Module 8 of 20

OOP II — inheritance & polymorphism

extends and super, the rules of overriding, dynamic dispatch and casting, equals/hashCode/toString, abstract classes, and when to compose instead.

Inheritance lets one class be a variation of another and polymorphism lets code written for the parent run correctly for every child. Together they are what makes object-oriented programs extensible — and, misused, what makes them fragile.

This module covers the mechanics exactly: constructor chaining through super, what is inherited and what is merely present, the precise rules for overriding versus hiding, how dynamic dispatch chooses a method at run time, up- and down-casting with instanceof patterns, the Object methods and the equals/hashCode contract that every collection depends on, abstract classes and the template method pattern, and finally the judgement — Liskov substitution, the fragile base class problem, composition, final and sealed — of when inheritance is the wrong tool.

By the end you can design a small hierarchy correctly, write equals and hashCode that survive a HashSet, and defend a composition-over-inheritance decision in an interview.

CONTENTS

1	Extending classes — super, constructors and what is inherited	14 min
2	Overriding — the rules and the traps	14 min
3	Polymorphism, casting and instanceof	14 min
4	The Object class — equals, hashCode and toString	16 min
5	Abstract classes and the template method	12 min
6	When not to inherit — composition, Liskov and final	13 min
✓	Checkpoint — Inheritance & polymorphism	24 min

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

LESSON 1 · 14 MIN

Extending classes — super, constructors and what is inherited

Inheritance lets a class be defined as a *variation* of another: a `SavingsAccount` **is an** `Account` with interest added. The subclass gets the parent's fields and methods, may add more, and may replace some. Java has single inheritance of classes (one parent), a root class every class descends from (`Object`), and precise rules about constructors that trip up most people the first time. This lesson is the mechanics; the next ones are what you can do with them.

`extends`

```
public class Account {
    protected long balanceCents;            // visible to subclasses
    private final String owner;             // NOT visible to subclasses dire

    public Account(String owner) { this.owner = owner; }
    public void deposit(long c) { balanceCents += c; }
    public long balance()       { return balanceCents; }
    public String owner()       { return owner; }
}

public class SavingsAccount extends Account {
    private final double rate;

    public SavingsAccount(String owner, double rate) {
        super(owner);                       // MUST call a parent constructor
        this.rate = rate;
    }

    public void addInterest() {
        balanceCents += Math.round(balanceCents * rate);   // protected field
    }
}
```

`SavingsAccount` has everything `Account` has — `deposit`, `balance`, `owner`, the fields — plus `rate` and `addInterest`. Code that works with an `Account` works with a `SavingsAccount`

(next lessons on polymorphism). The relationship is **is-a**: every savings account is an account. If you cannot say "X is a Y" truthfully, do not use `extends` (lesson 6).

What is inherited

All `public` and `protected` members, and package-private members if the subclass is in the same package.

`private` members are **present** in the object (the memory is there — `owner` exists inside every `SavingsAccount`) but **not accessible** by name; reach them through the parent's public/protected methods.

Static members are inherited in the sense that `SavingsAccount.someStatic()` resolves, but they are not overridden (Module 7).

Constructors are not inherited. Each class declares its own.

Constructors and `super(...)`

Every constructor begins by calling a parent constructor — explicitly with `super(args)` as its **first statement**, or implicitly with `super()` if you write nothing. The chain runs up to `Object`, and the parent's body finishes before the child's initialisers begin (the order from Module 7):

```
new SavingsAccount("Ada", 0.02)
  → SavingsAccount constructor starts → super("Ada")
    → Account constructor starts → (implicit) super() → Object()
    → Account field initialisers, Account body: owner = "Ada"
  → SavingsAccount field initialisers, body: rate = 0.02
```

Consequences:

If the parent has **no no-arg constructor**, every subclass constructor must call `super(...)` with arguments explicitly, or it will not compile ("constructor Account in class Account cannot be applied to given types").

If you declare no constructor in the subclass, the default one calls `super()` — which fails to compile for the same reason when the parent lacks a no-arg constructor.

`this(...)` and `super(...)` cannot both appear; a constructor that delegates with `this(...)` relies on the target to call `super`.

`super.method()`

`super` also reaches the parent's version of a method the subclass has overridden:

```
@Override
public void deposit(long c) {
    super.deposit(c);                       // do what Account does…
    log("deposited " + c);                  // …then more
}
```

`super.` is not a reference to a separate object — there is one object; `super.deposit` simply selects the parent's implementation. It cannot be chained (`super.super.x` is illegal).

`protected`

A parent field or method marked `protected` is accessible in subclasses (and in the same package). It is the parent saying "this is part of my extension interface". Use it sparingly: a protected field is a field two classes may mutate, which weakens the parent's invariants. Prefer `private` fields with protected *methods* when subclasses need controlled access.

The `Object` root

A class with no `extends` implicitly extends `java.lang.Object`. So every class inherits `toString`, `equals`, `hashCode`, `getClass`, `wait` / `notify` (Module 17) and `clone` / `finalize` (deprecated). Lesson 4 is about the ones you override.

Single inheritance, and why

A Java class extends **one** class. Multiple inheritance of implementation (C++) creates the "diamond problem" — two parents defining the same method — and complicates object layout. Java gives multiple inheritance of *type* through interfaces (Module 9), which since Java 8 may carry default methods with explicit conflict rules. The class chain stays a simple line: `SavingsAccount → Account → Object`.

`final` stops the chain

`public final class String` cannot be extended; `public final void deposit(long c)` cannot be overridden. Use `final` on classes designed as values (immutability needs it — Module 7) and on methods that subclasses must not alter (a template's fixed steps, security checks). The JIT does not need `final` to optimise; the reason is design, not speed.

Inheritance and access to the parent's state

A subclass that needs to change the parent's private state does so through the parent's methods. If that feels restrictive, the parent's API is missing an operation, or inheritance is the wrong tool. Reaching in via a protected field is the quick fix that becomes the maintenance problem.

A worked example

```
public class Employee {
    private final String name;
    protected final long baseSalary;
    public Employee(String name, long baseSalary) { this.name = name; this.bas
    public long pay() { return baseSalary; }
    public String name() { return name; }
}

public class Manager extends Employee {
    private final long bonus;
    public Manager(String name, long baseSalary, long bonus) { super(name, bas
    @Override public long pay() { return super.pay() + bonus; }
}
```

`Manager` reuses construction and `name()` from `Employee`, extends `pay()`, and can be used wherever an `Employee` is expected — the subject of lesson 3.

Interview angle

"What does a subclass inherit?" Public/protected (and package) members; private ones exist but are inaccessible; constructors are not inherited.

"What is the first statement of every constructor?" A `super(...)` or `this(...)` call, implicit `super()` if absent.

"Why does `class B extends A {}` fail when `A` has only `A(int)`?" The implicit `super()` has no matching constructor.

"Does Java support multiple inheritance?" Of classes, no; of interface types, yes.

Key takeaways

`extends` gives is-a: the subclass has the parent's accessible members plus its own.

Constructors chain to `super(...)` first; a parent without a no-arg constructor forces explicit calls.

`private` parent members exist in the object but are reached only through parent methods; `protected` opens them to subclasses and the package.

Every class descends from `Object`; `final` ends the line; one class parent only.

EXERCISE 1

Employees and managers

Write `class Employee` with a name and a monthly base salary (long), a constructor, `pay()` returning the base, and `toString()` giving `<name>:<pay>`. Write `class Manager extends Employee` adding a bonus, whose constructor calls `super(...)` and whose `pay()` returns `super.pay() + bonus`.

Read an integer `n`, then `n` lines `emp <name> <base>` or `mgr <name> <base> <bonus>`. Create the right object for each and print it; finally print `payroll=<sum of all pay()>`.

Example: `2` then `emp ada 5000`, `mgr bob 6000 1500` →

```
ada:5000
bob:7500
payroll=12500
```

INPUT 1

```
2
emp ada 5000
mgr bob 6000 1500
```

OUTPUT 1

```
ada:5000
bob:7500
payroll=12500
```

INPUT 2

```
1
mgr eve 1 0
```

OUTPUT 2

```
eve:1
payroll=1
```

REFERENCE SOLUTION

```java
import java.util.*;

class Employee {
    private final String name;
    protected final long base;

    Employee(String name, long base) {
        this.name = name;
        this.base = base;
    }

    long pay() { return base; }
    String name() { return name; }

    @Override
    public String toString() { return name + ":" + pay(); }
}

class Manager extends Employee {
    private final long bonus;

    Manager(String name, long base, long bonus) {
        super(name, base);
        this.bonus = bonus;
    }

    @Override
    long pay() { return super.pay() + bonus; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        long payroll = 0;
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            Employee e;
            if (kind.equals("mgr")) e = new Manager(in.next(), in.nextLong(), in.nex
            else e = new Employee(in.next(), in.nextLong());
            System.out.println(e);
            payroll += e.pay();
        }
        System.out.println("payroll=" + payroll);
    }
}
```

EXERCISE 2

Vehicles, constructed in order

Write `class Vehicle` (fields `name`, `wheels`; its constructor prints `Vehicle(<name>, <wheels>)`), `class Car extends Vehicle` (constructor `Car(String name)` calls `super(name, 4)` then prints `Car(<name>)`) and `class Motorbike extends Vehicle` (`super(name, 2)`, then `Motorbike(<name>)`). Add `String describe()` in `Vehicle` returning `<name> has <wheels> wheels`. Read `n` lines `<kind> <name>`; construct each and print `describe()`. The superclass constructor always runs first.

