Inheritance

Inheritance in Java; extends, the constructor chain, method overriding, field hiding, upcasting/downcasting, and composition vs inheritance.

Intermediate 50 min

Last updated: November 9, 2026
Read online: https://www.learnforgex.com/en/topics/inheritance

---

Inheritance

The Abstract Class and Interface lessons both leaned on a mechanism we never stopped to formally define: `extends`, `super`, `@Override`. Now we'll dig into that mechanism — **inheritance** — from the ground up. We'll cover how a class inherits another class's fields and behavior, how the constructor chain actually works, why the `Object` class is the common ancestor of every Java object, and finally the question of "when should you reach for inheritance, and when for composition?"

What Is Inheritance?

Inheritance is a mechanism that lets one class (the subclass) acquire the fields and methods of another class (the superclass). A real-world example: every `Dog` **is an** `Animal` — a dog's properties (name, age) and behavior (sleeping) really belong to the concept of "animal," not to dogs specifically. In Java you establish this relationship with the `extends` keyword:

```
class Animal {
    String name;
}

class Dog extends Animal {
    // Dog automatically inherits Animal's "name" field
}
```

This is called an **"is-a" relationship** — "Dog is an Animal" is a meaningful, true sentence. It's fundamentally different from the **"has-a"** relationship we'll meet later in "Inheritance vs. Composition" (a `Car` "has" an `Engine`, but a `Car` "is not" an `Engine`).

Why Does It Exist?

Say you write `Student`, `Teacher`, and `Admin` classes separately — all three have `name` and `email` fields plus a `login()` behavior. Copying those fields and that method into all three classes means both duplication and the risk of having to update three places whenever a bug needs fixing. Inheritance lets you define that shared part **once** in a `Person` superclass, then get it for free in all three by writing `Student extends Person`, `Teacher extends Person`, and `Admin extends Person` — each subclass only adds what's specific to it (say, `Student`'s `enrolledCourses`).

History

Inheritance isn't unique to Java — it traces back to the Simula 67 language in 1967, and from there to 1970s Smalltalk; it's one of the foundational pillars of object-oriented programming (OOP). A critical design decision in Java's first release in 1996 was to support only **single inheritance** for classes — a class can `extends` only one superclass. This was a deliberate reaction to the **multiple inheritance** C++ supported at the time: when two superclasses in C++ contributed a field or method with the same name, the resulting ambiguity (the **Diamond Problem**, which we'll cover in detail in "Why Java Doesn't Support Multiple Inheritance") caused serious tangles in C++ codebases. Java's solution: single inheritance for classes, but — as we saw in the Interface lesson — unlimited multiple "implementation" for interfaces. That split kept the power of inheritance while removing the ambiguity.

Creating a Subclass and Basic Terminology

A subclass is declared by adding `extends SuperclassName` to a `class` declaration. Both sides of this relationship go by more than one name, and all of them are synonyms — no matter which term a given source uses, it's talking about the same thing:

- The superclass: also called the **parent class** or **base class**
- The subclass: also called the **child class** or **derived class**

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void eat() {
        System.out.println(name + " is eating");
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    void bark() {
        System.out.println(name + " says: Woof!");
    }
}

class FirstInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Rex");
        dog.eat();  // inherited from Animal
        dog.bark(); // Dog's own method
    }
}
```

Expected Output

```
Rex is eating
Rex says: Woof!
```

`Dog` extends `Animal` and inherits its `name` field and `eat()` method without writing either — and adds its own behavior, `bark()`, on top. A `Dog` object has both `Animal`'s members and its own — inheritance is about **sharing** code, not **copying** it.

Constructors and super()

The first line of a subclass constructor — whether written explicitly or left implicit — always calls a constructor of the superclass; Java doesn't bend on this rule. The `super(...)` call is how you make that invocation **explicit**, and it can only appear as the **first statement** of a subclass constructor:

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
        System.out.println("Vehicle constructor running for " + brand);
    }
}

class Car extends Vehicle {
    int doors;

    Car(String brand, int doors) {
        super(brand); // must be the first statement
        this.doors = doors;
        System.out.println("Car constructor running, doors=" + doors);
    }
}

class ConstructorChainExample {
    public static void main(String[] args) {
        Car car = new Car("Toyota", 4);
        // Output order proves Vehicle's constructor runs FIRST:
        // Vehicle constructor running for Toyota
        // Car constructor running, doors=4
    }
}
```

Expected Output

```
Vehicle constructor running for Toyota
Car constructor running, doors=4
```

Notice the order of the output: `Vehicle`'s constructor runs **before** `Car`'s. That ordering isn't a coincidence, it's a requirement — the superclass's state has to be fully established before the subclass does its own extra work; otherwise `Car`'s constructor would be building on top of `Vehicle` state that doesn't exist yet.

