseInt` / `valueOf`.

EXERCISE 1

Shout and count

Read one line of text. Print two lines:

The line with every lowercase letter converted to uppercase **using char arithmetic** (`(char) (c - 'a' + 'A')`), everything else unchanged.

`digits=<sum>` where `sum` is the sum of the numeric values of all digit characters (`c - '0'`).

Input: one line (may contain spaces). **Output:** two lines.

Example: `abc 123 XYZ!` →

```
ABC 123 XYZ!
digits=6
```

INPUT 1

```
abc 123 XYZ!
```

OUTPUT 1

```
ABC 123 XYZ!
digits=6
```

INPUT 2

```
java 17 lts
```

OUTPUT 2

```
JAVA 17 LTS
digits=8
```

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 line = br.readLine();
        StringBuilder shout = new StringBuilder();
        int digits = 0;
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (c ⩾ 'a' && c ⩽ 'z') {
                shout.append((char) (c - 'a' + 'A'));
            } else {
                shout.append(c);
            }
            if (c ⩾ '0' && c ⩽ '9') {
                digits += c - '0';
            }
        }
        System.out.println(shout);
        System.out.println("digits=" + digits);
    }
}
```

EXERCISE 2

Lossy or not

Read an integer `n` and `n` `long` values. For each print `<x>: int=<(int) x> short=<(short) x> byte=<(byte) x> lossy=<true if the int cast changed the value>`. A narrowing cast keeps the low bits and drops the rest — watch what happens past each type's range.

Example: `3` then `100 300 3000000000` →

```
100: int=100 short=100 byte=100 lossy=false
300: int=300 short=300 byte=44 lossy=false
3000000000: int=-1294967296 short=24064 byte=0 lossy=true
```

INPUT 1

```
3
100 300 3000000000
```

OUTPUT 1

```
100: int=100 short=100 byte=100
lossy=false
300: int=300 short=300 byte=44
lossy=false
3000000000: int=-1294967296 short=24064
byte=0 lossy=true
```

INPUT 2

```
2
-129 32768
```

OUTPUT 2

```
-129: int=-129 short=-129 byte=127
lossy=false
32768: int=32768 short=-32768 byte=0
lossy=false
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long x = in.nextLong();
            int asInt = (int) x;
            System.out.println(x + ": int=" + asInt + " short=" + (short) x + " byte
        }
    }
}
```

QUIZ

1. `double d = 7 / 2;` — what is `d`?

A. 3.5

B. 3.0

C. 4.0

D. Compile error

2. `byte b = 10; b = b + 5;` — what happens?

A. `b` becomes 15

B. Compile error: `b + 5` is an `int`

C. `b` becomes 5

D. Runtime overflow

3. What is `(byte) 200`?

A. 200

B. 127

C. -56

D. Compile error

4. `int x = 10; x += 3.7;` — what is `x`?

A. 13.7

B. 13

C. 14

D. Compile error

5. Which conversion is a *widening* conversion that can still lose precision?

A. `int` → `long`

B. `long` → `double`

C. `double` → `int`

D. `char` → `int`

ANSWER KEY · CONVERSIONS, PROMOTION AND CASTING

1. B Both operands are `int`, so the division is integer division giving 3, which is then widened to 3.0. Cast an operand — `7 / 2.0` — to get 3.5.

2. B Binary numeric promotion makes `b + 5` an `int`, which cannot be assigned to a `byte` without a cast. `b += 5` compiles because compound assignment includes an implicit cast.

3. C Narrowing keeps the low 8 bits: 200 is `11001000`, whose top bit is now the sign, giving −56.

4. B `x += 3.7` is `x = (int) (x + 3.7)`: 13.7 truncated to 13. The hidden cast is why it compiles when `x = x + 3.7` would not.

5. B A `double` has 53 bits of mantissa; a `long` has 64 bits. Large longs round when widened. The rules call it widening because the magnitude is preserved.

LESSON 4 · 14 MIN

Integer arithmetic, overflow and the Math class

Integer arithmetic in Java is exact, fast, and *silently wrong* the moment a result leaves the range of its type. Every programmer who has computed the average of two large numbers, or the sum of a big array, or a hash, has met this. The lesson covers what the JVM actually does on overflow, how to detect and avoid it, the rules of `/` and `%` with negative numbers, and the tools in `java.lang.Math`.

Two's complement and wrap-around

An `int` is 32 bits interpreted in two's complement: the top bit is the sign. `Integer.MAX_VALUE` is `0x7FFFFFFF`; add one and the bits become `0x80000000`, which is `Integer.MIN_VALUE`. Nothing stops this — no exception, no flag, no warning:

```
int max = Integer.MAX_VALUE;
System.out.println(max + 1);          // -2147483648
System.out.println(Integer.MIN_VALUE - 1);   // 2147483647
System.out.println(Math.abs(Integer.MIN_VALUE)); // -2147483648  (!)
System.out.println(-Integer.MIN_VALUE);          // -2147483648  (!)
System.out.println(1_000_000 * 1_000_000);       // -727379968
```

`Math.abs(Integer.MIN_VALUE)` is negative because `2 147 483 648` does not exist as an `int`. This exact fact has caused real outages (hash-to-bucket code doing `Math.abs(hash) % n` and getting a negative index).

Multiplication overflows the earliest: two 5-digit numbers can exceed `int`. The **fix is `long`** — and the cast must happen *before* the multiplication:

```
int a = 100_000, b = 100_000;
long wrong = a * b;              // -1486618624: overflowed in int, then wider
long right = (long) a * b;       // 10000000000
long alsoRight = a * (long) b;
```

`long` overflows too, at 9.2×10^{18} — enough for most counting, not for a factorial of 21 or 2^{64}. Beyond that, `java.math.BigInteger`.

Detecting overflow: `Math.*Exact`

Since Java 8, the `Math` class has versions that throw instead of wrapping:

```
int sum = Math.addExact(a, b);           // ArithmeticException on overflow
int prod = Math.multiplyExact(a, b);
long big = Math.multiplyExact((long) a, b);
int neg = Math.negateExact(x);
int narrowed = Math.toIntExact(someLong);   // throws if it does not fit
```

Use these in financial code and anywhere silent wrap-around is worse than a crash. They JIT-compile to a single overflow-checked instruction, so the cost is negligible.

The average-of-two-ints bug

```
int mid = (low + high) / 2;              // overflows when low + high > MAX_VA
int mid = low + (high - low) / 2;        // safe when both are non-negative
int mid = (low + high) >>> 1;            // safe: unsigned shift treats the wi
```

Binary search in the JDK itself had this bug for nine years. In an interview, writing `low + (high - low) / 2` without being asked is a signal.

Division and remainder

Integer `/` **truncates toward zero**; `%` returns a result with **the sign of the dividend** (the left operand), such that `(a / b) * b + (a % b) == a` always holds:

```
 7 / 2   →  3       7 % 2  →  1
-7 / 2   → -3      -7 % 2  → -1
 7 / -2  → -3       7 % -2 →  1
-7 / -2  →  3      -7 % -2 → -1
```

So `%` is a *remainder*, not a mathematical modulo. A negative index into a circular buffer, `(i - 1) % n`, is negative when `i` is 0. The safe forms:

```
int mod = Math.floorMod(i - 1, n);        // always in [0, n) for positive n
int div = Math.floorDiv(-7, 2);           // -4: rounds toward negative infini
int adhoc = ((i - 1) % n + n) % n;        // the pre-Java-8 idiom
```

Division by zero: for integers, `x / 0` and `x % 0` throw `ArithmeticException: / by zero`. For floating point they do not — `1.0 / 0` is `Infinity` and `0.0 / 0` is `NaN` (next lesson).

Increment, decrement and evaluation order

