Module 7 of 20

OOP I — classes & objects

The object model, constructors and initialisation order, static members, access modifiers and immutability, and how to design a class.

Everything so far has been static methods in one class. Java is an object-oriented language, and objects — bundles of state and behaviour with their own identity — are how real programs are organised, protected and reasoned about.

This module is the object model from the ground up: what a class declares and what new creates, references and this, constructors and the exact initialisation order, static members and class initialisation, the four access levels and the discipline of encapsulation and immutability, and finally the design judgement — cohesion, composition, value versus entity, toString, builders — that turns syntax into good classes.

By the end you can write a class whose objects can never be invalid, explain initialisation order in an interview, and decide what belongs in a class and what does not.

CONTENTS

1	Classes, objects and references	14 min
2	Constructors and initialisation order	15 min
3	Static fields, methods and initialisation	13 min
4	Encapsulation, access modifiers and immutability	15 min
5	Designing a class — cohesion, composition and toString	13 min
✓	Checkpoint — Classes & objects	22 min

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

LESSON 1 · 14 MIN

Classes, objects and references

Up to now every program has been one class with static methods — Java used as a procedural language. Real Java is objects: bundles of *state* (fields) and *behaviour* (methods) that model something. This lesson is the object model itself: what a class declares, what `new` does, what a reference is, and how `this` ties them together. Everything in the next four modules stands on it.

A class is a blueprint

```
public class BankAccount {
    // state: each object has its own copy of these
    private String owner;
    private long balanceCents;

    // behaviour: operates on the state of the object it is called on
    public void deposit(long cents) {
        if (cents ⩽ 0) throw new IllegalArgumentException("deposit must be po
        balanceCents += cents;
    }

    public boolean withdraw(long cents) {
        if (cents > balanceCents) return false;
        balanceCents -= cents;
        return true;
    }

    public long balance() { return balanceCents; }
    public String owner()  { return owner; }
}
```

The class describes what every account *has* and *can do*. It creates nothing by itself. `BankAccount` is also a **type**: variables, parameters and fields can be declared as `BankAccount`.

An object is an instance

```
BankAccount a = new BankAccount();     // allocate an object on the heap, run
BankAccount b = new BankAccount();     // a second, independent object
a.deposit(500);
b.deposit(100);
a.balance()      // 500
b.balance()      // 100
```

`new` allocates memory for the fields (`owner` null, `balanceCents` 0 — the defaults), runs a constructor (next lesson), and returns a **reference** to the new object. Each object has its own fields; a method called on `a` reads and writes `a`'s fields.

References

`a` does not *contain* the account; it *points to* it. This has the same consequences as with arrays:

```
BankAccount c = a;        // alias: c and a refer to the same object
c.deposit(1);             // a.balance() is now 501
a == c                    // true — same object
a == b                    // false — different objects, even if their fields w
BankAccount d = null;     // a reference to nothing
d.balance();              // NullPointerException
```

Passing an object to a method passes the reference (Module 5); the method can call mutators on it and the caller sees the effect.

`this`

Inside an instance method, `this` is the reference to the object the method was called on. It is implicit — `balanceCents += cents` means `this.balanceCents += cents` — and explicit when needed:

- to disambiguate a field from a parameter of the same name (`this.owner = owner`);
- to pass the current object to another method (`registry.add(this)`);
- to return the current object for chaining (`return this;`);
- to call another constructor (`this(...)`, next lesson).

`this` does not exist in static methods, which is why they cannot touch instance fields.

Fields versus local variables

	Field	Local
Declared	in the class body	in a method
Lives	inside each object (or the class, if static)	in the stack frame
Default	0 / false / null	none — must assign before use
Lifetime	the object's	the block's
Access	any method of the class (and others, per modifier)	the block

The single most common design error is making something a field that should be a local: a loop counter or a temporary result stored in the object, leaking between calls. A field is for state that must *persist between method calls* and *belongs to the object*.

Instance methods and the receiver

`a.deposit(500)` — `a` is the **receiver**. The JVM passes it as a hidden first argument; inside, it is `this`. Two different accounts with the same code behave differently because their state differs. This is the essence of objects: **behaviour parameterised by state**. Compare the procedural version `deposit(account, 500)` — same idea, but the language now enforces that `deposit` belongs to accounts and can protect their fields.

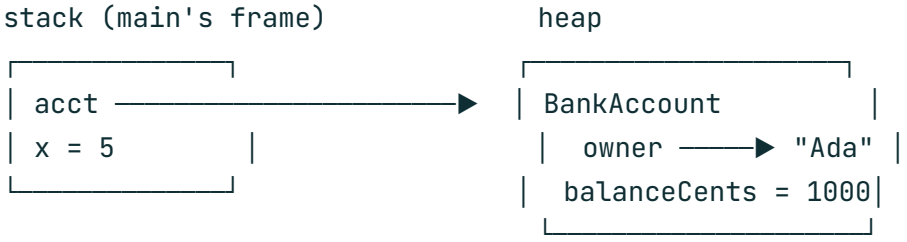
Multiple classes in one program

A program is many classes; each `.java` file holds one public class (plus optional package-private ones), and `main` lives in whichever class starts things:

```java
public class Main {
    public static void main(String[] args) {
        BankAccount acct = new BankAccount("Ada");
        acct.deposit(1000);
        System.out.println(acct.owner() + ": " + acct.balance());
    }
}
```

The exercises in this track put helper classes in the same file as `Main` without `public`; in a project they go in their own files and packages.

The object model, drawn



Locals on the stack, objects on the heap, strings also objects on the heap, arrows for references. Every question about aliasing, `==`, null and pass-by-value is answered by drawing this.

Objects everywhere you already used them

`"hello"` is a `String` object; `new Scanner(System.in)` is an object with a buffer as its state and `nextInt` as behaviour; `new StringBuilder()` is an object whose `append` mutates its state; arrays are objects. You have been calling instance methods on receivers since lesson one. What changes now is that you *write* the classes.

Naming and shape

A class is a noun (`BankAccount`, `HttpRequest`, `Matrix`); its methods are verbs or queries. Fields are `private` by default (next lessons); one concept per class; a class that needs "and"

in its description wants splitting. Keep classes small enough that the state they hold can be kept consistent — that is what the next four lessons are about.

Interview angle

"Difference between a class and an object?" Blueprint vs instance; a class is a type, an object is a value of it on the heap.

"What does `new` do?" Allocates and zero-initialises the fields, runs the constructor, returns a reference.

"What is `this`?" The receiver of the current instance method.

"Why can't a static method use `this`?" It is not called on an object.

Key takeaways

A class declares fields (state) and methods (behaviour); `new` creates an object with its own fields on the heap.

Variables hold references; `==` compares references; `null` is "no object"; aliases share one object.

`this` is the receiver; instance methods read and write the receiver's fields.

Fields persist and default; locals are per-call and must be assigned. Keep temporaries local.

EXERCISE 1

A counter with state

Write a class `Counter` with a private `int` field, and methods `increment()`, `decrement()`, `reset()` and `value()`. Read an integer `n`, then `n` commands (`inc`, `dec`, `reset`, or `print`); apply each to **one** `Counter` object and print its value whenever the command is `print`.

Example: `5` then `inc inc print dec print` →

```
2
1
```

INPUT 1

```
5
inc inc print dec print
```

OUTPUT 1

```
2
1
```

INPUT 2

```
4
dec print reset print
```

OUTPUT 2

```
-1
0
```

REFERENCE SOLUTION

```
import java.util.*;

class Counter {
    private int count;

    void increment() { count++; }
    void decrement() { count--; }
    void reset() { count = 0; }
    int value() { return count; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Counter c = new Counter();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "inc" -> c.increment();
                case "dec" -> c.decrement();
                case "reset" -> c.reset();
                default -> System.out.println(c.value());
            }
        }
    }
}
```

EXERCISE 2

Points that move

Write a class `Point` with `double x, y`, a constructor, `void translate(double dx, double dy)`, `double distanceTo(Point other)` and a `toString` returning `(x, y)` with one decimal each. Read an integer `n` and `n` commands on one point that starts at the origin: `move <dx> <dy>` translates

and prints the point; `dist <x> <y>` prints the distance to that point with two decimals. Use `Locale.ROOT`.