> If you never write `super(...)` in a subclass constructor, the compiler implicitly tries to call the superclass's **no-argument** constructor. If the superclass has no no-argument constructor (like `Vehicle` in this section), that implicit call fails to compile, and you're forced to call `super(...)` explicitly with the right arguments.

Method Overriding

A subclass can supply its own behavior for an inherited method by **redefining it with the exact same signature** — this is called **method overriding**. The `@Override` annotation isn't required, but it's strongly recommended: if you get the signature wrong (say, by changing a parameter's type), it lets the compiler catch that immediately.

```
class Shape {
    double area() {
        return 0.0;
    }
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double width;
    double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double area() {
        return width * height;
    }
}

class MethodOverridingExample {
    static void describe(Shape shape) {
        System.out.println("Area: " + shape.area());
    }

    public static void main(String[] args) {
        describe(new Circle(2));       // Area: 12.566370614359172
        describe(new Rectangle(3, 4)); // Area: 12.0
    }
}
```

`Circle` and `Rectangle` each override the `area()` they inherit from `Shape` with their own formula. The loop inside `describe()` only ever sees a `Shape` — it never knows which concrete class it's holding — yet calling `area()` still runs the right formula. Which implementation runs is decided at runtime, based on the object's **actual** class. This is called **dynamic dispatch**, and we'll come back to it in "Upcasting."

The super Keyword

The `super` keyword has three distinct uses, and we can see all of them in a single example: `super(...)` calls the superclass's constructor (as we saw in "Constructors and super()"), `super.method()` explicitly calls an **overridden** method from the superclass, and `super.field` accesses a superclass field directly:

```
class Employee {
    protected String name;
    protected double baseSalary;

    Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    String describe() {
        return name + " earns " + baseSalary;
    }
}

class Manager extends Employee {
    double bonus;

    Manager(String name, double baseSalary, double bonus) {
        super(name, baseSalary); // super(...) -- calls Employee's constructor
        this.bonus = bonus;
    }

    @Override
    String describe() {
        // super.describe() -- calls Employee's original describe(), builds on top of it
        return super.describe() + " plus a bonus of " + bonus;
    }

    double totalCompensation() {
        // super.baseSalary -- direct access to the inherited field
        return super.baseSalary + bonus;
    }
}

class SuperKeywordExample {
    public static void main(String[] args) {
        Manager manager = new Manager("Ada", 5000, 1000);
        System.out.println(manager.describe());           // Ada earns 5000.0 plus a bonus
of 1000.0
        System.out.println(manager.totalCompensation());  // 6000.0
    }
}
```

Before adding its own extra information, `Manager`'s `describe()` calls `super.describe()` — so it doesn't throw away `Employee`'s original behavior, it builds on top of it. This is a very common pattern in overriding: instead of rewriting the superclass's behavior from scratch, call it via `super.method()` and append to the result.

> `super.field` is used to disambiguate a field from a same-named `this.field` — but that's only ever necessary if the subclass has defined a **new field with the same name** as one in the superclass (field hiding); we'll cover that in detail in the next section, "Field Hiding vs. Method Overriding."