```
int i = 5;
int a = i++;      // a = 5, i = 6   (post-increment: use, then add)
int b = ++i;      // b = 7, i = 7   (pre-increment: add, then use)
int c = i++ + ++i;   // 7 + 9 = 16; i = 9. Legal, unreadable. Never write this
```

Java evaluates operands **left to right**, always — unlike C, where such expressions are undefined. It is specified, so you can predict it, but code that relies on it is code reviewers reject.

The `Math` class

Static methods, no import needed:

Method	Note
`abs`, `max`, `min`	Overloaded for `int`, `long`, `float`, `double`
`pow(double, double)`	Returns `double` — `(int) Math.pow(2, 10)` is 1024, but beware precision for large results; prefer a loop or `1 << 10` for powers of two
`sqrt`, `cbrt`, `hypot`	`sqrt(-1)` is `NaN`
`floor`, `ceil`, `round`, `rint`	`round(double)` → `long`, `round(float)` → `int`; rounds half up (`round(-2.5)` is `-2`)
`floorDiv`, `floorMod`	The mathematical versions of `/` and `%`
`addExact` … `toIntExact`	Overflow-checked
`random()`	A `double` in [0, 1); for anything serious use `java.util.Random` or `ThreadLocalRandom`
`PI`, `E`	Constants
`sin`, `cos`, `log`, `log10`, `exp`	Standard transcendental functions

`Math.pow` in integer code is a smell: it computes in floating point and `Math.pow(10, 15)` may not be exactly `1e15` when cast back. Use a loop, `Math.multiplyExact` in a loop, or `BigInteger.pow`.

Bit tricks worth knowing

```
x & 1                // 1 if odd
x >> 1               // x / 2 for non-negative x (floors for negative — differs
x << 3               // x * 8 (until it overflows)
x & (x - 1)          // clears the lowest set bit; zero iff x is a power of two
Integer.bitCount(x) // popcount
Integer.MAX_VALUE == ~Integer.MIN_VALUE   // true
```

Shifts are covered in the operators lesson; they are here because interviewers ask "divide by two without `/`".

`BigInteger` for the rest

```
import java.math.BigInteger;
BigInteger f = BigInteger.ONE;
for (int i = 2; i ⩽ 50; i++) f = f.multiply(BigInteger.valueOf(i));
System.out.println(f);   // 30414093201713378043612608166064768844377641568960
```

Arbitrary precision, immutable (every operation returns a new object), with `add`, `subtract`, `multiply`, `divide`, `mod`, `pow`, `compareTo`, `gcd`, `isProbablePrime`. Slow relative to `long` — use it when you need it, not by default.

Interview angle

"Result of `Integer.MAX_VALUE + 1`?" `Integer.MIN_VALUE`. *"How do you detect that?"* `Math.addExact`, or check the sign flip, or compute in `long`.

"`-7 % 3`?" `-1`. *"How do you get 2?"* `Math.floorMod(-7, 3)`.

"Why `low + (high - low) / 2`?" Avoids overflow in `low + high`.

"Is `i = i++` a bug?" Yes: `i` is unchanged — the post-increment value is overwritten by the assignment.

Key takeaways

Overflow wraps silently; `Math.abs(MIN_VALUE)` is negative; multiply in `long` by casting *an operand*.

`Math.addExact` and friends throw on overflow.

`/` truncates toward zero and `%` takes the dividend's sign; use `floorMod` for a true modulo.

Integer division by zero throws; floating-point division by zero gives Infinity/NaN.

`BigInteger` when `long` is not enough.

EXERCISE 1

Products that fit

Read an integer `n`, then `n` pairs of integers `a b` (each within the `int` range). For each pair print the exact product `a × b` followed by a space and either `fits` if the product is within the `int` range or `overflow` if it is not.

Compute in `long` — cast **before** multiplying.

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

Example: `2` then `100000 100000` and `-7 3` →

```
10000000000 overflow
-21 fits
```

INPUT 1

```
2
100000 100000
-7 3
```

OUTPUT 1

```
10000000000 overflow
-21 fits
```

INPUT 2

```
3
46341 46341
46340 46340
-2147483648 -1
```

OUTPUT 2

```
2147488281 overflow
2147395600 fits
2147483648 overflow
```

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++) {
            int a = in.nextInt();
            int b = in.nextInt();
            long product = (long) a * b;
            boolean fits = product ⩾ Integer.MIN_VALUE && product ⩽ Integer.MAX_VA
            out.append(product).append(fits ? " fits" : " overflow").append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Digit tricks

Read an integer `n` and `n` non-negative `long` values. For each print `<x>: digits=<count> sum=<digit sum> reversed=<digits reversed as a number> palindrome=<true|false> gcd=<gcd(x, digit sum)>`. Use `/ 10` and `% 10` — no strings — and Euclid's algorithm for the gcd (`gcd(x, 0) = x`).

Example: `2` then `1221 907` →

```
1221: digits=4 sum=6 reversed=1221 palindrome=true gcd=3
907: digits=3 sum=16 reversed=709 palindrome=false gcd=1
```

INPUT 1

```
2
1221 907
```

OUTPUT 1

```
1221: digits=4 sum=6 reversed=1221
palindrome=true gcd=3
907: digits=3 sum=16 reversed=709
palindrome=false gcd=1
```

INPUT 2

```
2
0 10
```

OUTPUT 2

```
0: digits=1 sum=0 reversed=0
palindrome=true gcd=0
10: digits=2 sum=1 reversed=1
palindrome=false gcd=1
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    static long gcd(long a, long b) { return b == 0 ? a : gcd(b, a % b); }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long x = in.nextLong();
            long digits = 0, sum = 0, reversed = 0;
            long t = x;
            do {
                digits++;
                sum += t % 10;
                reversed = reversed * 10 + t % 10;
                t /= 10;
            } while (t > 0);
            System.out.println(x + ": digits=" + digits + " sum=" + sum + " reversed
        }
    }
}
```

QUIZ

1. What does `Integer.MAX_VALUE + 1` evaluate to?

A. It throws `ArithmeticException`

B. `Integer.MIN_VALUE`

C. `Long` 2147483648

D. 0

2. What is `-7 % 3` in Java?

A. 2

B. -1

C. 1

D. -2

3. Which midpoint expression avoids overflow for non-negative `low` and `high`?

A. `(low + high) / 2`

B. `low + (high - low) / 2`

C. `(low + high) * 0.5`

D. `high - low / 2`

4. `long r = 100000 * 100000;` — what is `r`?

A. 10000000000

B. 1410065408

C. -1486618624

D. Compile error

5. What does `Math.abs(Integer.MIN_VALUE)` return?

A. 2147483648

B. 2147483647

C. -2147483648

D. It throws

ANSWER KEY · INTEGER ARITHMETIC, OVERFLOW AND THE MATH CLASS

1. B Integer overflow wraps silently in two's complement. Use `Math.addExact` to get an exception, or compute in `long`.

2. B `%` is a remainder whose sign follows the dividend: −7 = 3 × (−2) + (−1). `Math.floorMod(-7, 3)` gives the mathematical 2.

3. B `low + high` can exceed `Integer.MAX_VALUE`; `high - low` cannot. `(low + high) >>> 1` is the other safe form.

4. C Both literals are `int`, so the multiplication overflows in `int` (giving −1486618624) and only then widens to `long`. Write `100000L * 100000`.

5. C The positive 2 147 483 648 does not exist as an `int`, so negating `MIN_VALUE` wraps back to itself. `Math.absExact` (Java 15+) throws instead.

LESSON 5 · 15 MIN

Floating point — doubles, precision and BigDecimal

`0.1 + 0.2 == 0.3` is `false` in Java, as in every language that uses IEEE 754 binary floating point. This is not a bug and not a Java quirk — it is what happens when you store base-10 fractions in base-2 — but a programmer who does not understand *why* will write money code that loses cents, comparisons that fail randomly, and loops that never terminate. This lesson makes floating point predictable.

How a `double` stores a number

A `double` is 64 bits: 1 sign bit, 11 exponent bits, 52 fraction bits (plus an implicit leading 1, so 53 bits of precision). The value is `±1.fraction × 2^exponent`. Consequences:

- It can represent exactly any number of the form `m × 2^e` with `m` fitting in 53 bits: 0.5, 0.25, 3.0, 1 099 511 627 776, and every integer up to 2^{53} (about 9×10^{15}).
- It **cannot** represent 0.1, 0.2 or 0.3 exactly, because they are not sums of powers of two. `0.1` is stored as `0.1000000000000000055511151231257827…`.
- 15–17 significant decimal digits survive a round trip; beyond that, digits are noise.