Example: `3` then `move 3 4`, `dist 0 0`, `move -3 0` →

```
(3.0, 4.0)
5.00
(0.0, 4.0)
```

INPUT 1

```
3
move 3 4
dist 0 0
move -3 0
```

OUTPUT 1

```
(3.0, 4.0)
5.00
(0.0, 4.0)
```

INPUT 2

```
1
dist 1 1
```

OUTPUT 2

```
1.41
```

REFERENCE SOLUTION

```java
import java.util.*;

class Point {
    double x, y;
    Point(double x, double y) { this.x = x; this.y = y; }
    void translate(double dx, double dy) { x += dx; y += dy; }
    double distanceTo(Point o) { return Math.hypot(x - o.x, y - o.y); }
    @Override public String toString() { return String.format(Locale.ROOT, "(%.1f, %
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Point p = new Point(0, 0);
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            double a = in.nextDouble(), b = in.nextDouble();
            if (cmd.equals("move")) {
                p.translate(a, b);
                System.out.println(p);
            } else {
                System.out.println(String.format(Locale.ROOT, "%.2f", p.distanceTo(r
            }
        }
    }
}
```

QUIZ

1. What does `new BankAccount()` do?

A. Declares a class

B. Allocates an object with defaulted fields, runs a constructor, and returns a reference to it

C. Copies an existing account

D. Calls `main`

2. `BankAccount a = new BankAccount(); BankAccount b = a; b.deposit(5);` — `a.balance()` is…

A. 0

B. 5 — `a` and `b` refer to the same object

C. Undefined

D. Compile error

3. Inside an instance method, `this` is…

A. The class

B. A reference to the object the method was called on

C. The first parameter

D. Always null

4. Which should be a field rather than a local?

A. A loop index

B. A temporary sum inside one method

C. The account's balance, which must persist between calls

D. A parsed input token

5. Two `BankAccount` objects with identical field values: `a == b` is…

A. `true`

B. `false` — different objects

C. `true` if both are final

D. Compile error

ANSWER KEY · CLASSES, OBJECTS AND REFERENCES

1. B `new` is allocation plus construction; the expression's value is the reference the variable stores.

2. B Assignment copies the reference. One object, two names.

3. B `this` is the implicit receiver; `balance += x` means `this.balance += x`.

4. C Fields are for state that belongs to the object and outlives a single method call. Temporaries belong in locals.

5. B `==` on references compares identity. Content equality needs an `equals` override (Module 8).

LESSON 2 · 15 MIN

Constructors and initialisation order

A constructor turns freshly allocated memory into a valid object. Get it right and an object can never exist in a broken state; get it wrong and every method must defend against half-built objects. This lesson covers constructor rules, overloading and chaining with `this(...)`, the exact order in which fields, initialiser blocks and constructor bodies run, and the validation that belongs there.

Declaring a constructor

```
public class BankAccount {
    private final String owner;
    private long balanceCents;

    public BankAccount(String owner, long openingCents) {
        if (owner == null || owner.isBlank()) throw new IllegalArgumentExcept
        if (openingCents < 0) throw new IllegalArgumentException("opening bal
        this.owner = owner;
        this.balanceCents = openingCents;
    }
}
```

A constructor has the class's name, no return type (not even `void`), and any access modifier. It runs once per `new`. It may throw — and *should* when its arguments cannot make a valid object; a constructor that silently accepts garbage moves the crash somewhere far away.

A `final` field must be assigned exactly once by the end of every constructor — the compiler checks — which makes `final` the tool for "set once, at construction, never changes".

The default constructor

If you declare **no** constructor, the compiler generates a public no-argument one that does nothing beyond field initialisers. Declare *any* constructor and the default disappears — `new BankAccount()` becomes a compile error unless you add one explicitly. This surprises people who add a parameterised constructor to a class that other code creates with `new X()`.

Overloading and chaining with `this(...)`

```
public BankAccount(String owner) {
    this(owner, 0);            // delegate to the full constructor — must be t
}
public BankAccount(String owner, long openingCents) { … the real work … }
```

Constructors overload like methods and are resolved by the same rules. `this(...)` calls another constructor of the same class and **must be the first statement**; it is the idiom for defaults, and it keeps validation in one place. Chains may be several deep but must not be circular.

`super(...)` is the sibling call to the parent's constructor (Module 8); a constructor that starts with neither gets an implicit `super()`.

Field initialisers and initialiser blocks

```
public class Counter {
    private int count = 0;                                  // field initialiser
    private final List<String> log = new ArrayList<>();  // common for collect
    private final long createdAt;

    {                                                       // instance initialise
        createdAt = System.currentTimeMillis();
    }

    public Counter() { }
}
```

Field initialisers run for every constructor, in textual order, before the constructor body. Instance initialiser blocks are the same mechanism in block form — useful only when several constructors share setup that cannot be a field initialiser (an exception-throwing computation, a loop). Most classes use field initialisers for defaults and the constructor for arguments.

The order, exactly

For `new Sub(...)` where `Sub extends Base` (the full picture is Module 8; here is the sequence):

1. Memory allocated; **all fields set to defaults** (0/false/null) — including the subclass's.
2. `Sub`'s constructor begins; its first statement is `super(...)` (explicit or implicit).
3. `Base`'s field initialisers and instance blocks run, in textual order.
4. `Base`'s constructor body runs.
5. `Sub`'s field initialisers and instance blocks run, in textual order.
6. `Sub`'s constructor body runs.

The consequence worth remembering: a field initialiser cannot read a field declared later (illegal forward reference), and a constructor that calls a method which a subclass overrides runs that override *before the subclass's fields are initialised* — they are still at their defaults. Calling overridable methods from constructors is a known trap (Module 8).

Validation and invariants

An **invariant** is a condition that is always true of a valid object — balance ≥ 0, name non-blank, start ≤ end. The constructor establishes it; every mutator preserves it. Establishing it means checking arguments and throwing `IllegalArgumentException` (or `NullPointerException` via `Objects.requireNonNull`) on bad input:

```
this.owner = Objects.requireNonNull(owner, "owner");
if (start.isAfter(end)) throw new IllegalArgumentException("start after end: '
```

Messages should say what was wrong and what the value was. A constructor that validates is the reason the rest of the class can trust its fields.

Defensive copies in constructors

```
public Schedule(List<LocalDate> dates) {
    this.dates = List.copyOf(dates);       // our own immutable copy; the cali
}
public Scores(int[] values) {
    this.values = values.clone();
}
```

Storing a mutable argument directly aliases it (Module 6). Copy it — and *validate the copy*, not the original, so the caller cannot change it between the check and the store.

Static factory methods instead of constructors

```
public static BankAccount empty(String owner) { return new BankAccount(owner,
public static Color fromHex(String hex) { … }
Integer.valueOf(5); LocalDate.of(2026, 9, 16); List.of(1, 2); Optional.empty()
```

A static method that returns an instance has advantages a constructor lacks: a **name** (`fromHex` vs `fromRgb` instead of two constructors that both take ints), the freedom to **return a cached instance** (`Integer.valueOf`), to **return a subtype**, or to return an existing object. *Effective Java* item 1. Make the constructor private when factories are the intended entry point.

Copy constructors

`public Point(Point other) { this(other.x, other.y); }` — a constructor taking an instance of the same class. Java has no built-in copy; `clone()` is a broken legacy design most codebases avoid. A copy constructor or a static `copyOf` is the idiom.

Constructors and inheritance, briefly

Constructors are **not inherited**: a subclass must declare its own (or accept the default), and each one calls a parent constructor first. Module 8.