How Access Modifiers Affect Inheritance

A superclass's fields and methods are visible to a subclass to varying degrees, depending on their access modifier:

- `public`: Accessible from anywhere, and therefore from every subclass.
- `protected`: Accessible from the same package **and** from subclasses in other packages — a visibility level specifically designed for inheritance.
- **package-private** (no modifier at all): Accessible only from classes in the same package; not even a subclass in a different package can see it.
- `private`: Accessible only from within the declaring class itself — a subclass can't access it directly **even though it inherits it**.

```
class Base {
    public String publicField = "public";
    protected String protectedField = "protected";
    String packagePrivateField = "package-private"; // no modifier
    private String privateField = "private";

    private String getPrivateField() {
        return privateField;
    }

    protected String privateFieldViaGetter() {
        // A subclass cannot see privateField directly, but it CAN call this
        // protected getter, which is defined inside Base and has access.
        return getPrivateField();
    }
}

class Derived extends Base {
    void printAccessible() {
        System.out.println(publicField);              // OK -- public
        System.out.println(protectedField);            // OK -- protected, visible to
subclasses
        System.out.println(packagePrivateField);       // OK -- same package (default
package here)
        // System.out.println(privateField);           // compile error: privateField has
private access in Base
        System.out.println(privateFieldViaGetter());   // OK -- goes through Base's own
protected getter
    }
}

class AccessModifiersExample {
    public static void main(String[] args) {
        new Derived().printAccessible();
        // Note: if Derived lived in a DIFFERENT package than Base, packagePrivateField
        // would no longer compile -- package-private members are invisible across
        // package boundaries even to subclasses. protectedField would still work,
        // because `protected` is specifically designed to stay visible to subclasses
        // regardless of package.
    }
}
```

Expected Output

```
public
protected
package-private
private
```

A `private` field is technically **inherited** by a subclass (it's part of the subclass instance's memory layout), but the subclass can't reach it by name directly — only through a `public` / `protected` getter the superclass provides. "Inheriting" something and "being able to access" it are not the same thing.

Field Hiding vs. Method Overriding

This is a distinction most sources skip, and it trips a lot of people up: when a subclass overrides a **method**, the runtime resolves the call based on the object's actual class (dynamic dispatch, as covered in "Method Overriding") — but when a subclass declares a **new field with the same name** as one in the superclass (this is called **field hiding**), which field you get is decided not at runtime, but by the **compile- time static type of the variable**:

```
class Animal {
    String label = "Animal";
}

class Dog extends Animal {
    String label = "Dog"; // hides Animal's label, does NOT override it
}

class FieldHidingExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = dog; // same object, viewed through a different static type

        System.out.println(animal.label); // Animal -- resolved by the STATIC type (Animal)
        System.out.println(dog.label);    // Dog    -- resolved by the STATIC type (Dog)
    }
}
```

Expected Output

```
Animal
Dog
```

`animal.label` and `dog.label` print **different values** even though they refer to the same object — because field access isn't polymorphic, only method calls are. That inconsistency is exactly why you should **never redeclare a field as if you were overriding it**; we'll come back to this in "Common Mistakes."

> Unlike method overriding, field hiding is not real polymorphism — it's simply the subclass field **hiding** the superclass field of the same name. Both fields continue to exist separately in memory; you can still reach the superclass's copy with `super.field` (recall "The super Keyword").

final Classes and final Methods

The `final` keyword blocks inheritance at two different levels: a `final` **class can never be extended**, and a `final` **method can never be overridden** (though a subclass can still inherit and use it normally):

```
final class ImmutablePoint {
    final int x;
    final int y;

    ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

// class SubPoint extends ImmutablePoint { } // compile error: cannot inherit from final
ImmutablePoint

class Base {
    final void identify() {
        System.out.println("I am a Base, and this method cannot be overridden");
    }

    void greet() {
        System.out.println("Hello from Base");
    }
}

class Derived extends Base {
    // @Override
    // final void identify() { } // compile error: identify() cannot override final method
in Base

    @Override
    void greet() {
        System.out.println("Hello from Derived"); // greet() is NOT final, so overriding it
is fine
    }
}

class FinalClassAndMethodExample {
    public static void main(String[] args) {
        Derived derived = new Derived();
        derived.identify(); // inherited as-is, cannot be overridden
        derived.greet();    // overridden version runs
    }
}
```

Expected Output

```
I am a Base, and this method cannot be overridden
Hello from Derived
```

The most familiar example of this restriction is `String` itself — because `String` is `final`, nobody can write `class MyString extends String` and change its behavior. This is a deliberate design choice: it guarantees that none of the safety and performance assumptions built on `String`'s immutability (such as the string pool being safely shareable) can ever be broken by a subclass.