```
System.out.println(0.1 + 0.2);            // 0.30000000000000004
System.out.println(0.1 + 0.2 == 0.3);     // false
System.out.println(1.0 / 3);              // 0.3333333333333333
System.out.println((0.1 + 0.2) - 0.3);    // 5.551115123125783E-17
System.out.println(100.0 * 1.1);          // 110.00000000000001
```

`float` is the same design with 24 bits of precision (~7 digits). It exists for memory and graphics; for arithmetic use `double`.

Comparing doubles

Never `==` on computed values. Compare with a tolerance appropriate to the magnitude:

```
static boolean nearlyEqual(double a, double b, double eps) {
    return Math.abs(a - b) ⩽ eps * Math.max(1.0, Math.max(Math.abs(a), Math.a
}
nearlyEqual(0.1 + 0.2, 0.3, 1e-9);   // true
```

`Double.compare(a, b)` gives a total order (−1, 0, 1) that also handles `NaN` and `-0.0` consistently — use it in comparators. `==` is fine for values that were *assigned*, not computed (`x == 0.0` after `double x = 0.0`).

Loop counters must be integers:

```
for (double d = 0; d ≠ 1.0; d += 0.1) { … }   // never terminates: d skips ov
for (int i = 0; i < 10; i++) { double d = i / 10.0; … }   // right
```

Special values

Expression	Result
`1.0 / 0`	`Infinity`
`-1.0 / 0`	`-Infinity`
`0.0 / 0`	`NaN`
`Math.sqrt(-1)`	`NaN`
`Double.MAX_VALUE * 2`	`Infinity`
`1e-400`	`0.0` (underflow)

`NaN` (not a number) is *unordered*: every comparison with it is `false`, including `NaN == NaN`. Test with `Double.isNaN(x)` (or `x ≠ x`, the idiom the method uses). `NaN` is sticky — any arithmetic involving it gives `NaN` — which is how a single bad input silently poisons an average.

There are two zeros: `0.0` and `-0.0`. They are `==` but `Double.compare` orders `-0.0` first and `1 / -0.0` is `-Infinity`. You will meet `-0.0` when a negative number rounds to zero; formatting it prints `-0.0`, which surprises users.

Formatting

`System.out.println(double)` prints the *shortest* decimal that round-trips to the same double, hence `0.30000000000000004`. For display, format:

```
String.format("%.2f", 2.675)     // "2.68"
String.format("%.2f", 0.125)     // "0.13"
String.format("%.1f", 0.25)      // "0.3"
String.format("%10.3f", Math.PI) // "     3.142"
String.format("%e", 123456.789)  // "1.234568e+05"
```

A detail that differs from C: Java's `%f` rounds the **shortest decimal representation** of the double (the digits `Double.toString` would print) with `HALF_UP`, not the exact binary value. So `2.675` — stored as 2.67499999999999982… — still formats as `2.68`, because its shortest representation is `2.675`. C's `printf` would print `2.67`. `Math.round(x * 100) / 100.0` is the other common idiom for two places and *does* see the binary value: `Math.round(2.675 * 100)` is `Math.round(267.49999…)` = 267, giving 2.67. For money, neither is the tool — see below.

Money: never `double`

```
double price = 19.99;
double total = price * 3;          // 59.97000000000001
```

Accumulate that across a million transactions and cents disappear. Two correct approaches:

Integers in the smallest unit — store cents as `long`. Fast, exact, and what most payment systems do. `1999L * 3 == 5997L`.

`java.math.BigDecimal` — exact decimal arithmetic with explicit rounding:

```
import java.math.BigDecimal;
import java.math.RoundingMode;

BigDecimal price = new BigDecimal("19.99");      // from a String — never fro
BigDecimal total = price.multiply(BigDecimal.valueOf(3));            // 59.97
BigDecimal tax = total.multiply(new BigDecimal("0.18")).setScale(2, RoundingM
BigDecimal share = total.divide(BigDecimal.valueOf(7), 2, RoundingMode.HALF_E
```

Rules of `BigDecimal`:

Construct from a **String** or `BigDecimal.valueOf(double)`; `new BigDecimal(0.1)` gives the exact binary value, 0.1000000000000000055511….

`divide` without a scale and rounding mode throws `ArithmeticException` when the result does not terminate (`1/3`).

`equals` compares **scale too**: `new BigDecimal("2.0").equals(new BigDecimal("2.00"))` is `false`. Use `compareTo` for numeric equality.

Immutable: every operation returns a new object; `setScale` returns the rounded copy.

`RoundingMode.HALF_EVEN` ("banker's rounding") is what accounting usually wants; `HALF_UP` is what schools teach.

Parsing and printing

`Double.parseDouble("3.14")`, `Double.valueOf`, `Double.toString(x)`. `parseDouble` accepts `"1e3"`, `" 3.5 "` (whitespace is trimmed), `"NaN"`, `"Infinity"`, and hex floats; it rejects commas. It always uses the dot, whatever the locale — `Scanner.nextDouble` does not.

Performance note

Floating-point arithmetic is fast — as fast as integer arithmetic on modern CPUs. What is slow is `BigDecimal` (heap allocation per operation, roughly 100× slower) and `Math.pow` relative to a multiply. Do not avoid `double` for speed; avoid it for exactness.

`strictfp`

Historically, `strictfp` forced IEEE-exact results on hardware that used extended precision. Since Java 17 all floating-point arithmetic is strict and the keyword does nothing. If an interviewer mentions it, that is the answer.

Interview angle

"Why is `0.1 + 0.2 == 0.3`?" Binary fractions cannot represent 0.1 exactly; the two rounding errors do not cancel.

"How do you compare doubles?" With a tolerance, or `Double.compare` for ordering.

"How do you represent money?" `long` cents or `BigDecimal` from a String; never `double`.

"`NaN == NaN`?" `false`. `Double.isNaN`.

"What is `Double.MIN_VALUE`?" The smallest *positive* double (4.9e-324), not the most negative — that is `-Double.MAX_VALUE`. A classic trick question.

Key takeaways

Doubles are binary; 0.1 is not exact; 15–17 digits are meaningful.

Compare with a tolerance or `Double.compare`; never loop on a double counter.

`NaN` compares false to everything; `Infinity` results from division by zero.

Money: `long` in cents or `BigDecimal` built from strings with explicit rounding.

`Double.MIN_VALUE` is tiny and positive.

EXERCISE 1

The bill, to the cent

Read an integer `n`, then `n` prices, each a decimal with up to two places (such as `19.99`). Using `BigDecimal` built from the **strings**, print:

`subtotal=<sum>` with exactly two decimals,

`tax=<18% of the subtotal>` rounded to two decimals with `RoundingMode.HALF_UP`,

`total=<subtotal + tax>` with two decimals.

Input: `n`, then `n` prices (whitespace-separated). **Output:** three lines.

Example: `3` then `19.99 5.01 0.10` →