Common mistakes

Mistake	Effect
`public void BankAccount(...)`	A method named like the class, not a constructor; `new` still uses the default
Adding a constructor and forgetting `new X()` callers	They no longer compile
`this(...)` not first	Compile error
Assigning parameter to itself (`owner = owner`)	Field stays null (Module 5)
Storing a mutable argument without copying	Caller can mutate your state
Calling an overridable method in the constructor	Runs on a partially initialised subclass

Interview angle

"What happens if you declare no constructor?" A public no-arg default is generated — until you declare any constructor.

"Can a constructor be private?" Yes: singletons, factories, utility classes.

"Order of field initialisers and constructor body?" Initialisers first (after the super constructor), then the body.

"Can a constructor return a value?" No; it has no return type. Static factories can return anything.

Key takeaways

Constructors establish invariants: validate, throw on bad arguments, copy mutable inputs.

Declaring any constructor removes the default; `this(...)` chains and must come first.

Order: defaults → super constructor → field initialisers (textual) → constructor body.

`final` fields are assigned exactly once by the end of construction.

Static factories give names, caching and flexibility a constructor cannot.

EXERCISE 1

Validated accounts

Write a class `Account` with `private final String owner` and `private long cents`, two constructors — `Account(String owner)` delegating with `this(owner, 0)` to `Account(String owner, long openingCents)` — and the validation: a blank owner or a negative opening balance throws `IllegalArgumentException`. Add `deposit(long)` (positive amounts only, otherwise throw) and `balance()`.

Read an integer `n`, then `n` lines of `open <owner> [<cents>]` (one or two arguments); for each, try to create the account and print `<owner>:<balance>`, or `rejected` if the constructor threw. Catch the exception with `try { … } catch (IllegalArgumentException e) { … }`.

Example: `3` then `open ada 500`, `open bob`, `open eve -1` →

```
ada:500
bob:0
rejected
```

INPUT 1

```
3
open ada 500
open bob
open eve -1
```

OUTPUT 1

```
ada:500
bob:0
rejected
```

INPUT 2

```
2
open x 0
open y 9999999999
```

OUTPUT 2

```
x:0
y:9999999999
```

REFERENCE SOLUTION

```java
import java.util.*;

class Account {
    private final String owner;
    private long cents;

    Account(String owner) {
        this(owner, 0);
    }

    Account(String owner, long openingCents) {
        if (owner == null || owner.isBlank()) throw new IllegalArgumentException("ow
        if (openingCents < 0) throw new IllegalArgumentException("negative opening b
        this.owner = owner;
        this.cents = openingCents;
    }

    void deposit(long amount) {
        if (amount ⩽ 0) throw new IllegalArgumentException("deposit must be positiv
        cents += amount;
    }

    long balance() { return cents; }
    String owner() { return owner; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < n; i++) {
            String[] t = in.nextLine().trim().split("\\s+");
            try {
                Account a = t.length > 2 ? new Account(t[1], Long.parseLong(t[2])) :
                System.out.println(a.owner() + ":" + a.balance());
            } catch (IllegalArgumentException e) {
                System.out.println("rejected");
            }
        }
    }
}
```

EXERCISE 2

Constructor chaining, traced

Write `class Rectangle` with three constructors: `Rectangle()` calls `this(1, 1)`; `Rectangle(double side)` calls `this(side, side)`; `Rectangle(double w, double h)` assigns the fields. Each constructor prints `ctor(<its parameter list>)` **after** its `this(...)` call returns, e.g. `ctor()`, `ctor(side=3.0)`, `ctor(w=2.0,h=5.0)`. Read an integer `n` and `n` lines `<k> <values>` with `k` 0, 1 or 2; construct with the matching constructor and then print `area=<w*h>`.

Example: `2` then `0`, `1 3` →

```
ctor(w=1.0,h=1.0)
ctor()
area=1.0
ctor(w=3.0,h=3.0)
ctor(side=3.0)
area=9.0
```

INPUT 1

```
2
0
1 3
```

OUTPUT 1

```
ctor(w=1.0,h=1.0)
ctor()
area=1.0
ctor(w=3.0,h=3.0)
ctor(side=3.0)
area=9.0
```

INPUT 2

```
1
2 2 5
```

OUTPUT 2

```
ctor(w=2.0,h=5.0)
area=10.0
```

REFERENCE SOLUTION

```java
import java.util.*;

class Rectangle {
    double w, h;
    Rectangle() {
        this(1, 1);
        System.out.println("ctor()");
    }
    Rectangle(double side) {
        this(side, side);
        System.out.println("ctor(side=" + side + ")");
    }
    Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
        System.out.println("ctor(w=" + w + ",h=" + h + ")");
    }
    double area() { return w * h; }
}

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 k = in.nextInt();
            Rectangle r = k == 0 ? new Rectangle() : k == 1 ? new Rectangle(in.nextD
            System.out.println("area=" + r.area());
        }
    }
}
```

QUIZ

1. A class declares one constructor `Point(int x, int y)`. `new Point()` …

A. Works, using defaults

B. Is a compile error — declaring a constructor removes the default one

C. Throws at run time

D. Calls `Point(0, 0)`

2. Where must `this(...)` appear in a constructor?

A. Anywhere

B. As the first statement

C. Last

D. In a static block

3. Order of execution for `new Sub()` (with a superclass `Base`)?

A. Sub body, Base body, initialisers

B. Field defaults → Base initialisers → Base body → Sub initialisers → Sub body

C. Sub initialisers → Base body → Sub body

D. Random

4. `public void Point(int x) { … }` inside `class Point` is…

A. A constructor

B. An ordinary method that happens to share the class name — a classic mistake

C. A compile error

D. A static factory

5. Why prefer a static factory like `Color.fromHex("#fff")` over a constructor?

A. Factories are faster

B. It has a descriptive name and may return cached instances or subtypes

C. Constructors cannot take strings

D. Factories skip validation

ANSWER KEY · CONSTRUCTORS AND INITIALISATION ORDER

1. B The compiler generates a no-arg constructor only when none is declared. Add one explicitly if you need it.

2. B Constructor chaining must happen before any other statement so the object is initialised by exactly one path.

3. B Superclass first; within each class, field initialisers in textual order, then the constructor body.

4. B Constructors have no return type. With `void` it is just a method; `new Point(1)` will not call it.

5. B Names distinguish overloads with the same parameter types, and a factory need not create a new object every time.

LESSON 3 · 13 MIN

Static fields, methods and initialisation

`static` marks a member as belonging to the **class** rather than to any object: one copy for the whole program, reachable without an instance. Used well it gives you constants, counters, utility methods and factories. Used carelessly it gives you global mutable state, the source of some of the hardest bugs in Java programs. This lesson covers what `static` means at run time, when to use it, and the class-initialisation sequence behind it.

Static fields

```
public class Ticket {
    private static int nextId = 1;            // one for the class
    private final int id;                     // one per ticket

    public Ticket() {
        id = nextId++;                        // every ticket takes the next n
    }
    public static int issued() { return nextId - 1; }
}
```

There is exactly one `nextId` however many tickets exist — or none. It is created when the class is loaded and lives until the JVM exits. Access it as `Ticket.issued()` (through the class); accessing a static through an instance (`t.issued()`) compiles with a warning and misleads readers.

Legitimate static fields:

- **Constants**: `public static final int MAX = 100;` `private static final Pattern EMAIL = Pattern.compile(…);` — `static` so there is one, `final` so it never changes.
- **Caches and registries** shared by design (with care for threads — Module 17).
- **Counters** like `nextId` — and again, thread-unsafe as written; `AtomicInteger` fixes that.
- **Loggers**: `private static final Logger log = …`.

Illegitimate: anything that is really per-object state, or "global variables" reached from everywhere because passing them was inconvenient.

Static methods

```
public static int max(int a, int b) { return a > b ? a : b; }
public static Ticket parse(String text) { … return new Ticket(…); }
```

A static method has no `this`. It can use its parameters, static fields and other static methods; it cannot use instance members without an object. It is the right choice for:

Pure functions of their arguments: `Math.abs`, `Integer.parseInt`, `Character.isDigit`, your `gcd(a, b)`.