Example: `2` then `car Polo`, `bike Vespa` →

```
Vehicle(Polo, 4)
Car(Polo)
Polo has 4 wheels
Vehicle(Vespa, 2)
Motorbike(Vespa)
Vespa has 2 wheels
```

INPUT 1

```
2
car Polo
bike Vespa
```

OUTPUT 1

```
Vehicle(Polo, 4)
Car(Polo)
Polo has 4 wheels
Vehicle(Vespa, 2)
Motorbike(Vespa)
Vespa has 2 wheels
```

INPUT 2

```
1
bike Ducati
```

OUTPUT 2

```
Vehicle(Ducati, 2)
Motorbike(Ducati)
Ducati has 2 wheels
```

REFERENCE SOLUTION

```java
import java.util.*;

class Vehicle {
    final String name;
    final int wheels;
    Vehicle(String name, int wheels) {
        this.name = name;
        this.wheels = wheels;
        System.out.println("Vehicle(" + name + ", " + wheels + ")");
    }
    String describe() { return name + " has " + wheels + " wheels"; }
}

class Car extends Vehicle {
    Car(String name) {
        super(name, 4);
        System.out.println("Car(" + name + ")");
    }
}

class Motorbike extends Vehicle {
    Motorbike(String name) {
        super(name, 2);
        System.out.println("Motorbike(" + name + ")");
    }
}

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 kind = in.next(), name = in.next();
            Vehicle v = kind.equals("car") ? new Car(name) : new Motorbike(name);
            System.out.println(v.describe());
        }
    }
}
```

QUIZ

1. `class A { A(int x) {} } class B extends A { }` — result?

A. Compiles

B. Compile error: B's implicit `super()` finds no `A()` constructor

C. Runtime error

D. B gets `A(int)` automatically

2. A `private` field of the parent…

A. Does not exist in subclass objects

B. Exists in every subclass object but is not accessible by name in the subclass

C. Becomes protected

D. Is copied into the subclass

3. Which is the first statement of a constructor that neither calls `this(...)` nor `super(...)`?

A. Nothing special

B. An implicit `super()`

C. An implicit `this()`

D. A call to `Object.init`

4. How many class parents can a Java class have?

A. Any number

B. Exactly one (implicitly `Object` if unspecified)

C. Two

D. Zero

5. `super.super.method()` is…

A. Valid

B. A compile error — `super` cannot be chained

C. Valid only in abstract classes

D. The same as `super.method()`

ANSWER KEY · EXTENDING CLASSES — SUPER, CONSTRUCTORS AND WHAT IS INHERITED

1. B Constructors are not inherited, and B's default constructor calls `super()`. B must declare a constructor that calls `super(someInt)`.

2. B The memory is there; access is not. Reach it through the parent's public or protected methods.

3. B Every constructor chains to a parent constructor; the compiler inserts `super()` when you write neither.

4. B Single inheritance of classes avoids the diamond problem; multiple inheritance of type comes from interfaces.

5. B `super` selects the immediate parent's implementation; skipping a level would break the parent's encapsulation.

LESSON 2 · 14 MIN

Overriding — the rules and the traps

A subclass **overrides** a parent method by declaring one with the same signature; calls on the object then run the subclass version, whatever the reference's declared type. The rules for a valid override are exact — the compiler enforces some and warns about none of the rest — and the difference between overriding and *hiding* is a reliable interview question. This lesson is the checklist.

The rules of a valid override

Given `class Sub extends Base` and a method `m` in `Base`:

Rule	Detail
Same name and parameter types	Exactly. Different parameters = an *overload*, not an override.
Return type	Same, or a **subtype** for reference returns (covariant return: `Base.copy()` returns `Base`, `Sub.copy()` may return `Sub`). Primitive returns must match exactly.
Access	Not more restrictive. `public` in the parent must stay `public`; `protected` may become `public`.
Checked exceptions	May throw fewer or narrower checked exceptions, never new or broader ones (Module 12). Unchecked ones are free.
Not `final`, not `static`, not `private`	A `final` method cannot be overridden; a `static` one is hidden, not overridden; a `private` one is invisible, so a same-named method in the subclass is unrelated.
Instance ↔ instance	An instance method cannot override a static one or vice versa.

```
class Base {
    protected Object make() throws IOException { … }
}
class Sub extends Base {
    @Override
    public String make() { … }        // valid: wider access, covariant return
}
```

`@Override`

Always annotate an override. The annotation asks the compiler to *verify* the method overrides something; a typo in the name (`tostring`), a wrong parameter type (`equals(Point p)` instead of `equals(Object o)`), or a `final` parent method then fails to compile instead of silently creating a new method that nobody calls. `equals(Point)` is the classic: it compiles, `HashSet` never calls it, and the bug appears months later.

Calling the parent version

```
@Override
public String toString() {
    return super.toString() + "[extra]";
}
```

`super.m()` invokes the parent's implementation on the same object. Common in `toString`, in `equals` for subclasses that add fields, and in hook methods that extend rather than replace behaviour.

Overriding versus overloading

```
class Printer {
    void print(Object o) { System.out.println("object"); }
}
class FancyPrinter extends Printer {
    void print(String s) { System.out.println("string"); }      // OVERLOAD —
    @Override void print(Object o) { System.out.println("fancy object"); }
}

Printer p = new FancyPrinter();
p.print("hi");        // "fancy object" — the reference type Printer has only
                     //                  and then dispatched to FancyPrinter
((FancyPrinter) p).print("hi");   // "string" — now the overload set includes
```

Two decisions happen: the **compiler** picks the overload from the reference's static type (Module 5); the **JVM** picks the override from the object's runtime class. Keep both in mind when a call does something unexpected.

Static methods are hidden, not overridden

```
class A { static String who() { return "A"; } }
class B extends A { static String who() { return "B"; } }

A ref = new B();
ref.who();         // "A" — resolved by the STATIC type; static methods do not
B.who();           // "B"
```

`B.who` *hides* `A.who`. There is no polymorphism for statics; calling them through an instance reference is legal and misleading, which is why style checkers reject it. `@Override` on a static method is a compile error — the compiler's way of saying so.

Fields are hidden too

```
class A { String name = "A"; }
class B extends A { String name = "B"; }
A a = new B();
a.name             // "A" — fields are resolved by static type, never overrid
((B) a).name       // "B"
```

An object of class `B` has *two* `name` fields. This is confusing enough that redeclaring a parent's field is treated as a bug. Fields are private; behaviour is overridden through methods.

Overriding `Object` methods

`toString`, `equals`, `hashCode` are the three every class overrides eventually (lesson 4). `equals` **must** take `Object` — `public boolean equals(Object o)` — or it is an overload that collections ignore. `clone` and `finalize` are legacy; do not override them.

Calling overridable methods from constructors

```
class Base {
    Base() { init(); }                // calls the override before Sub's fiel
    void init() { }
}
class Sub extends Base {
    private final List<String> items = new ArrayList<>();
    @Override void init() { items.add("x"); }    // NullPointerException: iter
}
```

The parent constructor runs before the subclass's field initialisers (Module 7), so an override invoked from it sees default field values. Rule: constructors call only `private`, `static` or `final` methods. Frameworks that need post-construction hooks provide an explicit `init()` called after `new`.

Covariant returns and fluent APIs

```
class Builder { Builder name(String n) { …; return this; } }
class FancyBuilder extends Builder {
    @Override FancyBuilder name(String n) { super.name(n); return this; }   //
    FancyBuilder colour(String c) { … }
}
new FancyBuilder().name("x").colour("red");    // compiles only because name()
```

Covariant returns exist for exactly this — and for `clone()` returning the precise type.

Exceptions in overrides

If `Base.load() throws IOException`, `Sub.load()` may throw `IOException`, a subclass of it, or nothing — but not `Exception` or `SQLException`. Callers holding a `Base` reference catch `IOException`; a broader exception would escape their handlers. Unchecked exceptions are not part of the signature and may be added freely.

Interview angle

"Can you override a static method?" No — hiding. *"A private one?"* No — invisible. *"A final one?"* No.

"Can an override return a different type?" A subtype of the parent's reference return type.

"Why must `equals` take `Object`?" Otherwise it overloads instead of overriding and collections do not call it.

"Can an override be more restrictive?" No: access may widen, never narrow.

"What does `@Override` do at run time?" Nothing; it is a compile-time check.

Key takeaways

Same signature, compatible (covariant) return, no narrower access, no broader checked exceptions; not for `static`, `private` or `final` methods.

Always `@Override`; `equals(Object)`, not `equals(MyType)`.

Overloads are chosen by the compiler from the static type; overrides by the JVM from the runtime class.

Static methods and fields are hidden, not overridden — resolved by reference type.

Never call overridable methods from a constructor.

EXERCISE 1

Override or overload?

Complete the program so its output demonstrates the difference between overriding and overloading. `Printer` has `describe(Object o)` returning `"object"` and `static String kind()` returning `"base"`. `FancyPrinter` **overrides** `describe(Object)` to return `"fancy object"`, adds an

overload `describe(String s)` returning `"fancy string"`, and **hides** `kind()` with one returning `"fancy"`.

`main` (already written) reads a line of text and prints four lines using a `Printer`-typed reference to a `FancyPrinter`, then a `FancyPrinter`-typed reference. Make the classes so the expected output appears.

Expected output for any input line:

```
fancy object
fancy string
base
fancy
```

INPUT 1

```
hello
```

OUTPUT 1

```
fancy object
fancy string
base
fancy
```

REFERENCE SOLUTION

```java
import java.util.*;

class Printer {
    String describe(Object o) { return "object"; }
    static String kind() { return "base"; }
}

class FancyPrinter extends Printer {
    @Override
    String describe(Object o) { return "fancy object"; }

    String describe(String s) { return "fancy string"; }

    static String kind() { return "fancy"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String text = in.nextLine();
        Printer p = new FancyPrinter();
        FancyPrinter f = new FancyPrinter();
        System.out.println(p.describe(text));
        System.out.println(f.describe(text));
        System.out.println(p.kind());
        System.out.println(f.kind());
    }
}
```

EXERCISE 2

toString up the chain

Write `class Person` (`name`) with `toString()` returning `Person[name=<name>]`; `class Employee extends Person` (`salary`) whose override returns `super.toString()` with `, salary=<salary>` inserted before the closing bracket; and `class Manager extends Employee` (`reports`) doing the same with `, reports=<n>`. Mark every override with `@Override`. Read `n` lines `<kind> <name> [salary] [reports]` and print each object through a `Person` reference — the most specific `toString` runs.