```
subtotal=25.10
tax=4.52
total=29.62
```

INPUT 1

```
3
19.99 5.01 0.10
```

OUTPUT 1

```
subtotal=25.10
tax=4.52
total=29.62
```

INPUT 2

```
2
0.1 0.2
```

OUTPUT 2

```
subtotal=0.30
tax=0.05
total=0.35
```

REFERENCE SOLUTION

```
import java.io.*;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        BigDecimal subtotal = BigDecimal.ZERO;
        for (int i = 0; i < n; i++) {
            subtotal = subtotal.add(new BigDecimal(in.next()));
        }
        subtotal = subtotal.setScale(2, RoundingMode.HALF_UP);
        BigDecimal tax = subtotal.multiply(new BigDecimal("0.18")).setScale(2, Round
        BigDecimal total = subtotal.add(tax).setScale(2, RoundingMode.HALF_UP);
        System.out.println("subtotal=" + subtotal.toPlainString());
        System.out.println("tax=" + tax.toPlainString());
        System.out.println("total=" + total.toPlainString());
    }
}
```

EXERCISE 2

Equal, close or exact

Read an integer `n` and `n` pairs of decimal numbers as **strings**. For each pair print `<a> <b>: == <r1> close <r2> exact <r3>` where `r1` compares them as `double`s with `==`, `r2` is whether they differ by less than `1e-9`, and `r3` compares them as `new BigDecimal(text)` with `compareTo`. The three answers disagree exactly when floating point is being floating point.

Example: `2` then `0.3 0.30000000000000004`, `1.0 1.00` →

```
0.3 0.30000000000000004: == false close true exact false
1.0 1.00: == true close true exact true
```

INPUT 1

```
2
0.3 0.30000000000000004
1.0 1.00
```

OUTPUT 1

```
0.3 0.30000000000000004: == false close
true exact false
1.0 1.00: == true close true exact true
```

INPUT 2

```
1
0.1 0.10000001
```

OUTPUT 2

```
0.1 0.10000001: == false close false
exact false
```

REFERENCE SOLUTION

```
import java.math.BigDecimal;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String a = in.next(), b = in.next();
            double da = Double.parseDouble(a), db = Double.parseDouble(b);
            boolean eq = da == db;
            boolean close = Math.abs(da - db) < 1e-9;
            boolean exact = new BigDecimal(a).compareTo(new BigDecimal(b)) == 0;
            System.out.println(a + " " + b + ": == " + eq + " close " + close + " ex
        }
    }
}
```

QUIZ

1. Why is `0.1 + 0.2 == 0.3` false?

A. Java rounds doubles to 15 digits on printing only

B. 0.1 and 0.2 cannot be represented exactly in binary and the rounding errors do not cancel

C. `==` on doubles compares references

D. It is true in Java 17+

2. Which is the right way to represent money?

A. `double` with `%.2f` when printing

B. `float`

C. `long` cents or `BigDecimal` constructed from a `String`

D. `BigDecimal` constructed from a `double`

3. What is `Double.MIN_VALUE`?

A. The most negative double

B. The smallest positive double (about 4.9e-324)

C. Negative infinity

D. NaN

4. `double x = 0.0 / 0;` and then `x == x` — what is printed?

A. `true`

B. `false`

C. `ArithmeticException`

D. Compile error

5. `new BigDecimal("2.0").equals(new BigDecimal("2.00"))` is…

A. `true`

B. `false`, because `equals` compares scale as well as value

C. A compile error

D. `true` only with `HALF_UP`

ANSWER KEY · FLOATING POINT — DOUBLES, PRECISION AND BIGDECIMAL

1. B Binary floating point stores sums of powers of two; 0.1 is a repeating fraction in base 2. The sum is 0.30000000000000004, not the double nearest 0.3.

2. C Integers in the smallest unit are exact; `BigDecimal` from a string is exact. `new BigDecimal(0.1)` captures the binary error, and `double` accumulates it.

3. B `MIN_VALUE` for the floating types is the smallest *positive* value. The most negative double is `-Double.MAX_VALUE`.

4. B `0.0 / 0` is `NaN`, and `NaN` is unordered — every comparison with it, including with itself, is `false`. Test with `Double.isNaN`. Only *integer* division by zero throws.

5. B `BigDecimal.equals` considers 2.0 and 2.00 different (scale 1 vs 2). `compareTo` returns 0 for them — use it for numeric equality.

LESSON 6 · 14 MIN

Operators and precedence

Java's operators are C's, with the sharp edges filed off: no pointer arithmetic, no comma operator, no implicit truthiness, and a specified evaluation order. This lesson covers each family with the details interviewers probe — short-circuiting, bitwise versus logical, the three shifts, and the precedence table you should know rather than guess.

Arithmetic

`+ - * / %`, unary `-` and `+`, `++` and `--`. Integer versus floating behaviour was covered in the last two lessons. `+` is also string concatenation when either operand is a `String`. There is no exponent operator; `Math.pow` or a loop.

Relational and equality

`< ⩽ > ⩾` work on numeric types (and `char`). `==` and `≠` work on everything — but for **references** they compare *identity* (is it the same object?), never contents. `"hi" == "hi"` may be true (string pool) or false (built at run time); `equals` is the content comparison. Module 3 and Module 8 make this precise; for now: `==` on objects is almost always wrong unless you mean "the same object".

Comparisons produce `boolean`, and `boolean` is not a number, so `a < b < c` does not compile (the first comparison yields a `boolean`, which `<` cannot take).

Logical: `&&`, `||`, `!` — and `&`, `|`, `^`

```
if (list ≠ null && !list.isEmpty()) …     // safe: right side not evaluated w
if (x > 0 || expensiveCheck()) …          // expensiveCheck runs only when x
```

`&&` and `||` **short-circuit**: the right operand is evaluated only if needed. This is a correctness tool, not just an optimisation — it is what makes `obj ≠ null && obj.method()` safe.

`&`, `|` and `^` on booleans are the *non-short-circuit* logical operators: both sides always evaluate. Use them when the right side has a side effect you want regardless (rare), or for

XOR, which has no `^^` form. On integers the same three symbols are bitwise (below).

`!` negates a boolean. There is no `not` keyword.

Bitwise and shifts

On integral types (`int`, `long`; smaller types promote to `int`):

Op	Meaning	Example (8-bit view)
`&`	AND	`0b1100 & 0b1010` → `0b1000`
`|`	OR	`0b1100 | 0b1010` → `0b1110`
`^`	XOR	`0b1100 ^ 0b1010` → `0b0110`
`~`	NOT (one's complement)	`~5` → `-6` (all bits flipped; `~x == -x - 1`)
`<<`	shift left, fill with 0	`1 << 4` → 16
`>>`	arithmetic shift right, fill with the sign bit	`-16 >> 2` → -4
`>>>`	logical shift right, fill with 0	`-16 >>> 28` → 15

Shift facts that come up:

- The shift distance is taken **modulo the width**: `1 << 32` is `1` (not 0) for an `int`, because 32 & 31 = 0. For `long`, modulo 64.
- `>>` keeps the sign, so it divides by 2^n rounding toward negative infinity (`-7 >> 1` is `-4`, while `-7 / 2` is `-3`).
- `>>>` exists because there are no unsigned types; it treats the value as unsigned bits. `(a + b) >>> 1` is the overflow-safe midpoint.
- `x ^ x == 0`, `x ^ 0 == x`, XOR is its own inverse — the basis of "find the single number" problems and swap-without-temp (`a ^= b; b ^= a; a ^= b;` — do not use in real code).
- Masks: `flags & FLAG_X` tests, `flags | FLAG_X` sets, `flags & ~FLAG_X` clears, `flags ^ FLAG_X` toggles.
- `Integer.toBinaryString(x)`, `Integer.bitCount`, `Integer.highestOneBit`, `Integer.numberOfTrailingZeros`, `Long.reverse` — the toolkit in the wrapper classes.

Assignment and compound assignment

`=` is an expression that yields the assigned value, so `a = b = c = 0` works (right-associative). The compound forms `+= -= *= /= %= &= |= ^= <<= >>= >>>=` include the implicit cast to the left-hand type (previous lessons). `x = x + 1` and `x += 1` and `x++` all do the same thing for an `int`; they differ for `byte` / `short` / `char` (only the last two compile without a cast).

The ternary `?:`