Factories: `LocalDate.of`, `List.of`, `Ticket.parse`.

`main` — nothing exists yet when it starts.

If a method uses no instance state, IDEs suggest making it static; that suggestion is usually right. If it *does* use instance state, it must not be static, however tempting for calling convenience.

Static methods are **not polymorphic**: a subclass can declare a static method with the same signature, but that *hides* the parent's rather than overriding it, and the call is resolved by the reference's static type. Module 8. (`@Override` on a static method is an error.)

Static initialiser blocks

```
public class Config {
    private static final Map<String, String> DEFAULTS;
    static {
        Map<String, String> m = new HashMap<>();
        m.put("host", "localhost");
        m.put("port", "8080");
        DEFAULTS = Collections.unmodifiableMap(m);    // final static assigned
    }
}
```

A `static { … }` block runs once, when the class is initialised, in textual order with static field initialisers. Use it when a static value needs more than an expression to build (loops, try/catch). `Map.of(…)` and `List.of(…)` make many such blocks unnecessary today.

When a class is initialised

The JVM initialises a class **lazily**, the first time it is *actively used*: an instance is created, a static method is called, a non-constant static field is read or written, or a subclass is initialised. Loading a class (`Class.forName` without initialisation, or just referring to `Ticket.class`) does not trigger it; reading a compile-time constant (`Ticket.MAX`) does not either — the constant was inlined.

Initialisation runs the static initialisers and blocks top to bottom, exactly once, and is thread-safe (the JVM locks it). Superclasses first. If it throws, the class is marked erroneous and every later use throws `NoClassDefFoundError` — the "worked on the first request, broken forever after" symptom.

A static field initialised from another static field declared **below** it sees the default value:

```
static int a = b + 1;    // b is still 0 here
static int b = 5;        // a is 1, not 6
```

The compiler allows this when the read goes through a method or a qualified name; declare in dependency order and the problem disappears.

Utility classes

A class of only static methods — `Math`, `Arrays`, `Collections`, your `StringUtils`:

```
public final class Geometry {
    private Geometry() { }                    // no instances, ever
    public static double area(double r) { return Math.PI * r * r; }
}
```

`final` and a private constructor say "do not instantiate or extend". Static imports (`import static Geometry.area`) make the calls read naturally.

Singletons

A class with one instance, reached through a static accessor:

```
public final class Registry {
    private static final Registry INSTANCE = new Registry();
    private Registry() { }
    public static Registry get() { return INSTANCE; }
}
```

Initialised on first use by the class-initialisation lock, so it is thread-safe without code. An `enum` with one constant is the idiom *Effective Java* prefers (serialization-safe, reflection-safe). Singletons are global state in a costume; frameworks with dependency injection largely replaced them.

Static nested classes

`static class Node { … }` inside another class is a class that merely lives in its namespace and has no reference to an outer instance — Module 11. Very common for helper types (`Map.Entry`, list nodes).

The costs of static state

Hidden dependencies: a method that reads a static field depends on something not in its signature.

Testing: static state persists between tests; resetting it is manual and fragile.

Concurrency: shared without saying so; every thread sees the same field.

Lifetime: a static reference keeps its object alive until the class is unloaded — static collections are the classic Java memory leak (Module 16).

The default is instance state passed explicitly; reach for `static` for constants, pure functions and factories, and be deliberate about anything mutable.

Interview angle

"Can a static method be overridden?" No — hidden, resolved statically.

"When does a static block run?" Once, at class initialisation, on first active use.

"Can a static method access `this`?" No.

"What is a static nested class?" A nested class without an outer-instance reference.

"How do you implement a thread-safe singleton?" A `static final` field initialised at class init, or an enum.

Key takeaways

`static` = one per class, alive for the program; access through the class name.

Static methods have no `this`: for pure functions, factories and `main`; they hide rather than override.

Static initialisers run once, lazily, in textual order, under a lock; failures poison the class.

Constants, loggers, factories: yes. Mutable global state: only deliberately, with thread safety in mind.

EXERCISE 1

Numbered tickets

Write a class `Ticket` with a `private static int nextId = 1`, a `private final int id` assigned from it in the constructor, a `private final String holder`, and `static int issued()` returning how many tickets exist. Read an integer `n` and `n` names; create a ticket for each and print `#<id> <holder>`; finally print `issued=<count>`.

Example: `2` then `ada bob` →

```
#1 ada
#2 bob
issued=2
```

INPUT 1

```
2
ada bob
```

OUTPUT 1

```
#1 ada
#2 bob
issued=2
```

INPUT 2

```
0
```

OUTPUT 2

```
issued=0
```

REFERENCE SOLUTION

```java
import java.util.*;

class Ticket {
    private static int nextId = 1;
    private final int id;
    private final String holder;

    Ticket(String holder) {
        this.id = nextId++;
        this.holder = holder;
    }

    static int issued() { return nextId - 1; }

    @Override
    public String toString() { return "#" + id + " " + holder; }
}

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++) {
            Ticket t = new Ticket(in.next());
            System.out.println(t);
        }
        System.out.println("issued=" + Ticket.issued());
    }
}
```

EXERCISE 2

Instance census

Write `class Sensor` with a `private static int created` counter incremented in the constructor, an instance field `name`, a static factory `static Sensor of(String name)`, and `static int created()`. Read an integer `n` and `n` names; create each through the factory and print `created <name> (#<created so far>)`. Then print `total=<Sensor.created()>` and `sameCounter=<whether two different instances report the same created() through the class>` — always `true`, because the counter belongs to the class, not to any instance.

Example: `2` then `temp humidity` →

```
created temp (#1)
created humidity (#2)
total=2
sameCounter=true
```

INPUT 1

```
2
temp humidity
```

OUTPUT 1

```
created temp (#1)
created humidity (#2)
total=2
sameCounter=true
```

INPUT 2

```
0
```

OUTPUT 2

```
total=0
sameCounter=true
```

REFERENCE SOLUTION

```java
import java.util.*;

class Sensor {
    private static int created = 0;
    final String name;
    private Sensor(String name) {
        this.name = name;
        created++;
    }
    static Sensor of(String name) { return new Sensor(name); }
    static int created() { return created; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Sensor> sensors = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            Sensor s = Sensor.of(in.next());
            sensors.add(s);
            System.out.println("created " + s.name + " (#" + Sensor.created() + ")");
        }
        System.out.println("total=" + Sensor.created());
        boolean same = sensors.size() < 2 || Sensor.created() == Sensor.created();
        System.out.println("sameCounter=" + same);
    }
}
```

QUIZ

1. How many copies of a `static` field exist?

A. One per object

B. One per class, however many objects exist

C. One per thread

D. One per method call

2. When does a `static { … }` block run?

A. On every `new`

B. Once, when the class is initialised on first active use

C. At compile time

D. When `main` starts, for all classes

3. A subclass declares a static method with the same signature as its parent's. This is…

A. Overriding

B. Hiding — resolved by the reference's static type, not polymorphic

C. A compile error

D. Overloading

4. Which is a good use of a static field?

A. An account's balance

B. A `static final Pattern` compiled once for the class

C. A per-request user id

D. A loop counter

5. `static int a = b + 1; static int b = 5;` — the value of `a` is…

A. 6

B. 1 — `b` is still 0 when `a`'s initialiser runs

C. 5

D. Compile error

ANSWER KEY · STATIC FIELDS, METHODS AND INITIALISATION

1. B Static members belong to the class and live from class initialisation until the JVM exits.

2. B Class initialisation is lazy and happens exactly once, under a JVM lock, superclasses first.

3. B Static methods are not dispatched on the receiver's runtime class. `@Override` on a static method is rejected.

4. B Constants, loggers and shared immutable helpers are what `static` is for; per-object state is not.

5. B Static initialisers run top to bottom; `b` has only its default at that point. Declare in dependency order.

LESSON 4 · 15 MIN

Encapsulation, access modifiers and immutability

Encapsulation is the practice of keeping an object's state private and letting it change only through methods that keep it valid. It is what makes a class more than a struct: the class, not its callers, is responsible for its invariants. This lesson covers the four access levels, the getter/setter convention and its limits, and the strongest form of encapsulation — immutability — which Java rewards with thread safety and simplicity.