Example: `3` then `person ann`, `employee bob 5000`, `manager cy 9000 4` →

```
Person[name=ann]
Person[name=bob, salary=5000]
Person[name=cy, salary=9000, reports=4]
```

INPUT 1

```
3
person ann
employee bob 5000
manager cy 9000 4
```

OUTPUT 1

```
Person[name=ann]
Person[name=bob, salary=5000]
Person[name=cy, salary=9000, reports=4]
```

INPUT 2

```
1
manager solo 1 0
```

OUTPUT 2

```
Person[name=solo, salary=1, reports=0]
```

REFERENCE SOLUTION

```java
import java.util.*;

class Person {
    final String name;
    Person(String name) { this.name = name; }
    @Override public String toString() { return "Person[name=" + name + "]"; }
}

class Employee extends Person {
    final int salary;
    Employee(String name, int salary) { super(name); this.salary = salary; }
    @Override public String toString() {
        String base = super.toString();
        return base.substring(0, base.length() - 1) + ", salary=" + salary + "]";
    }
}

class Manager extends Employee {
    final int reports;
    Manager(String name, int salary, int reports) { super(name, salary); this.report
    @Override public String toString() {
        String base = super.toString();
        return base.substring(0, base.length() - 1) + ", reports=" + reports + "]";
    }
}

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 kind = in.next();
            Person p = switch (kind) {
                case "person" -> new Person(in.next());
                case "employee" -> new Employee(in.next(), in.nextInt());
                default -> new Manager(in.next(), in.nextInt(), in.nextInt());
            };
            System.out.println(p);
        }
    }
}
```

QUIZ

1. Which change is **not** allowed in an override?

A. Widening access from `protected` to `public`

B. Returning a subtype of the parent's reference return type

C. Narrowing access from `public` to `protected`

D. Throwing fewer checked exceptions

2. `public boolean equals(Point p)` in class `Point` is…

A. A valid override

B. An overload — collections still use `Object.equals` (identity)

C. A compile error

D. Automatically promoted to `equals(Object)`

3. `A ref = new B(); ref.staticMethod();` where both classes declare `staticMethod` — which runs?

A. B's — dynamic dispatch

B. A's — static methods are resolved by the reference type

C. Both

D. Compile error

4. What does `@Override` do?

A. Makes the method virtual

B. Asks the compiler to verify the method really overrides a supertype method

C. Calls the parent version automatically

D. Prevents further overriding

5. Calling an overridable method from a constructor is dangerous because…

A. It is a compile error

B. The subclass override runs before the subclass's fields are initialised

C. It causes infinite recursion

D. It is slow

ANSWER KEY · OVERRIDING — THE RULES AND THE TRAPS

1. C An override may not be more restrictive — callers holding a parent reference must still be able to call it.

2. B The parameter type differs from `Object`, so it does not override. `@Override` would have caught it.

3. B Static methods are hidden, not overridden. Call them through the class name to avoid confusion.

4. B It has no run-time effect. It turns a typo or a wrong parameter type into a compile error.

5. B Parent constructors run first; the subclass's field initialisers have not executed yet, so the override sees defaults (null, 0).

LESSON 3 · 14 MIN

Polymorphism, casting and instanceof

Polymorphism — "many forms" — is the ability to treat a `SavingsAccount`, a `CheckingAccount` and a `BusinessAccount` all as `Account` and have each respond in its own way. It is the feature that makes object-oriented code extensible without modification: new subclasses plug into old code. This lesson covers how it works at run time (dynamic dispatch), how references move between types (up- and down-casting), and the modern `instanceof` pattern that replaced the cast-after-check idiom.

Subtype polymorphism

```
Account[] accounts = { new SavingsAccount("a", 0.02), new CheckingAccount("b"
for (Account acct : accounts) {
    acct.monthEnd();          // each object runs ITS class's monthEnd
}
```

A variable of type `Account` may hold any `Account` *or subclass instance*. Calling an overridden method on it runs the version belonging to the **object's actual class**, decided at run time. The loop does not know or care which kinds of account it holds — and a `PremiumAccount` written next year will work in it unchanged. That is the whole point.

Dynamic dispatch

At the bytecode level a call like `acct.monthEnd()` is an `invokevirtual` instruction. The JVM looks at the object `acct` refers to, finds its class's *virtual method table*, and jumps to the entry for `monthEnd` — the most specific override in the object's class chain. This lookup happens on every call (the JIT makes it nearly free, and inlines it when only one implementation is ever seen). Static, private and final methods skip it — there is nothing to choose.

Two rules follow:

The runtime class decides overrides; the reference type decides *what you are allowed to call*. `Account acct = new SavingsAccount(…); acct.addInterest();` does not compile — `Account` has no `addInterest` — even though the object could do it.

Fields and static methods are not dispatched (previous lesson).

Upcasting: always safe, usually implicit

```
SavingsAccount s = new SavingsAccount("a", 0.02);
Account a = s;                    // upcast: a SavingsAccount IS an Account
Object o = s;                     // everything is an Object
```

Widening a reference to a supertype needs no cast and cannot fail: the object *is* that type. You lose access to subclass-specific members through the wider reference; you gain the ability to mix types in one collection or parameter.

Downcasting: explicit, checked, can fail

```
Account a = getAccount();
SavingsAccount s = (SavingsAccount) a;    // compiles; at run time the JVM CHE
s.addInterest();
```

If `a` actually refers to a `CheckingAccount`, the cast throws `ClassCastException: CheckingAccount cannot be cast to SavingsAccount`. The compiler allows the cast because it *might* be right; the JVM verifies it. Casting between unrelated classes (`(String) account`) is a compile error — the compiler can see it can never succeed.

A downcast is a signal: the code is asking "which kind are you?" — something polymorphism was supposed to make unnecessary. Frequent downcasts suggest a missing method on the supertype.

`instanceof` and pattern matching

The traditional guard:

```
if (a instanceof SavingsAccount) {
    SavingsAccount s = (SavingsAccount) a;      // safe: just checked
    s.addInterest();
}
```

Since Java 16 the check and the cast are one expression:

```
if (a instanceof SavingsAccount s) {           // binds s if the test passes
    s.addInterest();
}
if (!(a instanceof SavingsAccount s)) return;  // s is in scope for the rest o
if (a instanceof SavingsAccount s && s.rate() > 0.01) { … }   // && may use s,
```

`instanceof` is `false` for `null` — no NPE — and true for the class and any subclass. `obj instanceof Object` is true for any non-null object. Pattern variables are the modern idiom; Java 21's pattern switch extends it to whole type hierarchies (Module 19).

`getClass()` versus `instanceof`

`a.getClass() == SavingsAccount.class` is true only for *exactly* that class, not subclasses. `instanceof` includes subclasses. In `equals` implementations the choice matters (next lesson); elsewhere, `instanceof` is almost always what you mean.

Polymorphism versus `switch` on type

```
// non-polymorphic: every new kind requires editing this method
double fee(Account a) {
    if (a instanceof SavingsAccount) return 0;
    if (a instanceof CheckingAccount) return 2;
    …
}
// polymorphic: each kind knows its fee
abstract double fee();      // in Account; overridden per subclass
```

The polymorphic version follows the **open/closed principle** — open to extension by adding subclasses, closed to modification of existing code. The `instanceof` chain is appropriate when the *operations* vary more than the *types* (a visitor over a fixed set of shapes), which is where sealed types and pattern switch shine.

Arrays, generics and variance

`Account[] accounts = new SavingsAccount[3];` compiles (arrays are covariant) and `accounts[0] = new CheckingAccount(…)` throws `ArrayStoreException`. `List<Account>`

`list = new ArrayList<SavingsAccount>();` does **not** compile — generics are invariant, on purpose, to catch that error at compile time (Module 13 introduces wildcards to relax it safely).

Polymorphism through interfaces

Everything here applies equally to interface types (Module 9): `List<String> list = new ArrayList<>();` is polymorphism — `list.add` dispatches to `ArrayList`'s implementation, and swapping in a `LinkedList` changes nothing else. Programming to the interface type is the everyday face of polymorphism.

Interview angle

"What is polymorphism?" One reference type, many runtime behaviours via overriding; calls dispatch on the object's class.

"Compile-time vs run-time polymorphism?" Overloading (static) vs overriding (dynamic).

"What happens on a bad downcast?" `ClassCastException` at run time.

"`instanceof` with `null`?" `false`.

"Why does `List<Animal> = new ArrayList<Dog>()` fail while `Animal[] = new Dog[1]` compiles?" Generics are invariant for safety; arrays are covariant and checked at run time.

Key takeaways

A supertype reference can hold any subtype object; overridden methods dispatch on the object's runtime class.

The reference type limits what you may call; the object decides what runs.

Upcasts are implicit and safe; downcasts are explicit and checked — `ClassCastException` on failure.

`instanceof Type var` tests and binds in one step, is false for null, and includes subclasses.

Prefer overriding to `instanceof` chains when adding *types*; the reverse when adding *operations*.

EXERCISE 1

Shapes in one array

Write an abstract-free hierarchy: `class Shape` with `double area()` returning 0 and `String name()` returning `"shape"`; `Circle`, `Rect` and `Square extends Rect` overriding both (`Square` passes the side twice to `Rect`'s constructor and names itself `"square"`).

Read an integer `n` and `n` lines `circle <r>`, `rect <w> <h>` or `square <s>`; store them in a `Shape[]`, then print each as `<name> <area with 2 decimals>` and a final line `squares=<count of objects that are instanceof Square>` — note that every `Square` is also a `Rect`.