```
int max = a > b ? a : b;
String label = count == 1 ? "item" : "items";
```

The only three-operand operator. Both branches must produce compatible types, and numeric promotion applies to them: `true ? 1 : 2.0` is `1.0`, and `flag ? 'a' : 0` is a `char` because 0 is a constant that fits — but `flag ? 'a' : someInt` is an `int`. Nest it at most once; beyond that use `if`.

A famous trap:

```
Integer x = null;
int y = flag ? x : 0;   // NullPointerException when flag is true: x is unboxe
```

And the subtler one: `Object o = true ? Integer.valueOf(1) : Double.valueOf(2.0);` prints `1.0`, because both wrapper operands are unboxed and promoted to `double`.

`instanceof`

`obj instanceof Type` is `true` if `obj` is non-null and is a `Type` (or subtype). Since Java 16 it can bind a variable: `if (obj instanceof String s) { s.length(); }`. Module 8.

Precedence, from tightest to loosest

Level	Operators	Associativity
1	postfix `x++ x--`, member `.`, call `()`, index `[]`	left
2	unary `++x --x +x -x ~ !`, cast `(type)`	right
3	`* / %`	left
4	`+ -`	left
5	`<< >> >>>`	left
6	`< ⩽ > ⩾ instanceof`	left
7	`== ≠`	left
8	`&`	left
9	`^`	left
10	`|`	left
11	`&&`	left
12	`||`	left
13	`?:`	right
14	`= += -= …`	right

The rows that cause bugs:

Bitwise binds looser than comparison: `x & 1 == 1` parses as `x & (1 == 1)`, a compile error (`int & boolean`). Write `(x & 1) == 1`.

Shift binds looser than `+`: `1 << 2 + 3` is `1 << 5` = 32, not 7.

Cast binds tighter than binary operators: `(double) a / b` casts `a` only.

`&&` before `||`: `a || b && c` is `a || (b && c)`.

When in doubt, parenthesise. Reviewers never complain about clarifying parentheses.

Evaluation order

Operands are evaluated **left to right**, fully, before the operator applies — and this is guaranteed by the language, unlike C/C++. `f() + g()` calls `f` first. Method arguments are evaluated left to right. `a[i] = i++` stores into the *old* index. Predictable, but still write code that does not depend on it.

Operators Java does not have

No `**`, no `<>` for not-equal, no `->` for member access (`->` is the lambda arrow), no comma operator, no `sizeof`, no pointer `*` / `&`, no operator overloading (`+` on strings is the one built-in exception). `BigInteger` arithmetic is method calls because of this.

Interview angle

"Difference between `&` and `&&`?" `&&` short-circuits; `&` evaluates both and is also bitwise on integers.

"`>>` vs `>>>`?" Arithmetic (sign-filling) vs logical (zero-filling) right shift.

"`1 << 32` for an int?" 1 — the distance is masked to 5 bits.

"Result of `~7`?" −8.

Key takeaways

- `&&` / `||` short-circuit; `&` / `|` / `^` do not, and double as bitwise operators.
- `>>` keeps the sign, `>>>` does not; shift distances wrap at the type's width.
- Bitwise operators bind looser than comparisons — parenthesise masks.
- `?:` unboxes and promotes both branches; a `null` wrapper there throws.
- Evaluation is strictly left to right.

EXERCISE 1

Bit report

Read an integer `n`, then `n` integers. For each value `x` print one line:

```
<x> bin=<binary> ones=<count> parity=<even|odd> half=<x >> 1> logical=<x >>> 1> pow2=<yes|no>
```

where `binary` is `Integer.toBinaryString(x)`, `ones` is `Integer.bitCount(x)`, parity comes from `x & 1`, and `pow2` is `yes` when `x > 0` and `(x & (x - 1)) == 0`.

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

INPUT 1

```
3
8 7 -16
```

OUTPUT 1

```
8 bin=1000 ones=1 parity=even half=4
logical=4 pow2=yes
7 bin=111 ones=3 parity=odd half=3
logical=3 pow2=no
-16 bin=11111111111111111111111111110000
ones=28 parity=even half=-8
logical=2147483640 pow2=no
```

INPUT 2

```
2
1 0
```

OUTPUT 2

```
1 bin=1 ones=1 parity=odd half=0
logical=0 pow2=yes
0 bin=0 ones=0 parity=even half=0
logical=0 pow2=no
```

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++) {
            int x = in.nextInt();
            boolean pow2 = x > 0 && (x & (x - 1)) == 0;
            out.append(x)
               .append(" bin=").append(Integer.toBinaryString(x))
               .append(" ones=").append(Integer.bitCount(x))
               .append(" parity=").append((x & 1) == 1 ? "odd" : "even")
               .append(" half=").append(x >> 1)
               .append(" logical=").append(x >>> 1)
               .append(" pow2=").append(pow2 ? "yes" : "no")
               .append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Predict the operators

Read an integer `k` and `k` expression ids in `1..12`. For each print `<id>: <value>`, writing the expression literally in Java (no hard-coded answers). Predict each before running.

id	expression	id	expression
1	`5 / 2 * 2.0`	7	`int i = 5; i++ + ++i`
2	`7 % -3`	8	`true
3	`1 << 3	1`	9
4	`6 & 3 ^ 1`	10	`10 - 2 - 3`
5	`-7 >> 1`	11	`2 + 3 * 4 % 5`
6	`-7 >>> 28`	12	`int x = 5; x += x++ * 2; x`

Example: `3` then `1 3 7` →

```
1: 4.0
3: 9
7: 12
```

INPUT 1

```
3
1 3 7
```

OUTPUT 1

```
1: 4.0
3: 9
7: 12
```

INPUT 2

```
5
2 4 5 6 8
```

OUTPUT 2

```
2: 1
4: 3
5: -4
6: 15
8: true
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static String eval(int id) {
        switch (id) {
            case 1: return String.valueOf(5 / 2 * 2.0);
            case 2: return String.valueOf(7 % -3);
            case 3: return String.valueOf(1 << 3 | 1);
            case 4: return String.valueOf(6 & 3 ^ 1);
            case 5: return String.valueOf(-7 >> 1);
            case 6: return String.valueOf(-7 >>> 28);
            case 7: { int i = 5; return String.valueOf(i++ + ++i); }
            case 8: return String.valueOf(true || false && false);
            case 9: return String.valueOf((int) 'A' + 1);
            case 10: return String.valueOf(10 - 2 - 3);
            case 11: return String.valueOf(2 + 3 * 4 % 5);
            case 12: { int x = 5; x += x++ * 2; return String.valueOf(x); }
            default: return "?";
        }
    }

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

QUIZ

1. What is the difference between `&&` and `&` on booleans?

A. None

B. `&&` short-circuits (skips the right side when the left is false); `&` always evaluates both

C. `&` is bitwise and cannot be used on booleans

D. `&&` is slower

2. What does `-16 >>> 28` give for an `int`?

A. -1

B. 15

C. -16

D. 0

3. `x & 1 == 1` does not compile. Why?