The four access levels

Modifier	Same class	Same package	Subclass (other package)	Everyone
`private`	✓			
(none — package-private)	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Rules of thumb:

- **Fields: `private`.** Always, unless it is a `public static final` constant. A public mutable field is a value nobody can validate, log, or change the representation of.
- **Methods: as narrow as works.** `public` for the API, `private` for helpers, package-private for collaboration within a package, `protected` only when subclasses genuinely need a hook.
- **Classes: `public` for the API, package-private for internals.** A top-level class can only be `public` or package-private; nested classes may use all four.

Access is checked at compile time *and* by the JVM, so a private field really is private — barring reflection, which the module system (Module 19) can also close.

`protected` is more open than it sounds: it includes the whole package, not just subclasses. And `private` is per-**class**, not per-object: `other.balance` inside `BankAccount` is legal for

another `BankAccount` — which is how `equals` can compare fields.

Getters and setters

```
public class Person {
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = Objects.requireNonNull(name
    public int getAge() { return age; }
    public void setAge(int age) {
        if (age < 0 || age > 150) throw new IllegalArgumentException("age " +
        this.age = age;
    }
    public boolean isAdult() { return age ⩾ 18; }     // boolean getters use
}
```

The `getX` / `setX` / `isX` names are the **JavaBeans** convention, and many frameworks (serialization, templating, ORM) find properties by it, so it is worth following when a class is a data holder. But a setter for every field is *not* encapsulation — it is a public field with extra steps. Ask, for each field: does it change after construction? Who should be allowed to change it? Under what checks? Often the answer is a **behavioural** method (`deposit`, `rename`, `markPaid`) rather than a setter, and often the field is simply `final`.

A getter that returns a mutable field (a `List`, an array, a `Date`) leaks the state; return a copy or an unmodifiable view (Module 6, Module 14).

Modern Java offers `record` (Module 10) for pure data carriers — accessors without the `get` prefix, no setters, generated `equals` / `hashCode` / `toString`.

Invariants are the point

```
public final class Range {
    private final int lo, hi;          // invariant: lo ⩽ hi

    public Range(int lo, int hi) {
        if (lo > hi) throw new IllegalArgumentException(lo + " > " + hi);
        this.lo = lo; this.hi = hi;
    }
    public int length() { return hi - lo; }
    public boolean contains(int x) { return lo ⩽ x && x < hi; }
    public Range shift(int by) { return new Range(lo + by, hi + by); }    // r
}
```

Because the fields are private and final and the only way in is a validating constructor, `length()` can never be negative and `contains` never has to check `lo ⩽ hi`. The invariant is established once and holds forever. That is the payoff of encapsulation: the rest of the code trusts the object.

Immutability

An object is **immutable** if its state cannot change after construction. Recipe (*Effective Java* item 17):

- No mutators — no setters, no methods that change fields.
- All fields `private final`.
- The class `final` (or all constructors private with static factories), so no subclass can add mutation.
- Mutable components are never shared: defensive copies in and out (`List.copyOf`, `clone`).
- "Modifying" operations return a **new** instance (`shift` above, `String.toUpperCase`, `BigDecimal.add`, `LocalDate.plusDays`).

Why bother:

- **Thread safety for free** — nothing to synchronise.
- **Safe sharing** — one instance can be handed to anyone, cached, used as a map key.
- **Simplicity** — an immutable object has exactly one state to reason about; no "what if it changed between these two lines".
- **Failure atomicity** — a method that throws cannot leave the object half-modified.

The cost is allocation for every "change" — usually negligible, occasionally significant (a `String` built by `+=` — hence `StringBuilder`, the mutable companion). Design value-like types (money, points, dates, ranges, ids) immutable by default; use mutability for entities with identity and lifecycle (an account, a session, a connection).

`final` on a field is necessary but not sufficient: `private final List<String> items` can still be `items.add(x)`-ed. Immutability is about the reachable object graph, not the reference.

Package-private and the "friend" pattern

Java has no C++ `friend`. Classes that must share internals live in the same package and use package-private members; the package is the unit of encapsulation above the class. Keep such packages small and the shared members few.

Tell, don't ask

```
// asking: pull the state out and decide outside
if (account.getBalance() ⩾ amount) account.setBalance(account.getBalance() -
// telling: let the object do it, keeping the invariant inside
boolean ok = account.withdraw(amount);
```

The first form spreads the rule ("balance must not go negative") across every caller; the second keeps it in one place. When you find yourself writing `get`-compute-`set`, move the computation into the class.

Interview angle

"What is encapsulation?" Private state, controlled access, invariants maintained by the class.

"Difference between `protected` and package-private?" `protected` adds subclasses in other packages to package access.

"How do you make a class immutable?" No setters, private final fields, final class, defensive copies, return new instances.

"Why are immutable objects thread-safe?" No state changes, so no races.

"Is a class with getters and setters for every field encapsulated?" Not meaningfully.

Key takeaways

- Fields private; methods as narrow as possible; `protected` includes the package.
- Getters/setters are a convention, not a design: prefer behavioural methods and `final` fields; never leak mutable internals.
- Constructors establish invariants; mutators preserve them; callers never see a broken object.
- Immutable by default for value types: private final fields, final class, copies in and out, new instances for changes.

EXERCISE 1

An immutable range

Write a `final` class `Range` with `private final int lo, hi` (invariant `lo ⩽ hi`, else the constructor throws `IllegalArgumentException`), and methods `length()` (`hi - lo`), `contains(int x)` (`lo ⩽ x < hi`), `shift(int by)` returning a **new** `Range`, and `toString()` giving `[lo, hi)`.

Read an integer `n`, then `n` commands: `new <lo> <hi>` (replace the current range, or print `invalid` and keep the old one), `shift <by>`, `contains <x>` (print `true` / `false`), `len` (print the length), `print`.

Example: `5` then `new 2 5`, `print`, `shift 3`, `print`, `contains 7` →

```
[2, 5)
[5, 8)
true
```

INPUT 1

```
5
new 2 5
print
shift 3
print
contains 7
```

OUTPUT 1

```
[2, 5)
[5, 8)
true
```

INPUT 2

```
4
new 5 2
print
new 1 1
len
```

OUTPUT 2

```
invalid
[0, 0)
0
```

REFERENCE SOLUTION

```
import java.util.*;

final class Range {
    private final int lo, hi;

    Range(int lo, int hi) {
        if (lo > hi) throw new IllegalArgumentException(lo + " > " + hi);
        this.lo = lo;
        this.hi = hi;
    }

    int length() { return hi - lo; }
    boolean contains(int x) { return lo ⩽ x && x < hi; }
    Range shift(int by) { return new Range(lo + by, hi + by); }

    @Override
    public String toString() { return "[" + lo + ", " + hi + ")"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Range r = new Range(0, 0);
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "new" -> {
                    int lo = in.nextInt(), hi = in.nextInt();
                    try {
                        r = new Range(lo, hi);
                    } catch (IllegalArgumentException e) {
                        System.out.println("invalid");
                    }
                }
                case "shift" -> r = r.shift(in.nextInt());
                case "contains" -> System.out.println(r.contains(in.nextInt()));
                case "len" -> System.out.println(r.length());
                default -> System.out.println(r);
            }
        }
    }
}
```

EXERCISE 2

A thermostat with invariants

Write `class Thermostat` with a private `target` (degrees) and private constants `MIN = 5`, `MAX = 30`. `setTarget(int)` rejects values outside `[MIN, MAX]` by throwing `IllegalArgumentException("target out of range: <v>")` and leaves the field unchanged; `getTarget()` returns it; `adjust(int delta)` moves the target but **clamps** to the range. Read an integer `n` and `n` commands `set <v>` or `adjust <d>`; print `target=<v>` after each, or the exception message when `set` rejects. The target starts at 20.