Example: `3` then `circle 1`, `rect 2 3`, `square 2` →

```
circle 3.14
rect 6.00
square 4.00
squares=1
```

INPUT 1

```
3
circle 1
rect 2 3
square 2
```

OUTPUT 1

```
circle 3.14
rect 6.00
square 4.00
squares=1
```

INPUT 2

```
2
square 1.5
square 0
```

OUTPUT 2

```
square 2.25
square 0.00
squares=2
```

REFERENCE SOLUTION

```java
import java.util.*;

class Shape {
    double area() { return 0; }
    String name() { return "shape"; }
}

class Circle extends Shape {
    private final double r;
    Circle(double r) { this.r = r; }
    @Override double area() { return Math.PI * r * r; }
    @Override String name() { return "circle"; }
}

class Rect extends Shape {
    protected final double w, h;
    Rect(double w, double h) { this.w = w; this.h = h; }
    @Override double area() { return w * h; }
    @Override String name() { return "rect"; }
}

class Square extends Rect {
    Square(double s) { super(s, s); }
    @Override String name() { return "square"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Shape[] shapes = new Shape[n];
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            switch (kind) {
                case "circle" -> shapes[i] = new Circle(in.nextDouble());
                case "rect" -> shapes[i] = new Rect(in.nextDouble(), in.nextDouble()
                default -> shapes[i] = new Square(in.nextDouble());
            }
        }
        int squares = 0;
        for (Shape s : shapes) {
            System.out.printf("%s %.2f%n", s.name(), s.area());
            if (s instanceof Square) squares++;
        }
        System.out.println("squares=" + squares);
    }
}
```

EXERCISE 2

Payments, dispatched

Write an abstract-free hierarchy: `class Payment` with `amount` and `double fee()` returning 0 and `String kind()` returning `"payment"`; `class Card extends Payment` (fee 2% of the amount, kind `"card"`); `class Transfer extends Payment` (flat fee 1.50 when the amount is below 100, else 0, kind `"transfer"`); `class Cash extends Payment` (kind `"cash"`). Read `n` lines `<kind> <amount>`, store them in a `Payment[]`, and print `<kind> <amount> fee <fee>` (two decimals) per item plus `total fees <sum>`. Also print `cards=<count via instanceof>`.

Example: `3` then `card 50`, `transfer 80`, `cash 20` →

```
card 50.00 fee 1.00
transfer 80.00 fee 1.50
cash 20.00 fee 0.00
total fees 2.50
cards=1
```

INPUT 1

```
3
card 50
transfer 80
cash 20
```

OUTPUT 1

```
card 50.00 fee 1.00
transfer 80.00 fee 1.50
cash 20.00 fee 0.00
total fees 2.50
cards=1
```

INPUT 2

```
2
transfer 100
card 0
```

OUTPUT 2

```
transfer 100.00 fee 0.00
card 0.00 fee 0.00
total fees 0.00
cards=1
```

REFERENCE SOLUTION

```java
import java.util.*;

class Payment {
    final double amount;
    Payment(double amount) { this.amount = amount; }
    double fee() { return 0; }
    String kind() { return "payment"; }
}
class Card extends Payment {
    Card(double amount) { super(amount); }
    @Override double fee() { return amount * 0.02; }
    @Override String kind() { return "card"; }
}
class Transfer extends Payment {
    Transfer(double amount) { super(amount); }
    @Override double fee() { return amount < 100 ? 1.5 : 0; }
    @Override String kind() { return "transfer"; }
}
class Cash extends Payment {
    Cash(double amount) { super(amount); }
    @Override String kind() { return "cash"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Payment[] all = new Payment[n];
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            double amount = in.nextDouble();
            all[i] = switch (kind) {
                case "card" -> new Card(amount);
                case "transfer" -> new Transfer(amount);
                default -> new Cash(amount);
            };
        }
        double total = 0;
        int cards = 0;
        for (Payment p : all) {
            total += p.fee();
            if (p instanceof Card) cards++;
            System.out.println(String.format(Locale.ROOT, "%s %.2f fee %.2f", p.kinc
        }
        System.out.println(String.format(Locale.ROOT, "total fees %.2f", total));
        System.out.println("cards=" + cards);
    }
}
```

QUIZ

1. `Account a = new SavingsAccount(); a.addInterest();` where only `SavingsAccount` declares `addInterest` …

A. Runs `addInterest`

B. Compile error: the reference type `Account` has no such method

C. Throws `ClassCastException`

D. Runs `Account`'s version

2. A downcast to the wrong class results in…

A. A compile error

B. `ClassCastException` at run time

C. null

D. Silent truncation

3. `null instanceof String` is…

A. `true`

B. `false`

C. `NullPointerException`

D. Compile error

4. `if (o instanceof Circle c && c.radius() > 1)` — the pattern variable `c` …

A. Is in scope on both sides of `&&`

B. Is in scope in the right operand and the `if` body, because the test must have passed

C. Is null when the test fails

D. Requires a separate cast

5. Which is compile-time polymorphism?

A. Overriding

B. Overloading

C. Dynamic dispatch

D. Downcasting

ANSWER KEY · POLYMORPHISM, CASTING AND INSTANCEOF

1. B The reference type limits what you may call; the object's class decides what runs for methods that *are* callable.

2. B The compiler allows casts that might succeed; the JVM checks the object's actual class when the cast executes.

3. B `instanceof` is always false for null — one reason it is a safe guard before a cast.

4. B Flow scoping: `c` exists wherever the compiler can prove the `instanceof` was true. With `||` it would not be.

5. B Overload resolution happens at compile time from argument types; overriding is resolved at run time.

LESSON 4 · 16 MIN

The Object class — equals, hashCode and toString

Every class inherits eleven methods from `java.lang.Object`. Three of them — `equals`, `hashCode` and `toString` — define how your objects compare, hash and print, and every collection, every map and every log line depends on them. Their default implementations are identity-based; the moment a class is a *value*, you override them together, according to a contract the whole library assumes. This lesson is that contract and the correct way to meet it.

What `Object` provides

Method	Default behaviour
`equals(Object)`	`this == o` — identity
`hashCode()`	A number derived from the object's identity (not the address, but stable for its life)
`toString()`	`getClass().getName() + "@" + Integer.toHexString(hashCode())`
`getClass()`	The runtime `Class` object — final, cannot be overridden
`clone()`	Shallow field copy, `protected`, needs `Cloneable` — legacy, avoid
`finalize()`	Deprecated for removal
`wait`, `notify`, `notifyAll`	Thread coordination — Module 17, final

The `equals` contract

For any non-null references `x`, `y`, `z`:

Reflexive: `x.equals(x)` is true.

Symmetric: `x.equals(y)` ⇔ `y.equals(x)`.

Transitive: `x.equals(y)` and `y.equals(z)` ⇒ `x.equals(z)`.

Consistent: repeated calls give the same result while the objects are unchanged.

`x.equals(null)` is false — never throws.

And the bridge to hashing: **equal objects must have equal hash codes**. (Unequal objects may share a hash code; that is a collision, and allowed.)

The canonical `equals`

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;                                  // fast path, and rej
    if (o == null || getClass() ≠ o.getClass()) return false;    // type check
    Point other = (Point) o;
    return x == other.x && y == other.y;                         // field comparison
}
```

Line by line:

The parameter is `Object`. `equals(Point p)` is an overload that `HashSet`, `List.contains` and `Objects.equals` never call.

`getClass() ≠ o.getClass()` makes `equals` false across different classes, including subclasses — which keeps symmetry when a subclass adds fields. The alternative `o instanceof Point` lets subclasses be equal to parents and needs care (a `ColorPoint` equal to a `Point` but not vice versa violates symmetry). Use `instanceof` only when subclasses add no state or when the class is `final`.

Compare primitives with `==`, `double` / `float` with `Double.compare(a, b) == 0` (handles NaN and −0.0), references with `Objects.equals(a, b)` (null-safe), arrays with `Arrays.equals`.

The matching `hashCode`

```
@Override
public int hashCode() {
    return Objects.hash(x, y);              // combines the same fields equals
}
```

`Objects.hash(...)` boxes and builds an array — fine for most classes. For hot paths, the hand-rolled form:

```
int h = Integer.hashCode(x);
h = 31 * h + Integer.hashCode(y);
h = 31 * h + (name == null ? 0 : name.hashCode());
return h;
```

Rules: use exactly the fields `equals` uses (a subset is legal; a superset breaks the contract); never use a field that is mutable if the object will be a map key; a constant `hashCode` is *legal* and turns every `HashMap` into a linked list.

Why the contract matters: `HashMap`

`map.put(key, v)` computes `key.hashCode()`, picks a bucket, and stores the entry. `map.get(k2)` computes `k2.hashCode()`, goes to *that* bucket, and compares with `equals`. If `k2.equals(key)` but their hash codes differ, the lookup searches the wrong bucket and finds nothing. Override `equals` without `hashCode` and your objects vanish from hash-based collections — a bug that appears as "contains returned false for something I just added". `HashSet`, `HashMap`, `Hashtable`, `ConcurrentHashMap` and `LinkedHashMap` all depend on it.

Mutating a key after insertion is the other classic: the entry sits in the bucket computed from the *old* hash. Keys should be immutable.

`toString`

Covered in Module 7: override it early, include identifying fields, keep it single-line, never rely on parsing it. Records generate `Point[x=1, y=2]`; IDEs generate the same shape. `Objects.toString(o, "default")` is the null-safe helper.

Records do all three