A. `&` cannot be used on `int`

B. `==` binds tighter than `&`, so it becomes `x & (1 == 1)`, an `int & boolean`

C. `1 == 1` is always true, which is an error

D. Missing spaces

4. `int v = 1 << 32;` — what is `v`?

A. 0

B. 1

C. 4294967296

D. Compile error

5. `Integer x = null; int y = flag ? x : 0;` with `flag` true — what happens?

A. `y` is 0

B. `y` is null

C. `NullPointerException`: the ternary unboxes `x`

D. Compile error

ANSWER KEY · OPERATORS AND PRECEDENCE

1. B `a != null && a.f()` is safe because of short-circuiting; with `&`, `a.f()` runs even when `a` is null. `&` on booleans is legal and evaluates both sides.

2. B `>>>` shifts in zeros. −16 is `1111…10000`; moving it 28 places right leaves the top four ones: `1111` = 15. `>>` would give −1.

3. B Bitwise operators sit below the comparison operators in precedence. Write `(x & 1) == 1`.

4. B The shift distance is masked to 5 bits for an `int` (32 & 31 = 0), so this is `1 << 0`. For a `long` the mask is 63.

5. C One branch is a primitive, so the expression's type is `int` and the chosen `Integer` is unboxed — and unboxing null throws.

LESSON 7 · 15 MIN

Wrapper classes and autoboxing

Every primitive has an object twin — `Integer` for `int`, `Double` for `double`, `Character` for `char`, `Boolean` for `boolean`, and `Byte`, `Short`, `Long`, `Float`. They exist so that primitives can go where only objects are allowed: collections, generics, reflection, `Object` parameters. Since Java 5 the compiler converts between the two automatically (**autoboxing** and **unboxing**), which is convenient and the source of three notorious bugs. This lesson is about using wrappers deliberately.

What a wrapper is

```
Integer boxed = Integer.valueOf(42);   // an object on the heap holding an int
int raw = boxed.intValue();            // back to the primitive
```

A wrapper is an **immutable** object with one field. `Integer` has no setter; "changing" one means making a new one. This is why `Integer count = 0; count++;` allocates a new `Integer` on every increment.

The wrappers also carry the static utilities you use constantly:

```
Integer.parseInt("42")            // String → int (throws NumberFormatExceptio
Integer.valueOf("42")             // String → Integer
Integer.toString(42)              // int → String
Integer.toBinaryString(42)        // "101010"; also toHexString, toOctalStrin
Integer.MAX_VALUE, Integer.MIN_VALUE, Integer.SIZE (32), Integer.BYTES (4)
Integer.compare(a, b)             // -1 / 0 / 1, without overflow — use in co
Integer.bitCount(x), Integer.reverse(x), Integer.highestOneBit(x)
Character.isDigit(c), Character.isLetter(c), Character.toUpperCase(c), Charact
Double.isNaN(d), Double.compare(a, b), Double.MAX_VALUE, Double.MIN_VALUE
Boolean.parseBoolean("true")
```

Autoboxing and unboxing

The compiler inserts `valueOf` and `xxxValue` calls for you:

```
Integer a = 5;              // Integer.valueOf(5)
int b = a;                  // a.intValue()
List<Integer> xs = new ArrayList<>();
xs.add(7);                  // boxes
int first = xs.get(0);      // unboxes
Integer sum = a + b;        // unbox a, add, box the result
```

This happens in assignments, method arguments, return values, arithmetic, comparisons (`<`, `>` unbox), and collections. It is invisible in the source and visible in profilers.

Bug 1: `==` on wrappers

```
Integer x = 127, y = 127;
System.out.println(x == y);        // true
Integer p = 128, q = 128;
System.out.println(p == q);        // false  (!)
System.out.println(p.equals(q));   // true
```

`==` on two references compares identity. `Integer.valueOf` **caches** the values −128 to 127 (and `Boolean`, `Byte`, `Character` ≤ 127, `Short` / `Long` in the same range), returning the same object each time; outside that range it allocates a new one. So `==` "works" for small numbers and fails for large ones, which is the worst kind of bug — it passes tests and fails in production with real ids.

Rules: compare wrappers with `equals` or unbox first (`x.intValue() == y.intValue()`), or better, keep them primitive. When one side is a primitive, `==` unboxes the other and compares values correctly: `Integer x = 1000; x == 1000` is `true`.

(`new Integer(5)` is deprecated for removal precisely because it always allocated, defeating the cache; never write it.)

Bug 2: unboxing `null`

```
Map<String, Integer> counts = new HashMap<>();
int n = counts.get("missing");     // NullPointerException — get returns null,
```

Any place a wrapper is `null` and the code needs a primitive — arithmetic, comparison, a primitive parameter, a `switch`, a ternary with a primitive branch — throws `NullPointerException`. The map lookup above is the most common form. Fixes: `counts.getOrDefault("missing", 0)`, or keep an `Integer` and null-check, or `counts.merge(key, 1, Integer::sum)` for counting.

Bug 3: performance in loops

```
Long sum = 0L;                                  // wrapper!
for (long i = 0; i < 10_000_000; i++) {
    sum += i;                                   // unbox, add, box: 10 million allo
}
```

Joshua Bloch's example from *Effective Java*: this runs roughly ten times slower than with `long sum`. Prefer primitives in fields, locals and arithmetic; box only at the boundary with a collection or API. For hot collections of numbers, use `int[]` / `long[]` or a primitive-specialised library.

`equals` across wrapper types

```
Integer i = 1;
Long l = 1L;
System.out.println(i.equals(l));    // false — different classes
System.out.println(i.equals(1));    // true — 1 boxes to Integer
System.out.println(l.equals(1));    // false — 1 boxes to Integer, not Long
```

`equals` requires the same wrapper class. The last line is a real trap: `map.get(1)` on a `Map<Long, …>` returns `null` because the key `Integer 1` never equals `Long 1L`. Always match the key type: `map.get(1L)`.

`compareTo` and `Comparable`

Every wrapper implements `Comparable<Self>` — `Integer.compareTo`, `Double.compareTo` — which is what `Collections.sort` and `TreeMap` use. In a comparator, prefer the static

`Integer.compare(a, b)` over `a - b`: the subtraction overflows for large values of opposite sign and returns the wrong sign.

When wrappers are the right choice

Collections and generics — `List<Integer>`, `Map<String, Double>`: no choice.

"Absent" values — an `Integer` field that is `null` when unknown, where 0 would be a real value. (`Optional<Integer>` or `OptionalInt` are the modern alternatives.)

Reflection and frameworks that need `Object`.

Everywhere else, primitives.

`Number` and the hierarchy

`Integer`, `Long`, `Double`, `Float`, `Short`, `Byte` (and `BigInteger`, `BigDecimal`, `AtomicInteger`…) extend the abstract class `java.lang.Number`, which declares `intValue()`, `longValue()`, `doubleValue()`… — so a method taking `Number` accepts any of them and can convert. `Character` and `Boolean` are not `Number`s. All wrappers are `final` and immutable.