Example: `3` then `set 25`, `set 40`, `adjust 10` →

```
target=25
target out of range: 40
target=30
```

INPUT 1

```
3
set 25
set 40
adjust 10
```

OUTPUT 1

```
target=25
target out of range: 40
target=30
```

INPUT 2

```
2
adjust -100
set 5
```

OUTPUT 2

```
target=5
target=5
```

REFERENCE SOLUTION

```java
import java.util.*;

class Thermostat {
    private static final int MIN = 5, MAX = 30;
    private int target = 20;
    void setTarget(int v) {
        if (v < MIN || v > MAX) throw new IllegalArgumentException("target out of ra
        target = v;
    }
    int getTarget() { return target; }
    void adjust(int delta) { target = Math.max(MIN, Math.min(MAX, target + delta));
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Thermostat t = new Thermostat();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            int v = in.nextInt();
            try {
                if (cmd.equals("set")) t.setTarget(v); else t.adjust(v);
                System.out.println("target=" + t.getTarget());
            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

QUIZ

1. Which members can a class in **another package** that is **not** a subclass access?

A. `public` and `protected`

B. `public` only

C. Everything but `private`

D. Package-private and `public`

2. `private` restricts access to…

A. The same object only

B. The same class — other instances of that class may access it too

C. The same package

D. Subclasses

3. Which is **not** required for an immutable class?

A. No setters

B. Private final fields

C. Getters for every field

D. Defensive copies of mutable components

4. A getter returns the internal `List<String>` directly. The consequence is…

A. None

B. Callers can mutate the object's state through the returned list

C. A compile error

D. The list is copied automatically

5. "Tell, don't ask" suggests replacing `if (a.getBalance() ⩾ x) a.setBalance(a.getBalance() - x)` with…

A. More getters

B. `a.withdraw(x)` — keeping the rule inside the class

C. A static method

D. A public field

ANSWER KEY · ENCAPSULATION, ACCESS MODIFIERS AND IMMUTABILITY

1. B Package-private and `protected` both require the same package (or, for `protected`, a subclass). Only `public` crosses all boundaries.

2. B Access is per class, not per object, which is what lets `equals(Other o)` read `o.field`.

3. C Immutability is about the absence of mutation; accessors are optional. Also mark the class `final` and return new instances for changes.

4. B Returning a mutable internal object leaks it. Return a copy or `Collections.unmodifiableList(list)`.

5. B The invariant (no overdraft) lives in one place instead of in every caller.

LESSON 5 · 13 MIN

Designing a class — cohesion, composition and toString

Knowing the syntax of classes is not the same as knowing what to put in one. This lesson is about the design choices: what belongs together, how objects hold other objects (composition), values versus entities, the first `Object` method you override (`toString`), and the object lifecycle from `new` to garbage collection. It also names the builder, the pattern for objects with many optional parts.

Cohesion: one reason to change

A class should model one concept, and everything in it should be about that concept. `Order` holds lines, a customer reference and a status, and knows how to compute its total and whether it can be cancelled. It does not know how to format itself as HTML, send email, or talk to a database — those are other classes' jobs. The test: describe the class in one sentence without "and". A class that keeps growing methods for unrelated callers is two classes.

The mirror rule — **don't repeat yourself** — says the *same* concept should live in one place. If three classes each compute a shipping cost, there is a `ShippingPolicy` waiting to exist.

Composition: objects made of objects

```
public class Order {
    private final Customer customer;                // has-a
    private final List<OrderLine> lines = new ArrayList<>();
    private Status status = Status.OPEN;

    public void add(Product p, int qty) { lines.add(new OrderLine(p, qty)); }
    public long totalCents() {
        long sum = 0;
        for (OrderLine l : lines) sum += l.subtotalCents();
        return sum;
    }
}
```

An `Order` *has* a customer and lines; it delegates the subtotal of each line to the `OrderLine`. Composition is the primary way to build larger behaviour from smaller pieces, and it is what "favour composition over inheritance" (Module 8) points at: a `Stack` that *has* a list is more robust than one that *is* a list.

Wiring: pass collaborators in through the constructor (`new Order(customer)`), not by creating them inside from global state — that keeps the class testable and its dependencies visible.

Values versus entities

	Value object	Entity
Identity	by content — two `Money(5, "EUR")` are the same	by id — two customers named "Ada" are different
Mutability	immutable	usually mutable, with lifecycle
`equals` / `hashCode`	on all fields	on the id
Examples	`Money`, `Point`, `LocalDate`, `Range`, `Email`	`Customer`, `Order`, `Session`, `Connection`

Deciding which you are writing settles most other questions: values get `final` fields, no setters and full-field equality (records do all of this — Module 10); entities get an id, behavioural methods and equality on the id.

`toString`

Every class inherits `Object.toString()`, which returns `ClassName@1b6d3586` — the class and a hash. Override it the moment a class exists, because it is what appears in `println`, string concatenation, debuggers and logs:

```
@Override
public String toString() {
    return "Order[" + id + ", " + customer.name() + ", " + lines.size() + " li
}
```

Conventions: include the fields that identify and describe the object; keep it one line; do not include secrets (passwords, tokens); do not make callers parse it — provide accessors for data. `@Override` is mandatory practice: it makes a typo (`toSting`) a compile error instead of a silent new method. `equals` and `hashCode`, the other two you will override, are Module 8's subject.

The object lifecycle

`new`: memory allocated on the heap, fields defaulted, constructor run.

Use: references passed around; the object lives while any reference reaches it from a *GC root* (a local on some stack, a static field, a thread).

Unreachable: the last reference is dropped — a local goes out of scope, a field is nulled, a collection removes it.

Collected: the garbage collector reclaims the memory at a time of its choosing. You cannot force it (`System.gc()` is a hint) and you cannot observe it reliably.

There are **no destructors**. `finalize()` existed and is deprecated for removal — it ran at an unpredictable time, once, and could resurrect objects. For resources that must be released deterministically (files, sockets, locks), the object implements `AutoCloseable` and callers use `try`-with-resources (Module 12). Memory is the GC's job; everything else is `close()`'s.

Builders for many-part objects

When a constructor would need six parameters, four of them optional, a telescoping set of constructors becomes unreadable and a "setter everything" object breaks immutability. The **builder** pattern separates assembly from the immutable result:

```
Pizza p = new Pizza.Builder(Size.LARGE)
    .cheese()
    .topping("olives")
    .topping("basil")
    .build();
```

`Builder` is a static nested class with a fluent setter per option returning `this`, and a `build()` that validates and calls the private `Pizza` constructor. `StringBuilder` and `HttpRequest.newBuilder()` are the pattern in the JDK. Use it at four-plus optional parameters; below that, overloaded constructors or static factories suffice.

Class layout conventions

Order inside a class, top to bottom: constants → static fields → instance fields → constructors → static factories → public methods → private helpers → `equals` / `hashCode` / `toString`. One top-level public class per file, named for it. Keep fields at the top so a reader learns the state before the behaviour.

A checklist for a new class

- What concept is this? One sentence, no "and".
- Value or entity? Decide equality and mutability accordingly.
- What are the invariants? Enforce them in the constructor and every mutator.
- Which fields are `final`? Ideally all, for a value.
- What is `private`? Everything but the API.
- Does it hold mutable objects? Copy in, copy out, or make them immutable.
- `toString` written? `@Override` on it?
- Could a caller ever see a half-built or inconsistent object? Then fix the constructor.

Interview angle

- *"Composition vs inheritance?"* Has-a vs is-a; prefer has-a — it hides the inner object's API and can change at run time.
- *"How does Java free memory?"* Garbage collection of unreachable objects; no destructors; `close()` for resources.
- *"Why override `toString`?"* Debuggability: default output is `Class@hash`.
- *"What is the builder pattern for?"* Constructing immutable objects with many optional parameters readably.

Key takeaways

- One concept per class; compose larger behaviour from smaller objects passed in through constructors.
- Decide value vs entity first — it settles equality, mutability and design.

Override `toString` immediately, with `@Override`; keep secrets out of it.

Objects live while reachable and die when the GC says; resources are closed explicitly, never finalised.

Builders for many optional parts; static factories and overloads for few.

EXERCISE 1

Orders made of lines

Write two classes. `OrderLine` holds a product name, a quantity and a unit price in cents, with `subtotalCents()`. `Order` holds an id and a list of lines, with `add(String name, int qty, long unitCents)`, `totalCents()` and a `toString()` of the form `Order#<id>[<n> lines, total=<cents>]`.

Read an integer `id`, an integer `n`, then `n` lines `<name> <qty> <unitCents>`. Add each line and, at the end, print the order via `toString`, then one line per order line as `<name> x<qty> = <subtotal>`.

Example: `42`, `2`, `pen 3 150`, `pad 1 499` →