```
record Point(int x, int y) { }
```

A record (Module 10) generates `equals`, `hashCode` and `toString` over its components, correctly, with `Double.compare` semantics for doubles and `Arrays`-unaware equality for

array components (a reason not to put arrays in records). For value classes, records are the modern answer; the hand-written form above is for classes that cannot be records.

`getClass` and class literals

`obj.getClass()` returns the runtime `Class<?>`; `Point.class` is the compile-time literal for the same object. `getClass().getSimpleName()` is handy in `toString`. `getClass() == Point.class` is an exact-class test; `instanceof` includes subclasses.

`clone` — why you avoid it

`Object.clone()` is `protected`, performs a shallow copy, and throws `CloneNotSupportedException` unless the class implements the marker interface `Cloneable` — an API that fights the language (no constructor runs, `final` fields cannot be deep-copied). Arrays are the one place `clone()` is idiomatic. For everything else, a **copy constructor** or a static `copyOf` factory.

Common mistakes

Mistake	Consequence
`equals(Point)` instead of `equals(Object)`	Overload; collections use identity
`equals` without `hashCode`	Lost in hash-based collections
`hashCode` using fields `equals` ignores	Equal objects, different hashes
Comparing doubles with `==`	`NaN != NaN`; `0.0 == -0.0` differs from `Double.compare`
`instanceof` in `equals` with a stateful subclass	Broken symmetry
Mutable key fields	Entries become unreachable after mutation
`equals` that throws on null	Violates the contract; `Objects.equals` avoids it

Interview angle

"What is the `equals` / `hashCode` contract?" Equal objects must have equal hash codes; `equals` is reflexive, symmetric, transitive, consistent, false for null.

"What happens if you override `equals` only?" Hash-based collections fail to find your objects.

"Can two unequal objects have the same `hashCode`?" Yes — a collision.

"`getClass()` or `instanceof` in `equals`?" `getClass()` for strict symmetry with subclasses; `instanceof` for final classes or stateless subclasses.

"Why are strings good map keys?" Immutable, with cached, content-based hash codes.

Key takeaways

Override `equals(Object)`, `hashCode` and `toString` together for value-like classes; records do it for you.

`equals`: identity fast path, null/type check, field comparison with `==` / `Double.compare` / `Objects.equals`.

`hashCode` from the same fields (`Objects.hash`); equal ⇒ same hash, always.

Keys must be immutable; a mutated key is a lost entry.

Avoid `clone`; write copy constructors.

EXERCISE 1

Points in a set

Write `final class Point` with `int x, y`, a constructor, and correct `equals(Object)`, `hashCode()` and `toString()` (`(x, y)`). Read an integer `n` and `n` pairs `x y`; add each as a `Point` to a `HashSet<Point>`. Print `distinct=<set size>`, then `contains(0,0)=<true|false>`.

With a broken `equals` / `hashCode` the set would count duplicates separately — the hidden cases check that.

Example: `4` then `1 2`, `1 2`, `0 0`, `2 1` →

```
distinct=3
contains(0,0)=true
```

INPUT 1

```
4
1 2
1 2
0 0
2 1
```

OUTPUT 1

```
distinct=3
contains(0,0)=true
```

INPUT 2

```
3
5 5
5 5
5 5
```

OUTPUT 2

```
distinct=1
contains(0,0)=false
```

REFERENCE SOLUTION

```
import java.util.*;

final class Point {
    private final int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point p = (Point) o;
        return x == p.x && y == p.y;
    }

    @Override
    public int hashCode() { return Objects.hash(x, y); }

    @Override
    public String toString() { return "(" + x + ", " + y + ")"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Set<Point> set = new HashSet<>();
        for (int i = 0; i < n; i++) set.add(new Point(in.nextInt(), in.nextInt()));
        System.out.println("distinct=" + set.size());
        System.out.println("contains(0,0)=" + set.contains(new Point(0, 0)));
    }
}
```

EXERCISE 2

Money as a map key

Write a final `class Money` with `long cents` and `String currency`, a correct `equals(Object)`, a consistent `hashCode` and a `toString` like `12.50 USD`. Read an integer `n` and `n` lines `<cents> <currency>`. Count each distinct amount with a `HashMap<Money, Integer>` and print `<money> x<count>` in first-seen order (use a `LinkedHashMap`), then `distinct=<size>`. Also print `equal=<new Money(100,"USD").equals(new Money(100,"USD"))> sameHash=<their hash codes are equal>`.

Example: `3` then `1250 USD`, `1250 USD`, `1250 EUR` →

```
12.50 USD x2
12.50 EUR x1
distinct=2
equal=true sameHash=true
```

INPUT 1

```
3
1250 USD
1250 USD
1250 EUR
```

OUTPUT 1

```
12.50 USD x2
12.50 EUR x1
distinct=2
equal=true sameHash=true
```

INPUT 2

```
1
5 GBP
```

OUTPUT 2

```
0.05 GBP x1
distinct=1
equal=true sameHash=true
```

REFERENCE SOLUTION

```
import java.util.*;

final class Money {
    final long cents;
    final String currency;
    Money(long cents, String currency) { this.cents = cents; this.currency = currenc
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Money)) return false;
        Money m = (Money) o;
        return cents == m.cents && currency.equals(m.currency);
    }
    @Override public int hashCode() { return Objects.hash(cents, currency); }
    @Override public String toString() { return String.format(Locale.ROOT, "%d.%02d
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<Money, Integer> counts = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) {
            Money m = new Money(in.nextLong(), in.next());
            counts.merge(m, 1, Integer::sum);
        }
        counts.forEach((m, c) -> System.out.println(m + " x" + c));
        System.out.println("distinct=" + counts.size());
        Money a = new Money(100, "USD"), b = new Money(100, "USD");
        System.out.println("equal=" + a.equals(b) + " sameHash=" + (a.hashCode() ==
    }
}
```

QUIZ

1. You override `equals` but not `hashCode`. In a `HashSet`…

A. Everything works

B. Equal objects may land in different buckets and `contains` returns false for them

C. The set throws

D. Duplicates are rejected correctly

2. Two objects that are not `equals` …

A. Must have different hash codes

B. May have the same hash code — a collision

C. Cannot be in the same map

D. Must be different classes

3. How should two `double` fields be compared inside `equals`?

A. `a == b`

B. `Double.compare(a, b) == 0`

C. `a.equals(b)`

D. `Math.abs(a - b) < 1`

4. Why must the parameter of `equals` be `Object`?

A. Style

B. So it overrides `Object.equals` and is called by collections; a narrower parameter is an overload

C. For performance

D. The compiler requires it

5. Why are mutable objects poor `HashMap` keys?

A. They are slower

B. Changing a key after insertion changes its hash code, leaving the entry in the wrong bucket

C. They cannot override `hashCode`

D. They use more memory

ANSWER KEY · THE OBJECT CLASS — EQUALS, HASHCODE AND TOSTRING

1. B Hash-based collections locate the bucket by `hashCode` first. Unequal hashes for equal objects break the lookup.

2. B The contract only constrains equal objects. Collisions are expected and handled by the collection.

3. B `Double.compare` treats `NaN` as equal to itself and distinguishes `0.0` from `-0.0`, matching `Double.equals` and `hashCode`.

4. B `List.contains`, `HashMap.get` and `Objects.equals` all call `equals(Object)`.

5. B The entry was placed using the old hash. A later `get` computes the new hash and looks elsewhere.

LESSON 5 · 12 MIN

Abstract classes and the template method

Sometimes a parent class knows *that* something must be done but not *how*: every shape has an area, every report has a body, every account settles at month end — but only the concrete kind knows the formula. An **abstract class** declares those methods without bodies and forbids creating instances of itself, leaving subclasses to fill in the blanks. This lesson covers the rules, the template-method pattern that abstract classes exist for, and how to choose between an abstract class and an interface.

Declaring one

```
public abstract class Shape {
    private final String name;

    protected Shape(String name) { this.name = name; }

    public abstract double area();                    // no body: subclasses m
    public abstract double perimeter();

    public String describe() {                        // concrete: shared by a
        return String.format("%s with area %.2f", name, area());
    }
}

public class Circle extends Shape {
    private final double r;
    public Circle(double r) { super("circle"); this.r = r; }
    @Override public double area()      { return Math.PI * r * r; }
    @Override public double perimeter() { return 2 * Math.PI * r; }
}
```

Rules:

A class with any abstract method must be declared `abstract`; an abstract class may also have none (abstract only to prevent instantiation).

`new Shape("x")` is a compile error: abstract classes cannot be instantiated. `Shape s = new Circle(1)` is fine — the *type* is usable.

A concrete subclass must implement **every** inherited abstract method, or be abstract itself.

Abstract classes can have constructors (called via `super`), fields, concrete methods, static members, `private` members — everything a class can, minus instantiation.

Abstract methods cannot be `private`, `static` or `final` — each would make overriding impossible.

`describe()` calls `area()`, which does not exist yet in `Shape`. That is the trick: the parent writes an algorithm in terms of operations it leaves abstract, and dynamic dispatch fills them in at run time.

The template method pattern

```
public abstract class Report {
    public final String render() {                    // the fixed skeleton —
        StringBuilder sb = new StringBuilder();
        sb.append(header());
        sb.append(body());
        sb.append(footer());
        return sb.toString();
    }
    protected String header() { return "== " + title() + " ==\n"; }    // defau
    protected String footer() { return "\n-- end --\n"; }
    protected abstract String title();                 // required
    protected abstract String body();                  // required
}
```

`render()` is the **template**: the steps and their order are fixed; some steps have defaults (*hooks*), some must be supplied. Subclasses customise by overriding the pieces, never the skeleton. This is the primary use of abstract classes — sharing an algorithm's structure while varying its parts — and it appears throughout the JDK (`AbstractList` implements `contains`, `indexOf`, iteration and more on top of two abstract methods, `get(int)` and `size()`).

Partial implementations

`AbstractList`, `AbstractMap`, `AbstractSet` show the other face: skeletal implementations of an interface that turn "implement 25 methods" into "implement 2". If you write an interface

others will implement, providing an `AbstractX` next to it is a kindness (Module 9 covers when default methods do this job instead).

Abstract class or interface?

	Abstract class	Interface
Fields / state	Yes, including private	Only constants
Constructors	Yes	No
Multiple inheritance	No — one parent	Yes — implement many
Method bodies	Yes	Default/static/private methods (Java 8+), no state
Access levels	All	Public (and private helpers)
Says	"is a kind of" with shared implementation	"can do" — a capability

Choose an **abstract class** when subclasses share *state* and *code* and are conceptually variants of one thing (kinds of `Shape`, kinds of `Report`). Choose an **interface** when unrelated classes should share a *capability* (`Comparable`, `Runnable`, `Closeable`) or when you need multiple types on one class. Frequently both: an interface for the type, an abstract class as a convenient partial implementation (`List` + `AbstractList`).

Since Java 8 interfaces with default methods cover many cases that once needed abstract classes; the remaining distinctive power of an abstract class is **state plus constructors** — a template that owns fields.

Abstract classes and constructors

`Shape`'s constructor runs when a `Circle` is built (`super("circle")`). Abstract-class constructors exist to initialise the shared fields; they are usually `protected`, since only subclasses call them. The same warning as always: do not call abstract methods from the constructor — the subclass has not initialised its fields yet.

Abstract classes in APIs

Marking a class abstract is a commitment: it cannot be instantiated, so callers need a concrete subclass or a factory. Pairing an abstract class with static factories (`Shape.circle(1.0)`) hides the subclass names entirely and lets the implementation change later. `Calendar.getInstance()`, `NumberFormat.getInstance()` are this shape in the JDK.

Interview angle

"Can an abstract class have a constructor?" Yes — called by subclass constructors to set shared state.

"Can it have no abstract methods?" Yes; then `abstract` only prevents instantiation.

"Can an abstract method be static?" No — nothing could override it.

"Abstract class vs interface?" State and constructors vs multiple inheritance and capabilities.

"What is the template method pattern?" A final skeleton method calling abstract/hook steps that subclasses supply.

Key takeaways

`abstract` methods have no body; `abstract` classes cannot be instantiated; concrete subclasses must implement every abstract method.

Abstract classes carry state, constructors and concrete code — the vehicle for the template method pattern.

Fix the skeleton (`final`), vary the steps (abstract and hook methods).

Shared state and implementation → abstract class; capability across unrelated types → interface; often both.

EXERCISE 1

A report template

Write `abstract class Report` with a `final String render()` that concatenates `header()`, `body()` and `footer()`; `header()` returns `"== " + title() + " ==\n"` and `footer()` returns `"-- end --\n"` by default; `title()` and `body()` are abstract. Write `SalesReport` (title `Sales`, body:

one line per item `<name>: <amount>`) and `InventoryReport` (title `Inventory`, body: `items=<count>` on one line, and it overrides `footer()` to return `"-- counted --\n"`).

Read a word `sales` or `inventory`, an integer `n`, then `n` lines `<name> <amount>`; build the matching report and print `render()`.

Example: `sales`, `2`, `pen 10`, `pad 5` →