Interview angle

"`Integer a = 127, b = 127; a == b`*?"* True. *"With 128?"* False — the `valueOf` cache covers −128…127.

"What happens when you unbox null?" `NullPointerException`.

"`Integer.valueOf` *vs* `new Integer`*?"* `valueOf` uses the cache; the constructor always allocates and is deprecated.

"Why prefer `Integer.compare(a, b)` *to* `a - b`*?"* Overflow.

"Is `Integer` *mutable?"* No; all wrappers are immutable and final.

Key takeaways

Wrappers are immutable objects; autoboxing inserts `valueOf` / `xxxValue` calls silently.

`==` on wrappers compares identity; the cache makes it "work" for −128…127 only. Use `equals` or primitives.

Unboxing `null` throws — `map.get` into an `int` is the classic.

Boxed arithmetic in loops allocates; keep primitives in hot code.

`equals` needs the same wrapper class; `Integer 1` ≠ `Long 1L`.

EXERCISE 1

Cached or fresh?

Read an integer `n`, then `n` integers. For each value `v`, box it twice — `Integer a = v; Integer b = v;` — and print `<v> same` if `a == b` is true (the same cached object) or `<v> different` otherwise. Then print a last line `equals=<count>` with how many of the pairs were `equals` — which should be all of them.

Input: `n`, then `n` integers. **Output:** `n + 1` lines.

Example: `3` then `127 128 -128` →

```
127 same
128 different
-128 same
equals=3
```

INPUT 1

```
3
127 128 -128
```

OUTPUT 1

```
127 same
128 different
-128 same
equals=3
```

INPUT 2

```
4
0 -129 1000 5
```

OUTPUT 2

```
0 same
-129 different
1000 different
5 same
equals=4
```

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();
        int equal = 0;
        for (int i = 0; i < n; i++) {
            int v = in.nextInt();
            Integer a = v;
            Integer b = v;
            out.append(v).append(a == b ? " same" : " different").append('\n');
            if (a.equals(b)) equal++;
        }
        out.append("equals=").append(equal).append('\n');
        System.out.print(out);
    }
}
```

EXERCISE 2

Parse the votes

Read tokens until end of input. Each token should be an integer vote; parse it with `Integer.parseInt`, counting tokens that throw `NumberFormatException` as invalid. Collect the valid values in a `List<Integer>`, then print `valid=<n> invalid=<m>` and — when there is at least one valid vote — `sum=<sum> min=<min> max=<max>` computed by unboxing (a plain `int` accumulator; compare with `Integer.compare` or `Math.min/max`, never `==`).

Example: `4 -2 x 10 7.5` →

```
valid=3 invalid=2
sum=12 min=-2 max=10
```

INPUT 1

```
4 -2 x 10 7.5
```

OUTPUT 1

```
valid=3 invalid=2
sum=12 min=-2 max=10
```

INPUT 2

```
a b c
```

OUTPUT 2

```
valid=0 invalid=3
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        List<Integer> votes = new ArrayList<>();
        int invalid = 0;
        while (in.hasNext()) {
            String token = in.next();
            try {
                votes.add(Integer.parseInt(token));
            } catch (NumberFormatException e) {
                invalid++;
            }
        }
        System.out.println("valid=" + votes.size() + " invalid=" + invalid);
        if (!votes.isEmpty()) {
            int sum = 0, min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
            for (int v : votes) {        // unboxed once, here
                sum += v;
                min = Math.min(min, v);
                max = Math.max(max, v);
            }
            System.out.println("sum=" + sum + " min=" + min + " max=" + max);
        }
    }
}
```

QUIZ

1. `Integer a = 1000, b = 1000; a == b` is…

A. `true`

B. `false` — two different objects outside the −128…127 cache

C. A compile error

D. `true` only on 64-bit JVMs

2. `Map<String, Integer> m = new HashMap<>(); int n = m.get("x");` — result?

A. `n` is 0

B. `NullPointerException` when unboxing the null returned by `get`

C. Compile error

D. `n` is -1

3. Why is `Long sum = 0L; for (...) sum += i;` slow?

A. `Long` arithmetic is done in software

B. Every `+=` unboxes, adds and boxes a new `Long` object

C. `Long` cannot be a loop accumulator

D. It is not slow

4. `Integer i = 1; Long l = 1L; i.equals(l)` is…

A. `true`

B. `false` — different wrapper classes are never equal

C. Compile error

D. `true` after unboxing

5. Which comparator body is correct for two `Integer`s `a` and `b`?

A. `return a - b;`

B. `return Integer.compare(a, b);`

C. `return a == b ? 0 : 1;`

D. `return a > b;`

ANSWER KEY · WRAPPER CLASSES AND AUTOBOXING

1. B Autoboxing calls `Integer.valueOf`, which caches only −128 to 127. Larger values get fresh objects, and `==` compares references. Use `equals`.

2. B `get` returns `null` for a missing key; assigning to an `int` unboxes it and throws. `m.getOrDefault("x", 0)` is the fix.

3. B Wrappers are immutable, so each iteration allocates. With a primitive `long` the loop is a register add. Keep accumulators primitive.

4. B `equals` checks the class first. This is why `map.get(1)` misses in a `Map<Long, V>`: the key must be `1L`.

5. B `a - b` overflows for large values of opposite sign and returns the wrong sign; `Integer.compare` does not. The other two do not even return a valid ordering.

CHECKPOINT · 25 MIN

Checkpoint — Types & operators

This checkpoint covers the eight primitives, variables and constants, conversions and promotion, integer arithmetic and overflow, floating point, operators and precedence, and the wrapper classes.

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, make sure these are automatic:

- The ranges of `int` and `long`, and what `Integer.MAX_VALUE + 1` gives.
- Why `byte b = a + b` fails and `b += a` compiles.
- What `(int) 3.99`, `Math.round(3.5)` and `-7 % 3` evaluate to.
- Why `0.1 + 0.2 == 0.3`, and what to use for money.
- The difference between `&&` and `&`, `>>` and `>>>`.
- When `Integer == Integer` is true and when it is not.

The programs exercise exactly the traps above: one is about doing arithmetic in the right type, the other about reading and formatting decimals without losing cents.

EXERCISE 1

Statistics without overflow

Read an integer `n` (1 ≤ n ≤ 100 000), then `n` integers each within the `int` range. Print four lines: `sum=<exact sum>`, `min=<smallest>`, `max=<largest>`, and `avg=<mean with exactly 2 decimals>`.

The sum of 100 000 values near `Integer.MAX_VALUE` does not fit in an `int`. Choose your types accordingly, and make sure the average is computed in floating point, not truncated.

Example: `3` then `2147483647 2147483647 1` →

```
sum=4294967295
min=1
max=2147483647
avg=1431655765.00
```

INPUT 1

```
3
2147483647 2147483647 1
```

OUTPUT 1

```
sum=4294967295
min=1
max=2147483647
avg=1431655765.00
```

INPUT 2

```
4
-5 5 -5 5
```

OUTPUT 2

```
sum=0
min=-5
max=5
avg=0.00
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        String[] tokens = new String(System.in.readAllBytes()).trim().split("\\s+");
        int n = Integer.parseInt(tokens[0]);
        long sum = 0;
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
        for (int i = 1; i ⩽ n; i++) {
            int v = Integer.parseInt(tokens[i]);
            sum += v;
            if (v < min) min = v;
            if (v > max) max = v;
        }
        System.out.println("sum=" + sum);
        System.out.println("min=" + min);
        System.out.println("max=" + max);
        System.out.printf("avg=%.2f%n", (double) sum / n);
    }
}
```

EXERCISE 2

Cents, exactly

A shop stores prices as strings with exactly two decimals. Read an integer `n`, then `n` lines each holding a price and a quantity (`19.99 3`). Work entirely in **cents using `long`** — convert each price by splitting on the dot, never by parsing it as a `double` — and print:

`total=<amount>` formatted as `whole.cc` (two-digit cents),

`discount=<amount>` — 15% of the total, in cents, rounded **half up** to the nearest cent, in the same format,

`due=<total - discount>` in the same format.

Example: `2` then `19.99 3` and `0.05 1` →