```
Order#42[2 lines, total=949]
pen x3 = 450
pad x1 = 499
```

INPUT 1

```
42
2
pen 3 150
pad 1 499
```

OUTPUT 1

```
Order#42[2 lines, total=949]
pen x3 = 450
pad x1 = 499
```

INPUT 2

```
7
0
```

OUTPUT 2

```
Order#7[0 lines, total=0]
```

REFERENCE SOLUTION

```
import java.util.*;

class OrderLine {
    private final String name;
    private final int qty;
    private final long unitCents;

    OrderLine(String name, int qty, long unitCents) {
        this.name = name;
        this.qty = qty;
        this.unitCents = unitCents;
    }

    long subtotalCents() { return qty * unitCents; }
    String name() { return name; }
    int qty() { return qty; }
}

class Order {
    private final int id;
    private final List<OrderLine> lines = new ArrayList<>();

    Order(int id) { this.id = id; }

    void add(String name, int qty, long unitCents) {
        lines.add(new OrderLine(name, qty, unitCents));
    }

    long totalCents() {
        long sum = 0;
        for (OrderLine l : lines) sum += l.subtotalCents();
        return sum;
    }

    List<OrderLine> lines() { return Collections.unmodifiableList(lines); }

    @Override
    public String toString() {
        return "Order#" + id + "[" + lines.size() + " lines, total=" + totalCents()
    }
}

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

```
            order.add(in.next(), in.nextInt(), in.nextLong());
        }
        System.out.println(order);
        for (OrderLine l : order.lines()) {
            System.out.println(l.name() + " x" + l.qty() + " = " + l.subtotalCents()
        }
    }
}
```

EXERCISE 2

A shopping cart

Design `class Cart` composed of `Item` objects (`name`, `unitCents`, `qty`). `add(name, unitCents, qty)` merges quantities when the name already exists; `remove(name)` deletes; `totalCents()` sums; `toString` lists items in insertion order as `<name> x<qty> = <total in dollars, two decimals>` per line followed by `TOTAL <dollars>`. Read an integer `n` and `n` commands (`add <name> <cents> <qty>`, `remove <name>`), then print the cart.

Example: `3` then `add pen 150 2`, `add pad 400 1`, `add pen 150 1` →

```
pen x3 = 4.50
pad x1 = 4.00
TOTAL 8.50
```

INPUT 1

```
3
add pen 150 2
add pad 400 1
add pen 150 1
```

OUTPUT 1

```
pen x3 = 4.50
pad x1 = 4.00
TOTAL 8.50
```

INPUT 2

```
2
add a 100 1
remove a
```

OUTPUT 2

```
TOTAL 0.00
```

REFERENCE SOLUTION

```
import java.util.*;

class Item {
    final String name;
    final int unitCents;
    int qty;
    Item(String name, int unitCents, int qty) { this.name = name; this.unitCents = u
}

class Cart {
    private final Map<String, Item> items = new LinkedHashMap<>();
    void add(String name, int unitCents, int qty) {
        Item existing = items.get(name);
        if (existing != null) existing.qty += qty;
        else items.put(name, new Item(name, unitCents, qty));
    }
    void remove(String name) { items.remove(name); }
    long totalCents() {
        long t = 0;
        for (Item it : items.values()) t += (long) it.unitCents * it.qty;
        return t;
    }
    @Override public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Item it : items.values()) {
            sb.append(String.format(Locale.ROOT, "%s x%d = %.2f%n", it.name, it.qty,
        }
        sb.append(String.format(Locale.ROOT, "TOTAL %.2f%n", totalCents() / 100.0));
        return sb.toString();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Cart cart = new Cart();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            if (cmd.equals("add")) cart.add(in.next(), in.nextInt(), in.nextInt());
            else cart.remove(in.next());
        }
        System.out.print(cart);
    }
}
```

QUIZ

1. "Favour composition over inheritance" means…

A. Never write subclasses

B. Build behaviour by holding other objects (has-a) rather than extending them (is-a) when a has-a relationship fits

C. Use interfaces only

D. Avoid constructors

2. The default `Object.toString()` returns…

A. The field values

B. `ClassName@hexHash`

C. An empty string

D. `null`

3. A value object's `equals` should compare…

A. References

B. All its fields

C. Only an id

D. Nothing — values are never equal

4. How is an object's memory freed in Java?

A. By calling its destructor

B. By `delete`

C. By the garbage collector once nothing references it

D. By `finalize()`, which you must call

5. The builder pattern is most useful when…

A. A class has one field

B. A class has many optional parameters and should end up immutable

C. You need a singleton

D. Performance is critical

ANSWER KEY · DESIGNING A CLASS — COHESION, COMPOSITION AND TOSTRING

1. B Composition hides the inner object's API, can be swapped at run time, and does not break when the inner class changes.

2. B Override it in every class you debug or log; annotate the override with `@Override` so a typo becomes a compile error.

3. B Values are defined by their contents; entities by identity. Deciding which you have settles equality and mutability.

4. C No destructors, no manual freeing. `finalize` is deprecated; resources are released via `close()` in try-with-resources.

5. B Fluent setters assemble the parts readably; `build()` validates and creates the immutable object once.

CHECKPOINT · 22 MIN

Checkpoint — Classes & objects

This checkpoint covers the object model, constructors and initialisation order, static members, access modifiers and immutability, and class design.

How it works. Twelve questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- What `new` does, step by step, and what `this` is.
- What happens to the default constructor when you declare one.
- The order in which field initialisers and constructor bodies run.
- Why a static method cannot read an instance field.
- The four access levels and what `protected` includes.
- The recipe for an immutable class.

The programs ask you to write two small classes from scratch — an entity with invariants and a value type with `toString` — and drive them from `main`.

EXERCISE 1

A stack class

Implement `class IntStack` backed by an `int[]` that **grows** (double the array with `Arrays.copyOf` when full), with `push(int)`, `pop()` (throws `IllegalStateException` when empty), `peek()`, `size()` and `isEmpty()`. Read an integer `n` then `n` commands: `push <v>`, `pop` (print the popped value or `empty`), `peek` (print the top or `empty`), `size` (print the size).