```
== Sales ==
pen: 10
pad: 5
-- end --
```

INPUT 1

```
sales
2
pen 10
pad 5
```

OUTPUT 1

```
== Sales ==
pen: 10
pad: 5
-- end --
```

INPUT 2

```
inventory
3
a 1
b 2
c 3
```

OUTPUT 2

```
== Inventory ==
items=3
-- counted --
```

REFERENCE SOLUTION

```
import java.util.*;

abstract class Report {
    final String render() {
        return header() + body() + footer();
    }
    String header() { return "== " + title() + " ==\n"; }
    String footer() { return "-- end --\n"; }
    abstract String title();
    abstract String body();
}

class SalesReport extends Report {
    private final String[] names;
    private final long[] amounts;
    SalesReport(String[] names, long[] amounts) { this.names = names; this.amounts =
    @Override String title() { return "Sales"; }
    @Override String body() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < names.length; i++) sb.append(names[i]).append(": ").appe
        return sb.toString();
    }
}

class InventoryReport extends Report {
    private final int count;
    InventoryReport(int count) { this.count = count; }
    @Override String title() { return "Inventory"; }
    @Override String body() { return "items=" + count + "\n"; }
    @Override String footer() { return "-- counted --\n"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String kind = in.next();
        int n = in.nextInt();
        String[] names = new String[n];
        long[] amounts = new long[n];
        for (int i = 0; i < n; i++) { names[i] = in.next(); amounts[i] = in.nextLong
        Report r = kind.equals("sales") ? new SalesReport(names, amounts) : new Inve
        System.out.print(r.render());
    }
}
```

EXERCISE 2

A template for reports

Write `abstract class Shape` with a final template method `String describe()` that returns `<name()>: area=<area() to two decimals>, perimeter=<perimeter() to two decimals>`, calling three abstract methods `name()`, `area()` and `perimeter()`. Implement `Circle(r)`, `Square(s)` and `Rect(w, h)`. Read `n` shape lines (`circle r`, `square s`, `rect w h`) and print `describe()` for each, then `largest=<name of the largest area>`.

Example: `2` then `circle 1`, `square 2` →

```
circle: area=3.14, perimeter=6.28
square: area=4.00, perimeter=8.00
largest=square
```

INPUT 1

```
2
circle 1
square 2
```

OUTPUT 1

```
circle: area=3.14, perimeter=6.28
square: area=4.00, perimeter=8.00
largest=square
```

INPUT 2

```
1
rect 2 3.5
```

OUTPUT 2

```
rect: area=7.00, perimeter=11.00
largest=rect
```

REFERENCE SOLUTION

```java
import java.util.*;

abstract class Shape {
    abstract String name();
    abstract double area();
    abstract double perimeter();
    final String describe() {
        return String.format(Locale.ROOT, "%s: area=%.2f, perimeter=%.2f", name(), a
    }
}
class Circle extends Shape {
    final double r;
    Circle(double r) { this.r = r; }
    String name() { return "circle"; }
    double area() { return Math.PI * r * r; }
    double perimeter() { return 2 * Math.PI * r; }
}
class Square extends Shape {
    final double s;
    Square(double s) { this.s = s; }
    String name() { return "square"; }
    double area() { return s * s; }
    double perimeter() { return 4 * s; }
}
class Rect extends Shape {
    final double w, h;
    Rect(double w, double h) { this.w = w; this.h = h; }
    String name() { return "rect"; }
    double area() { return w * h; }
    double perimeter() { return 2 * (w + h); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Shape largest = null;
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            Shape s = switch (kind) {
                case "circle" -> new Circle(in.nextDouble());
                case "square" -> new Square(in.nextDouble());
                default -> new Rect(in.nextDouble(), in.nextDouble());
            };
            System.out.println(s.describe());
            if (largest == null || s.area() > largest.area()) largest = s;
        }
        System.out.println("largest=" + (largest == null ? "none" : largest.name()))
```

```
    }
}
```

QUIZ

1. `new Shape()` where `Shape` is abstract…

A. Creates a default shape

B. Is a compile error

C. Throws at run time

D. Creates an anonymous subclass

2. A concrete subclass of an abstract class must…

A. Override every method

B. Implement every inherited abstract method

C. Declare a no-arg constructor

D. Be final

3. Can an abstract class have a constructor?

A. No

B. Yes — subclass constructors call it via `super(...)` to initialise shared state

C. Only a private one

D. Only if it has no abstract methods

4. In the template method pattern the skeleton method is typically…

A. Abstract

B. `final`, so subclasses vary the steps but not the order

C. Static

D. Private

5. Choose an abstract class over an interface when…

A. You need multiple inheritance

B. Subclasses share state and implementation, and are variants of one concept

C. The type is a capability like `Comparable`

D. Never

ANSWER KEY · ABSTRACT CLASSES AND THE TEMPLATE METHOD

1. B Abstract classes cannot be instantiated directly; only concrete subclasses can (or an anonymous subclass with a body).

2. B Otherwise it must itself be declared abstract.

3. B It cannot be called with `new` directly, but it runs as part of every subclass construction.

4. B The fixed algorithm calls abstract and hook methods; making it final protects the structure.

5. B Fields and constructors are what interfaces lack; capabilities across unrelated classes are what interfaces are for.

LESSON 6 · 13 MIN

When not to inherit — composition, Liskov and final

Inheritance is the most overused feature in object-oriented languages. It is easy to reach for because it saves typing, and it creates the tightest coupling Java offers: a subclass depends on the parent's *implementation*, not just its API, and every change to the parent risks breaking it. This lesson gives you the tests for when inheritance is right, the pattern to use when it is not, and the tools — `final` and `sealed` — for controlling who may extend what.

The is-a test, taken seriously

`extends` should mean the subclass **is** a parent, in every context the parent is used. The **Liskov substitution principle** makes this precise: any code correct for the parent must stay correct when handed a subclass. Violations are subtle:

```
class Rectangle { void setWidth(int w); void setHeight(int h); int area(); }
class Square extends Rectangle {          // a square IS a rectangle, mathemat
    @Override void setWidth(int w)  { super.setWidth(w); super.setHeight(w); }
    @Override void setHeight(int h) { super.setWidth(h); super.setHeight(h); }
}
Rectangle r = new Square();
r.setWidth(5); r.setHeight(3);
r.area();       // 9 — code that assumed width and height are independent is r
```

The subclass kept the *signatures* and broke the *contract*. Substitutability is about behaviour: preconditions no stronger, postconditions no weaker, invariants preserved. If a subclass must throw `UnsupportedOperationException` from an inherited method, or narrow what the parent accepted, it is not a subtype.

The fragile base class problem

```
class InstrumentedSet<E> extends HashSet<E> {
    private int added = 0;
    @Override public boolean add(E e) { added++; return super.add(e); }
    @Override public boolean addAll(Collection<? extends E> c) { added += c.si
}
new InstrumentedSet<String>().addAll(List.of("a", "b", "c"));   // added == 6
```

`HashSet.addAll` calls `add` for each element internally, so the count doubles. The subclass depended on a detail of the parent's implementation that the parent never promised — and if a JDK update changed `addAll`, the subclass would break or silently change behaviour. This is *Effective Java* item 18: inheritance across package boundaries is fragile because the subclass sees the parent's method calls as part of its contract.

Composition: the robust alternative

```
class InstrumentedSet<E> {
    private final Set<E> inner;          // HAS a set
    private int added = 0;
    InstrumentedSet(Set<E> inner) { this.inner = inner; }

    public boolean add(E e) { added++; return inner.add(e); }
    public boolean addAll(Collection<? extends E> c) {
        int before = inner.size();
        boolean changed = inner.addAll(c);
        added += inner.size() - before;
        return changed;
    }
    public int added() { return added; }
    // forward the rest of the Set methods as needed…
}
```

The wrapper *holds* a set and forwards calls. It depends only on `Set`'s public contract; it can wrap *any* `Set` implementation; and no internal self-call of the wrapped object can reach it. The cost is the forwarding boilerplate (an `AbstractSet`-style forwarding class, or an IDE's "delegate methods", absorbs it). This is the **decorator** pattern; `Collections.unmodifiableSet`, `BufferedReader` wrapping a `Reader`, and `synchronizedList` are all composition.

Rule of thumb: extend only when there is a genuine is-a *and* the parent was **designed for extension** (documented hooks, protected methods, no self-use surprises). Otherwise

compose.

Designing for extension — or forbidding it

If a class is meant to be extended, its author must document which methods call which (`addAll` calls `add`), provide protected hooks deliberately, avoid calling overridable methods from constructors, and test with actual subclasses. That is real work, so the default for classes not designed for it is to **prohibit** extension:

`final class` — no subclasses. Values, utility classes, security-sensitive types.

`final` methods — subclasses exist but this step is fixed (template skeletons).

Private constructors with static factories — extension impossible without a callable constructor.

`String`, `Integer`, `LocalDate` and every record are final; `HashMap` is not, and the JDK carries the burden of documenting its self-use.

Sealed classes (Java 17)

Between "anyone may extend" and "no one may" sits **sealed**: a class or interface that names exactly which types may extend it.