> Marking a method `final` is how you guarantee that its **behavior stays the same everywhere in the class hierarchy** — as we saw with the Template Method pattern in the Abstract Class lesson, the method that defines a fixed algorithm skeleton is typically marked `final`.

The Object Class

In Java, even if you never write `extends`, **every class implicitly extends `java.lang.Object`** — which is why `Object` sits at the root of the class hierarchy, the common ancestor of every class. `Object` provides a handful of fundamental methods that every object inherits; the three most commonly overridden are `toString()`, `equals(Object)`, and `hashCode()`:

```
import java.util.Objects;

class Point {
    int x;
    int y;

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

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

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

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

class ObjectClassExample {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        System.out.println(p1);                              // Point(1, 2) -- uses
overridden toString()
        System.out.println(p1.equals(p2));                    // true -- same x,y,
overridden equals()
        System.out.println(p1.hashCode() == p2.hashCode());   // true -- required when
equals() is true
    }
}
```

The default output of an un-overridden `toString()` (something like `ClassName@hashcode`) is almost never useful — which is why it's overridden in any class you want to print something meaningful. `equals()` and `hashCode()` must be overridden **together**: if two objects are equal per `equals()`, their `hashCode()`s **must** also be equal — otherwise hash-based collections like `HashMap`/`HashSet` will behave as if objects have gone missing.

> Overriding `equals()` and leaving `hashCode()` un-overridden is the single most common `Object`-method mistake — we'll revisit it in "Common Mistakes."

Upcasting

Assigning a subclass object to a variable of the superclass type is called **upcasting** — in Java this is always safe and happens implicitly (no cast needed), because every `Dog` already is an `Animal` (recall the is-a relationship from "What Is Inheritance?"):

```
class Animal {
    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }

    void bark() {
        System.out.println("Extra loud woof!");
    }
}

class UpcastingExample {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcasting -- implicit, always safe

        animal.makeSound(); // Woof! -- runtime type (Dog) decides, not the static type
(Animal)
        // animal.bark();   // compile error: bark() is not visible through an Animal
reference

        Animal[] animals = { new Animal(), new Dog() };
        for (Animal a : animals) {
            a.makeSound(); // each element uses ITS OWN real implementation
        }
    }
}
```

Expected Output

```
Woof!
Some generic animal sound
Woof!
```

In `Animal animal = new Dog();`, the variable's **static type** is `Animal`, but the object it points to still has a **runtime type** of `Dog`. Through `animal` you can only call the methods `Animal` defines (`bark()` isn't visible) — but when you call an overridden method like `makeSound()`, which implementation runs is always decided by the **actual** type (the dynamic dispatch we saw in "Method Overriding"). This is the foundation of writing polymorphic code: a single `List<Animal>` can work with one type, while each element still exhibits its own real behavior.

Downcasting and instanceof

Converting a superclass-typed variable back to a subclass type is called **downcasting** — unlike upcasting, this is **not always safe** and requires an explicit cast. If the object the variable points to isn't actually of that subclass, a `ClassCastException` is thrown at runtime. The `instanceof` operator is the safe way to check an object's actual type before attempting a cast:

```
class Animal {
}

class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meow!");
    }
}

class DowncastingExample {
    public static void main(String[] args) {
        Animal animal = new Dog(); // upcast first, as in the previous section

        if (animal instanceof Dog dog) {
            // pattern matching -- checks the type AND casts into `dog` in one step
            dog.bark(); // Woof!
        }

        Animal other = new Cat();
        if (other instanceof Dog otherDog) {
            otherDog.bark(); // never runs -- `other` is really a Cat
        } else {
            System.out.println("other is not a Dog"); // this branch runs instead
        }

        // Dog forced = (Dog) other; // compiles, but throws ClassCastException at runtime
    }
}
```

Expected Output

```
Woof!
other is not a Dog
```

Modern Java (16 and later) lets you combine the check and the cast into one line via **pattern matching** for `instanceof` — `if (animal instanceof Dog dog)` both checks the type and automatically declares `dog` at that type, no separate cast line required.

> Writing a cast like `(Dog) animal` directly, without an `instanceof` check first, will crash your program with a `ClassCastException` if the object isn't actually a `Dog`. Only downcast when it's genuinely necessary (say, an API only hands you the superclass type), and always guard it with an `instanceof` check.