Example: `6` then `push 1`, `push 2`, `peek`, `pop`, `pop`, `pop` →

```
2
2
1
empty
```

INPUT 1

```
6
push 1
push 2
peek
pop
pop
pop
```

OUTPUT 1

```
2
2
1
empty
```

INPUT 2

```
7
push 5
push 6
push 7
push 8
size
pop
size
```

OUTPUT 2

```
4
8
3
```

REFERENCE SOLUTION

```
import java.util.*;

class IntStack {
    private int[] items = new int[2];
    private int size = 0;

    void push(int v) {
        if (size == items.length) items = Arrays.copyOf(items, items.length * 2);
        items[size++] = v;
    }

    int pop() {
        if (size == 0) throw new IllegalStateException("empty");
        return items[--size];
    }

    int peek() {
        if (size == 0) throw new IllegalStateException("empty");
        return items[size - 1];
    }

    int size() { return size; }
    boolean isEmpty() { return size == 0; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        IntStack s = new IntStack();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "push" -> s.push(in.nextInt());
                case "pop" -> System.out.println(s.isEmpty() ? "empty" : String.valu
                case "peek" -> System.out.println(s.isEmpty() ? "empty" : String.val
                default -> System.out.println(s.size());
            }
        }
    }
}
```

EXERCISE 2

Money that adds up

Write an immutable `final class Money` with `private final long cents` and `private final String currency`, a constructor that rejects a negative amount or a blank currency, `plus(Money other)` (throws `IllegalArgumentException` if the currencies differ), `times(int factor)`, and `toString()` formatted as `<currency> <whole>.<cc>` (two-digit cents).

Read an integer `n`, then `n` lines `<currency> <cents>`. Sum all amounts of the **first** line's currency with `plus`, skipping lines whose currency differs (count them), then print the total and `skipped=<count>`.

Example: `3` then `EUR 150`, `USD 100`, `EUR 5` →

```
EUR 1.55
skipped=1
```

INPUT 1

```
3
EUR 150
USD 100
EUR 5
```

OUTPUT 1

```
EUR 1.55
skipped=1
```

INPUT 2

```
1
INR 7
```

OUTPUT 2

```
INR 0.07
skipped=0
```

REFERENCE SOLUTION

```
import java.util.*;

final class Money {
    private final long cents;
    private final String currency;

    Money(long cents, String currency) {
        if (cents < 0) throw new IllegalArgumentException("negative amount");
        if (currency == null || currency.isBlank()) throw new IllegalArgumentExcepti
        this.cents = cents;
        this.currency = currency;
    }

    Money plus(Money other) {
        if (!currency.equals(other.currency)) throw new IllegalArgumentException("cu
        return new Money(cents + other.cents, currency);
    }

    Money times(int factor) { return new Money(cents * factor, currency); }
    String currency() { return currency; }

    @Override
    public String toString() {
        return currency + " " + (cents / 100) + "." + String.format("%02d", cents %
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Money total = null;
        int skipped = 0;
        for (int i = 0; i < n; i++) {
            String currency = in.next();
            long cents = in.nextLong();
            Money m = new Money(cents, currency);
            if (total == null) total = m;
            else if (total.currency().equals(m.currency())) total = total.plus(m);
            else skipped++;
        }
        System.out.println(total);
        System.out.println("skipped=" + skipped);
    }
}
```

EXERCISE 3

Library loans

Model a small library. `class Book` has a `title` and a `loanedTo` (member name or `null`). `class Library` holds books in insertion order and supports `lend <title> <member>` (`lent <title> to <member>`, or `<title> is already out` / `no such book`), `return <title>` (`returned <title>`, or `<title> was not out`), and `report` (one line per book: `<title>: available` or `<title>: with <member>`). Read an integer `n` and `n` titles, then `m` commands.

Example: `2` then `Dune Emma`, then `3` then `lend Dune ann`, `lend Dune bob`, `report` →

```
lent Dune to ann
Dune is already out
Dune: with ann
Emma: available
```

INPUT 1

```
2
Dune Emma
3
lend Dune ann
lend Dune bob
report
```

OUTPUT 1

```
lent Dune to ann
Dune is already out
Dune: with ann
Emma: available
```

INPUT 2

```
1
Ulysses
3
return Ulysses
lend Odyssey cy
report
```

OUTPUT 2

```
Ulysses was not out
no such book
Ulysses: available
```

REFERENCE SOLUTION

```
import java.util.*;

class Book {
    final String title;
    String loanedTo;
    Book(String title) { this.title = title; }
}

class Library {
    private final Map<String, Book> books = new LinkedHashMap<>();
    void addBook(String title) { books.put(title, new Book(title)); }
    String lend(String title, String member) {
        Book b = books.get(title);
        if (b == null) return "no such book";
        if (b.loanedTo != null) return title + " is already out";
        b.loanedTo = member;
        return "lent " + title + " to " + member;
    }
    String giveBack(String title) {
        Book b = books.get(title);
        if (b == null) return "no such book";
        if (b.loanedTo == null) return title + " was not out";
        b.loanedTo = null;
        return "returned " + title;
    }
    void report() {
        for (Book b : books.values()) System.out.println(b.title + ": " + (b.loanedT
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Library lib = new Library();
        for (int i = 0; i < n; i++) lib.addBook(in.next());
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "lend" -> System.out.println(lib.lend(in.next(), in.next()));
                case "return" -> System.out.println(lib.giveBack(in.next()));
                default -> lib.report();
            }
        }
    }
}
```

CHECKPOINT QUESTIONS

1. `Point p = null; p.x = 1;` throws…

A. `IllegalStateException`

B. `NullPointerException`

C. Nothing

D. `ClassCastException`

2. Which statement about constructors is true?

A. They can return a value

B. They are inherited by subclasses

C. Declaring any constructor suppresses the default no-arg one

D. They must be public

3. A field initialiser `private List<String> xs = new ArrayList<>();` runs…

A. Only for the no-arg constructor

B. Before every constructor body, after the superclass constructor

C. After the constructor body

D. At class load

4. Why can `main` not call `deposit()` (an instance method) directly?

A. `deposit` is private

B. `main` is static and has no `this`; an object is needed

C. `main` cannot call methods

D. `deposit` must be final

5. Which access modifier lets classes in the same package and subclasses elsewhere see a member?

A. `private`

B. Package-private

C. `protected`

D. `public`

6. An immutable class should be declared `final` because…

A. Final classes are faster

B. A subclass could otherwise add mutable state or override behaviour, breaking the guarantee

C. The compiler requires it

D. Only final classes can have final fields

7. `static` methods are appropriate for…

A. Anything called often

B. Pure functions of their arguments and factory methods

C. Methods that modify fields

D. Methods that must be overridden

8. Storing a caller's `int[]` in a field without copying it…

A. Is always fine

B. Aliases the array: the caller can change the object's state later

C. Copies it automatically

D. Is a compile error

9. Which is a value object rather than an entity?

A. A customer account

B. A database connection

C. A `Money` amount

D. An HTTP session

10. `@Override` on `toString` is useful because…

A. It makes the method faster

B. A misspelled name would become a compile error instead of a silent new method

C. It is required for `println`

D. It makes the method static

11. Calling an overridable method from a constructor is risky because…

A. It is a compile error

B. A subclass override may run before the subclass's fields are initialised

C. It creates a memory leak

D. It is slow

12. When does an object become eligible for garbage collection?

A. When `System.gc()` is called

B. When its variable goes out of scope, always

C. When no chain of references from a GC root reaches it

D. When `finalize` returns

ANSWER KEY · CHECKPOINT — CLASSES & OBJECTS

1. B Dereferencing a null reference — field or method — always throws NPE.

2. C Constructors have no return type, are not inherited, and may have any access level.

3. B Instance initialisers run for each `new`, in textual order, before the constructor's own statements.

4. B Static context has no instance. `new BankAccount().deposit(…)` or store the object in a variable.

5. C `protected` = package + subclasses. Package-private = package only.

6. B Immutability is a promise about every instance; preventing subclassing keeps the promise for objects created by others.

7. B No `this` means no instance state; that fits utilities and factories, not behaviour that depends on an object.

8. B Defensive copies on the way in (and out) are how a class owns its data.

9. C Money is defined by its amount and currency, not by identity; equal contents mean the same value.

10. B The annotation asks the compiler to verify that a superclass or interface method is actually being overridden.

11. B The subclass constructor body — and its field initialisers — have not run yet, so the override sees default values.

12. C Reachability is the criterion. A local going out of scope matters only if it was the last reference.