```
public sealed abstract class Shape permits Circle, Square, Triangle { … }
public final class Circle extends Shape { … }
public final class Square extends Shape { … }
public non-sealed class Triangle extends Shape { … }    // this branch is oper
```

Each permitted subclass must be `final`, `sealed` or `non-sealed`, and must be in the same module (or package, if unnamed). The payoff: the compiler *knows* the complete set of subtypes, so a `switch` over a `Shape` with pattern matching (Java 21) is exhaustive without a `default`, and adding a fourth shape is a compile error at every such switch — the same safety enums give, for class hierarchies. Sealed types model closed sets of variants (AST nodes, states, results); Module 9 covers sealed *interfaces*, the more common form.

Inheritance done right

Inheritance is not evil; it is specific. Good uses:

A genuine is-a with shared state and a template: `Shape` / `Circle`, `AbstractList` / `ArrayList`, `Exception` hierarchies (`IOException extends Exception`).

Frameworks designed for it: `HttpServlet.doGet`, JUnit base classes (historically), `Thread` (though `Runnable` composition is preferred).

Sealed hierarchies modelling closed variant sets.

Warning signs: a subclass overriding most parent methods; a subclass that needs a parent's private field; `super` calls whose omission changes behaviour mysteriously; subclasses of classes you do not control; class names that are adjectives on the parent (`FastList`, `BetterMap`).

A decision procedure

Is it truly *is-a*, substitutable everywhere the parent is used? No → compose.

Do you control the parent, or is it documented for extension? No → compose.

Do you need to reuse the parent's *implementation* only, not be its *type*? Yes → compose (or a private static helper).

Is the set of subtypes closed and known? Yes → sealed.

Is the class a value or utility? Yes → `final`.

Interview angle

"Composition vs inheritance?" Has-a vs is-a; composition depends on the public contract only, can wrap any implementation, and avoids the fragile base class problem.

"What is the Liskov substitution principle?" Subtypes must be usable wherever the supertype is, without breaking correctness — behaviour, not just signatures.

"Why make a class `final`?" It was not designed for extension; values must stay immutable; security.

"What does `sealed` add?" A closed, compiler-known set of subtypes — exhaustive switches, safe modelling of variants.

"Give an example of the fragile base class problem." `HashSet.addAll` calling `add` and double-counting in a subclass.

Key takeaways

`extends` only for a real is-a that is substitutable in behaviour, with a parent designed for extension.

Composition (wrap and forward) depends only on public contracts and avoids fragile-base-class breakage.

`final` by default for classes not designed for extension; `final` methods for fixed template steps.

`sealed … permits` models closed sets of subtypes with compiler-checked exhaustiveness.

EXERCISE 1

Wrap, don't extend

The subclass below double-counts because `HashSet.addAll` calls `add` internally. Rewrite `CountingSet` as a **wrapper** that holds a `Set<String>` and forwards `add`, `addAll`, `contains` and `size`, counting attempted additions correctly (`addAll` adds the collection's size once).

Read an integer `n` and `n` commands: `add <w>`, `addAll <k> <w1> … <wk>`, `has <w>`, `size`, `attempts`. Print the result of each `has` / `size` / `attempts`.

Example: `4` then `add a`, `addAll 2 a b`, `size`, `attempts` →

```
2
3
```

INPUT 1

```
4
add a
addAll 2 a b
size
attempts
```

OUTPUT 1

```
2
3
```

INPUT 2

```
5
addAll 3 x y z
has y
has q
size
attempts
```

OUTPUT 2

```
true
false
3
3
```

REFERENCE SOLUTION

```
import java.util.*;

class CountingSet {
    private final Set<String> inner = new HashSet<>();
    private int attempts = 0;

    boolean add(String s) {
        attempts++;
        return inner.add(s);
    }

    boolean addAll(Collection<? extends String> c) {
        attempts += c.size();
        return inner.addAll(c);
    }

    boolean contains(String s) { return inner.contains(s); }
    int size() { return inner.size(); }
    int attempts() { return attempts; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        CountingSet set = new CountingSet();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "add" -> set.add(in.next());
                case "addAll" -> {
                    int k = in.nextInt();
                    List<String> words = new ArrayList<>();
                    for (int j = 0; j < k; j++) words.add(in.next());
                    set.addAll(words);
                }
                case "has" -> System.out.println(set.contains(in.next()));
                case "size" -> System.out.println(set.size());
                default -> System.out.println(set.attempts());
            }
        }
    }
}
```

EXERCISE 2

A stack that wraps a list

Do **not** extend `ArrayList`. Write `class Stack<T>` that holds a private `ArrayList<T>` and exposes only `push`, `pop` (returns `null` when empty), `peek` (`null` when empty), `size` and `isEmpty`. Read an integer `n` and `n` commands `push <word>`, `pop`, `peek`, `size`; print the result of every command except `push` (`empty` for a null). The point: a client can never call `get(0)` or `add(0, x)` on your stack, which `extends ArrayList` would have allowed.

Example: `5` then `push a`, `push b`, `peek`, `pop`, `size` →

```
b
b
1
```

INPUT 1

```
5
push a
push b
peek
pop
size
```

OUTPUT 1

```
b
b
1
```

INPUT 2

```
3
pop
peek
size
```

OUTPUT 2

```
empty
empty
0
```

REFERENCE SOLUTION

```java
import java.util.*;

class Stack<T> {
    private final ArrayList<T> items = new ArrayList<>();
    void push(T x) { items.add(x); }
    T pop() { return items.isEmpty() ? null : items.remove(items.size() - 1); }
    T peek() { return items.isEmpty() ? null : items.get(items.size() - 1); }
    int size() { return items.size(); }
    boolean isEmpty() { return items.isEmpty(); }
}

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

QUIZ

1. The Liskov substitution principle says…

A. Subclasses must have the same fields

B. Code correct for the supertype must remain correct for any subtype

C. Every class needs an interface

D. Inheritance depth must be one

2. The fragile base class problem occurs when…

A. A class is too small

B. A subclass depends on the parent's internal self-calls, which a parent change can break

C. Two classes share a name

D. A class has no constructor

3. Composition (wrap and forward) is more robust than inheritance because…

A. It is shorter

B. It depends only on the wrapped object's public contract and can wrap any implementation

C. It disables polymorphism

D. It is faster

4. A `sealed` class…

A. Cannot have subclasses

B. Lists exactly which classes may extend it, enabling exhaustive switches

C. Is the same as `final`

D. Cannot be abstract

5. Marking a class `final` is appropriate when…

A. It has many methods

B. It is a value type or was not designed for extension

C. It is abstract

D. It implements an interface

ANSWER KEY · WHEN NOT TO INHERIT — COMPOSITION, LISKOV AND FINAL

1. B Substitutability is behavioural: a `Square` that breaks `Rectangle`'s independent width/height contract is not a valid subtype even if it compiles.

2. B `HashSet.addAll` calling `add` is the canonical example — the subclass's counter doubles.

3. B The wrapper never sees the inner object's internal calls, and the inner object can be swapped.

4. B `permits` names the closed set; each permitted subclass is `final`, `sealed` or `non-sealed`.

5. B Designing for extension is real work; a class not designed for it should prohibit it, and immutability requires it.

CHECKPOINT · 24 MIN

Checkpoint — Inheritance & polymorphism

This checkpoint covers `extends` and constructor chaining, the rules of overriding and hiding, dynamic dispatch and casting, the `Object` methods and their contract, abstract classes and the template method, and the composition-versus-inheritance judgement.

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

Before you start, make sure you can answer:

- What the first statement of every constructor is, and why `class B extends A {}` can fail to compile.
- The rules for a valid override, and why `equals(Point)` is wrong.
- What runs for `Account a = new SavingsAccount(); a.monthEnd();`, and for a static method called the same way.
- What happens on a failed downcast, and what `instanceof` says about `null`.
- The `equals` / `hashCode` contract and what breaks if you override only one.
- What a template method is, and one reason to prefer composition.

The programs are a small class hierarchy with polymorphic dispatch, and a value class whose `equals` / `hashCode` are exercised through a `HashSet`.

EXERCISE 1

Accounts at month end

Write `abstract class Account` with an owner, a balance in cents, `deposit(long)`, `balance()`, and an abstract `monthEnd()`. `Savings` adds interest at month end: `balance += balance * rateBasisPoints / 10000` (integer arithmetic, rate given in basis points). `Checking` charges a fee: `balance -= fee` (may go negative). Print with `toString()` as `<owner> <kind> <balance>` where kind is `savings` or `checking`.

Read an integer `n` and `n` lines `savings <owner> <balance> <bp>` or `checking <owner> <balance> <fee>`; store them in an `Account[]`, call `monthEnd()` on every element polymorphically, then print each.