Why Java Doesn't Support Multiple Inheritance

Why does Java support only single inheritance for classes? The answer lies in the **Diamond Problem** we touched on in "History." If both `B` and `C` inherited from `A`, and `D` inherited from both `B` and `C` (a scenario Java simply won't compile), and both `B` and `C` had overridden some method from `A` in their own way, which version should `D` inherit? That ambiguity was a real problem in C++. Java resolves it for classes by banning the scenario outright — but a similar clash is still possible with interfaces' `default` methods (recall the "Default Methods" section in the Interface lesson), and Java resolves that with a different rule:

```
interface Flyer {
    default String move() {
        return "flying";
    }
}

interface Swimmer {
    default String move() {
        return "swimming";
    }
}

// A class implementing both would inherit two conflicting move() defaults.
// Java refuses to guess -- it forces the class to resolve the conflict itself.
class Multi implements Flyer, Swimmer {
    @Override
    public String move() {
        // Flyer.super.move() / Swimmer.super.move() let you reach either default explicitly
        return Flyer.super.move() + " and " + Swimmer.super.move();
    }
}

class DiamondProblemExample {
    public static void main(String[] args) {
        Multi multi = new Multi();
        System.out.println(multi.move()); // flying and swimming
    }
}
```

Expected Output

```
flying and swimming
```

`Multi` implements both `Flyer` and `Swimmer`, so it would inherit `move()` from both — and Java **doesn't resolve this automatically**, it's a compile error. The fix is for `Multi` to override `move()` itself, reaching either default explicitly with syntax like `Flyer.super.move()` if it needs to. So the "diamond" situation isn't entirely impossible for interfaces, but it's **never resolved silently or ambiguously** — the compiler forces you to make an explicit decision. For classes, though, Java cuts the ambiguity off at the root: a class can never `extends` two classes at once.

Inheritance vs. Composition

Inheritance is a powerful tool, but it isn't the answer to every "I need shared code" problem. **Composition** is when a class holds another class **as a field** instead of extending it — a "has-a" relationship rather than "is-a." Joshua Bloch's famous advice in *Effective Java* says exactly this: **"favor composition over inheritance"**:

```
class Engine {
    void start() {
        System.out.println("Engine starting...");
    }

    void stop() {
        System.out.println("Engine stopping...");
    }
}

// Inheritance approach: Car "is-a" Engine -- semantically wrong, and Car
// exposes every one of Engine's public methods whether it makes sense or not.
class CarWithInheritance extends Engine {
}

// Composition approach: Car "has-a" Engine -- Car controls exactly which
// Engine behavior it exposes, and can swap the engine implementation freely.
class CarWithComposition {
    private final Engine engine;

    CarWithComposition(Engine engine) {
        this.engine = engine;
    }

    void drive() {
        engine.start(); // delegation -- Car forwards to its Engine
        System.out.println("Car driving...");
    }
}

class CompositionVsInheritanceExample {
    public static void main(String[] args) {
        CarWithInheritance carA = new CarWithInheritance();
        carA.start(); // works, but semantically odd -- a Car IS an Engine?

        CarWithComposition carB = new CarWithComposition(new Engine());
        carB.drive(); // Engine starting... / Car driving...
    }
}
```

`CarWithInheritance` extends `Engine` to inherit its `start()` — but that's odd: a `Car` genuinely **isn't** an `Engine`, it **has** one. `CarWithComposition`, on the other hand, holds an `Engine` as a field and **calls** its `start()` from inside its own `start()` (delegation). This second approach lets you swap the engine at runtime (you could pass in a different `Engine` implementation, like `ElectricEngine`), keeps `Car` from accidentally exposing **every** public method of `Engine` (stronger encapsulation), and keeps `Car`'s behavior predictable even if `Engine` changes.

> Ask yourself: "Should the subclass meaningfully inherit **every** public method of the superclass?" If the answer is no (it would be strange for a `Car` to directly expose every method an `Engine` has), reach for composition instead. In "Real-World Examples" we'll see that even the JDK sometimes misuses inheritance in exactly this way.