```
total=60.02
discount=9.00
due=51.02
```

(9.003 rounds to 9.00; 60.02 × 0.15 = 9.003.)

INPUT 1

```
2
19.99 3
0.05 1
```

OUTPUT 1

```
total=60.02
discount=9.00
due=51.02
```

INPUT 2

```
1
0.10 3
```

OUTPUT 2

```
total=0.30
discount=0.05
due=0.25
```

REFERENCE SOLUTION

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

public class Main {
    static long cents(String price) {
        int dot = price.indexOf('.');
        long whole = Long.parseLong(price.substring(0, dot));
        long frac = Long.parseLong(price.substring(dot + 1));
        return whole * 100 + frac;
    }

    static String money(long cents) {
        return cents / 100 + "." + String.format("%02d", cents % 100);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        long total = 0;
        for (int i = 0; i < n; i++) {
            String[] parts = br.readLine().trim().split("\\s+");
            total += cents(parts[0]) * Long.parseLong(parts[1]);
        }
        // 15% = total * 15 / 100, rounded half up: add half the divisor before divi
        long discount = (total * 15 + 50) / 100;
        System.out.println("total=" + money(total));
        System.out.println("discount=" + money(discount));
        System.out.println("due=" + money(total - discount));
    }
}
```

EXERCISE 3

Binary clock

Read a number of seconds since midnight (`0 ≤ s < 86400`). Print the time as `HH:MM:SS` with `%02d` , then each component as a six-bit binary string (pad with zeros: `String.format("%6s", Integer.toBinaryString(v)).replace(' ', '0')`), then `bits set: <total ones across the three>` .

Example: `3661` →

```
01:01:01
000001 000001 000001
bits set: 3
```

INPUT 1

```
3661
```

OUTPUT 1

```
01:01:01
000001 000001 000001
bits set: 3
```

INPUT 2

```
0
```

OUTPUT 2

```
00:00:00
000000 000000 000000
bits set: 0
```

REFERENCE SOLUTION

```
import java.util.Scanner;

public class Main {
    static String six(int v) {
        return String.format("%6s", Integer.toBinaryString(v)).replace(' ', '0');
    }

    public static void main(String[] args) {
        int s = new Scanner(System.in).nextInt();
        int h = s / 3600, m = s / 60 % 60, sec = s % 60;
        System.out.printf("%02d:%02d:%02d%n", h, m, sec);
        System.out.println(six(h) + " " + six(m) + " " + six(sec));
        System.out.println("bits set: " + (Integer.bitCount(h) + Integer.bitCount(m)
    }
}
```

CHECKPOINT QUESTIONS

1. Which list contains **only** primitive types?

A. `int, Integer, char`

B. `byte, short, int, long, float, double, char, boolean`

C. `int, double, String, boolean`

D. `int, long, void`

2. `long big = 2147483648;` — does it compile?

A. Yes

B. No: the literal is an `int` and too large; it needs `L`

C. Yes, with a warning

D. Only with `var`

3. What is `(int) -3.99`?

A. -4

B. -3

C. 3

D. Compile error

4. `char c = 'x'; c++;` compiles, but `c = c + 1;` does not. Why?

A. `char` cannot be incremented

B. `c + 1` is an `int`; `c++` includes an implicit cast back to `char`

C. `c = c + 1` needs `'1'`

D. It is a compiler bug

5. `7 / 2 * 2.0` evaluates to…

A. 7.0

B. 6.0

C. 3.5

D. 6

6. How can you make integer overflow throw instead of wrapping?

A. `strictfp`

B. `Math.addExact` / `multiplyExact`

C. Compile with `-Xoverflow`

D. It always throws

7. `Math.floorMod(-1, 5)` is…

A. -1

B. 4

C. 1

D. 0

8. What is `1.0 / 0`?

A. `ArithmeticException`

B. `Infinity`

C. `NaN`

D. 0.0

9. Which comparison of two computed doubles `a` and `b` is appropriate?

A. `a == b`

B. `Math.abs(a - b) < 1e-9` (a tolerance suited to the magnitudes)

C. `a.equals(b)`

D. `(int) a == (int) b`

10. `new BigDecimal(0.1)` versus `new BigDecimal("0.1")` — which is exact 0.1?

A. Both

B. Only the `String` one; the `double` one captures 0.1000000000000000055511…

C. Only the `double` one

D. Neither

11. What is `~0`?

A. 0

B. 1

C. -1

D. `Integer.MAX_VALUE`

12. `true ? 1 : 2.0` has which type and value?

A. `int` 1

B. `double` 1.0

C. `Object` 1

D. Compile error

13. Which statement about `Integer.valueOf` is true?

A. It always allocates a new object

B. It returns cached objects for −128 to 127 and new ones otherwise

C. It caches every value

D. It is deprecated

14. `List<int> xs` — why does it fail?

A. Lists can only hold strings

B. Generics take reference types only; use `List<Integer>`

C. Missing `new`

D. `int` must be capitalised in all contexts

15. A local `int x;` is read before assignment. What happens?

A. `x` is 0

B. Compile error: variable might not have been initialized

C. Runtime `NullPointerException`

D. Undefined behaviour

ANSWER KEY · CHECKPOINT — TYPES & OPERATORS

1. B Those eight are the primitives. `Integer` and `String` are classes; `void` is not a type you can hold a value in.

2. B Integer literals are `int` unless suffixed. Even though the target is `long`, the literal itself must be valid: `2147483648L`.

3. B A floating → integral cast truncates toward zero. `Math.floor(-3.99)` would give −4.0.

4. B Binary numeric promotion makes `c + 1` an `int`. Increment, decrement and compound assignment carry an implicit narrowing cast.

5. B `/` and `*` share precedence and associate left to right: `7 / 2` is integer 3, then `3 * 2.0` is 6.0.

6. B The `*Exact` methods in `Math` (Java 8+) throw `ArithmeticException` on overflow. Plain operators never do.

7. B `floorMod` returns a result with the sign of the divisor: −1 = 5 × (−1) + 4. Plain `-1 % 5` gives −1.

8. B Floating-point division by zero follows IEEE 754: positive/zero is `Infinity`, zero/zero is `NaN`. Only integer division throws.

9. B Computed doubles carry rounding error; compare within a tolerance. `Double.compare` is for ordering, and primitives have no `equals`.

10. B The `double` constructor converts the binary value exactly, error included. Build `BigDecimal` from strings or `BigDecimal.valueOf`.

11. C `~` flips every bit: all 32 bits set is −1 in two's complement. In general `~x == -x - 1`.

12. B Both branches are numeric, so binary numeric promotion applies to the ternary: the result type is `double`, and 1 becomes 1.0.

13. B The cache is why `==` on small boxed integers is true and on large ones false. `new Integer(n)` is what is deprecated.

14. B Type parameters are erased to `Object`, so they cannot be primitives. The wrapper `Integer` is the type argument; autoboxing hides the conversion.

15. B Definite assignment is checked by the compiler for locals. Fields default to 0; locals must be assigned on every path first.

CodeKairo · Java study plan · Module 2: Types & operators. Solve the exercises online to track your progress and earn XP.