Example: `2` then `savings ada 100000 250`, `checking bob 5000 700` →

```
ada savings 102500
bob checking 4300
```

INPUT 1

```
2
savings ada 100000 250
checking bob 5000 700
```

OUTPUT 1

```
ada savings 102500
bob checking 4300
```

INPUT 2

```
1
checking c 100 250
```

OUTPUT 2

```
c checking -150
```

REFERENCE SOLUTION

```java
import java.util.*;

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

    Account(String owner, long cents) {
        this.owner = owner;
        this.cents = cents;
    }

    void deposit(long c) { cents += c; }
    long balance() { return cents; }
    abstract void monthEnd();
    abstract String kind();

    @Override
    public String toString() { return owner + " " + kind() + " " + cents; }
}

class Savings extends Account {
    private final long bp;
    Savings(String owner, long cents, long bp) { super(owner, cents); this.bp = bp;
    @Override void monthEnd() { cents += cents * bp / 10000; }
    @Override String kind() { return "savings"; }
}

class Checking extends Account {
    private final long fee;
    Checking(String owner, long cents, long fee) { super(owner, cents); this.fee = f
    @Override void monthEnd() { cents -= fee; }
    @Override String kind() { return "checking"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Account[] accounts = new Account[n];
        for (int i = 0; i < n; i++) {
            String type = in.next();
            String owner = in.next();
            long cents = in.nextLong();
            long extra = in.nextLong();
            accounts[i] = type.equals("savings") ? new Savings(owner, cents, extra)
        }
        for (Account a : accounts) a.monthEnd();
        for (Account a : accounts) System.out.println(a);
```

```
    }
}
```

EXERCISE 2

Distinct versions

Write `final class Version` with `int major, minor, patch`, a static factory `parse(String)` for `"1.2.3"`, correct `equals` / `hashCode`, and `toString` giving `major.minor.patch`. Read an integer `n` and `n` version strings. Print `distinct=<number of distinct versions>` (use a `HashSet`) and then the distinct versions in **ascending** order, one per line — implement `Comparable<Version>` comparing major, then minor, then patch, and sort with `Collections.sort`.

Example: `4` then `1.2.3 1.10.0 1.2.3 0.9.9` →

```
distinct=3
0.9.9
1.2.3
1.10.0
```

INPUT 1

```
4
1.2.3 1.10.0 1.2.3 0.9.9
```

OUTPUT 1

```
distinct=3
0.9.9
1.2.3
1.10.0
```

INPUT 2

```
2
2.0.0 2.0.0
```

OUTPUT 2

```
distinct=1
2.0.0
```

REFERENCE SOLUTION

```
import java.util.*;

final class Version implements Comparable<Version> {
    private final int major, minor, patch;
    private Version(int major, int minor, int patch) { this.major = major; this.minc

    static Version parse(String s) {
        String[] p = s.split("\\.");
        return new Version(Integer.parseInt(p[0]), Integer.parseInt(p[1]), Integer.p
    }

    @Override
    public int compareTo(Version o) {
        if (major != o.major) return Integer.compare(major, o.major);
        if (minor != o.minor) return Integer.compare(minor, o.minor);
        return Integer.compare(patch, o.patch);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Version)) return false;
        Version v = (Version) o;
        return major == v.major && minor == v.minor && patch == v.patch;
    }

    @Override
    public int hashCode() { return Objects.hash(major, minor, patch); }

    @Override
    public String toString() { return major + "." + minor + "." + patch; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Set<Version> set = new HashSet<>();
        for (int i = 0; i < n; i++) set.add(Version.parse(in.next()));
        List<Version> sorted = new ArrayList<>(set);
        Collections.sort(sorted);
        System.out.println("distinct=" + set.size());
        for (Version v : sorted) System.out.println(v);
    }
}
```

EXERCISE 3

Payroll hierarchy

Write `abstract class Employee` (`name`, abstract `double pay()`, `String kind()`) with `Salaried(name, annual)` paying `annual / 12`, `Hourly(name, rate, hours)` paying `rate × hours` with hours over 160 at 1.5×, and `Commissioned(name, base, sales)` paying `base + 5% of sales`. Read `n` lines, sort the employees by pay **descending** (ties by name), print `<name> (<kind>) <pay to two decimals>` per line, then `payroll <total>`.

Example: `3` then `salaried ann 60000`, `hourly bob 20 170`, `commissioned cy 1000 50000` →

```
ann (salaried) 5000.00
bob (hourly) 3500.00
cy (commissioned) 3500.00
payroll 12000.00
```

INPUT 1

```
3
salaried ann 60000
hourly bob 20 170
commissioned cy 1000 50000
```

OUTPUT 1

```
ann (salaried) 5000.00
bob (hourly) 3500.00
cy (commissioned) 3500.00
payroll 12000.00
```

INPUT 2

```
1
hourly zed 10 100
```

OUTPUT 2

```
zed (hourly) 1000.00
payroll 1000.00
```

REFERENCE SOLUTION

```java
import java.util.*;

abstract class Employee {
    final String name;
    Employee(String name) { this.name = name; }
    abstract double pay();
    abstract String kind();
}
class Salaried extends Employee {
    final double annual;
    Salaried(String name, double annual) { super(name); this.annual = annual; }
    double pay() { return annual / 12; }
    String kind() { return "salaried"; }
}
class Hourly extends Employee {
    final double rate, hours;
    Hourly(String name, double rate, double hours) { super(name); this.rate = rate;
    double pay() { return hours ⩽ 160 ? rate * hours : rate * 160 + rate * 1.5 * (h
    String kind() { return "hourly"; }
}
class Commissioned extends Employee {
    final double base, sales;
    Commissioned(String name, double base, double sales) { super(name); this.base =
    double pay() { return base + 0.05 * sales; }
    String kind() { return "commissioned"; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Employee> staff = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String kind = in.next();
            staff.add(switch (kind) {
                case "salaried" → new Salaried(in.next(), in.nextDouble());
                case "hourly" → new Hourly(in.next(), in.nextDouble(), in.nextDoubl
                default → new Commissioned(in.next(), in.nextDouble(), in.nextDoubl
            });
        }
        staff.sort(Comparator.comparingDouble(Employee::pay).reversed().thenComparin
        double total = 0;
        for (Employee e : staff) {
            total += e.pay();
            System.out.println(String.format(Locale.ROOT, "%s (%s) %.2f", e.name, e.
        }
        System.out.println(String.format(Locale.ROOT, "payroll %.2f", total));
```

```
    }
}
```

CHECKPOINT QUESTIONS

1. Which is inherited by a subclass?

A. Constructors

B. Private fields, by name

C. Public and protected methods

D. Static initialiser blocks as overridable methods

2. `class B extends A { B() { System.out.println("B"); super(); } }` is…

A. Valid

B. A compile error — `super()` must be the first statement

C. Valid but prints in reverse

D. Valid only if A is abstract

3. An override declares `throws Exception` where the parent declared `throws IOException`. Result?

A. Valid

B. Compile error: broader checked exception

C. Valid with a warning

D. Valid if caught

4. `A a = new B(); a.field` where both declare `field`…

A. B's field — fields are polymorphic

B. A's field — fields are resolved by the reference type

C. Compile error

D. Both

5. Dynamic dispatch selects a method based on…

A. The reference's declared type

B. The object's runtime class

C. The argument types

D. The calling class

6. `Object o = "text"; Integer i = (Integer) o;`

A. Compiles and runs

B. Compiles; throws `ClassCastException` at run time

C. Compile error

D. Converts the string to a number

7. Which `equals` type check keeps symmetry when subclasses add state?

A. `o instanceof Point`

B. `getClass() ≠ o.getClass()`

C. `o.equals(this)`

D. No check

8. `Objects.hash(x, y)` is used in `hashCode` so that…

A. It is unique

B. Equal objects (same x and y) get equal hash codes

C. It is faster than arithmetic

D. It works for null objects only

9. An abstract class with **no** abstract methods…

A. Is a compile error

B. Is legal — it simply cannot be instantiated

C. Becomes an interface

D. Is automatically final

10. In the template method pattern, a *hook* is…

A. An abstract method

B. A concrete method with a default that subclasses may override

C. A static method

D. A constructor

11. `Square extends Rectangle` with independent `setWidth` / `setHeight` violates…

A. Encapsulation

B. The Liskov substitution principle

C. Single inheritance

D. The `equals` contract

12. A wrapper class that forwards to an inner `Set` and adds counting is an example of…

A. Inheritance

B. The decorator/composition pattern

C. A singleton

D. Overloading

13. Permitted subclasses of a sealed class must be…

A. Abstract

B. `final`, `sealed` or `non-sealed`

C. In different packages

D. Interfaces

14. Which is compile-time polymorphism and which is run-time?

A. Overriding is compile-time; overloading is run-time

B. Overloading is compile-time; overriding is run-time

C. Both compile-time

D. Both run-time

ANSWER KEY · CHECKPOINT — INHERITANCE & POLYMORPHISM

1. C Constructors are never inherited; private members exist but are inaccessible; statics are hidden, not overridden.

2. B Parent construction must complete before the child's body runs, so the call must come first.

3. B Callers holding the parent type only handle `IOException`; a broader checked exception would escape their handlers.

4. B Fields are hidden, never overridden. This is one reason fields should be private.

5. B The JVM's `invokevirtual` looks up the most specific override in the object's class chain at each call.

6. B `Object` might hold an `Integer`, so the compiler allows the cast; the JVM finds a `String` and throws.

7. B Exact-class comparison makes a `Point` and a `ColorPoint` unequal in both directions. `instanceof` is fine for final classes.

8. B Deriving the hash from exactly the fields `equals` compares is what satisfies the contract.

9. B Sometimes the point is only to prevent `new` on a base class meant as a common parent.

10. B Hooks give optional customisation points; abstract methods give required ones.

11. B Code that sets width and height independently on a `Rectangle` breaks when handed a `Square`.

12. B The wrapper has-a set and depends only on the `Set` contract — immune to the inner class's self-calls.

13. B Each branch must declare whether the hierarchy stays closed there, continues sealed, or reopens.

14. B Overloads are chosen by the compiler from argument types; overrides by the JVM from the receiver's class.

CodeKairo · Java study plan · Module 8: OOP II — inheritance & polymorphism. Solve the exercises online to track your progress and earn XP.