Real-World Examples

Inheritance shows up throughout the JDK's own design, both used well and used poorly. The exception hierarchy in `java.io` is a classic, correct use: `IOException` extends `Exception`; more specific exceptions like `FileNotFoundException` and `EOFException` extend `IOException`. That means a single `catch (IOException e)` block can catch any of those specific subtypes. We can build the same pattern in our own code:

```
class AppException extends Exception {
    AppException(String message) {
        super(message);
    }
}

class ValidationException extends AppException {
    ValidationException(String message) {
        super(message);
    }
}

class NotFoundException extends AppException {
    NotFoundException(String message) {
        super(message);
    }
}

class RealWorldHierarchyExample {
    static void handle(AppException e) {
        // One catch site handles every subtype, exactly like catching java.io.IOException
        // handles FileNotFoundException, EOFException, and every other subtype.
        System.out.println("Handled: " + e.getMessage());
    }

    public static void main(String[] args) {
        handle(new ValidationException("email is invalid"));
        handle(new NotFoundException("user not found"));
    }
}
```

`ValidationException` and `NotFoundException` both extend `AppException` — just like the JDK's `IOException` subtypes extend `IOException`. The `handle()` method catches both with a single `catch (AppException e)`, while `getMessage()` still returns each exception's own message.

There's also a famous example of inheritance **misused** in the JDK: `java.util.Stack` extends `java.util.Vector` — which means `Stack` also (accidentally) inherits every one of `Vector`'s methods, including things like `add(int, E)` that insert at an arbitrary position, even though a stack should only ever be worked with from the top, via `push` / `pop`. This is exactly the kind of misuse we discussed in "Inheritance vs. Composition" — modern JDK collections (like `ArrayDeque`) are preferred over `Stack` for this reason.

Best Practices

- Always use the `@Override` annotation when overriding a method — it catches signature mistakes at compile time (see "Method Overriding").
- **Never** declare a new field with the same name as one in the superclass — field hiding is a static-type-dependent behavior most developers don't expect (see "Field Hiding vs. Method Overriding").
- If you override `equals()`, always override `hashCode()` alongside it — otherwise objects vanish in hash-based collections (see "The Object Class").
- Always check with `instanceof` before downcasting, and prefer the pattern-matching syntax where possible (see "Downcasting and instanceof").
- Whenever a subclass wouldn't meaningfully expose **all** of a superclass's public behavior, prefer composition over inheritance — "favor composition over inheritance" (see "Inheritance vs. Composition").
- If you never want a class to be extended (say, it carries an immutable value), mark it `final` — `String` itself is the most familiar example of this principle (see "final Classes and final Methods").

Common Mistakes

1. Redeclaring a superclass field under the same name as if it were being overridden. This is field hiding, not method overriding, and it resolves based on the static type — usually unwanted, confusing behavior (see "Field Hiding vs. Method Overriding").

2. Overriding `equals()` and forgetting `hashCode()`. Two equal objects producing different hash codes makes objects appear to "vanish" in collections like `HashMap` / `HashSet` (see "The Object Class").

3. Writing a downcast without an `instanceof` check first. If the object isn't actually of that type, this throws a `ClassCastException` at runtime (see the warning in "Downcasting and instanceof").

4. Reaching for inheritance purely to share code, when the "is-a" relationship isn't actually meaningful. The JDK's own `Stack extends Vector` is the classic example — composition is a much better fit in cases like this (see "Real-World Examples").

5. Forgetting `super(...)` in a subclass constructor and assuming the superclass has a no-argument constructor. If the superclass only has a constructor that takes arguments, skipping an explicit, correctly-parameterized `super(...)` call fails to compile (see the tip in "Constructors and super()").

6. Building very deep inheritance chains (like `A → B → C → D → E`). Every extra level makes the next one harder to understand, and predicting the effect of a change at a higher level without reading the whole chain becomes impossible — a good rule of thumb is to avoid going more than three levels deep.

Summary, Cheat Sheet, and Glossary

Inheritance has been Java's fundamental mechanism for sharing code between classes since its first release in 1996. Key takeaways:

- The relationship `extends` establishes is an **"is-a"** relationship — a subclass inherits the superclass's fields and methods
- The first line of a subclass constructor always (explicitly or implicitly) calls the superclass's constructor — the superclass's state is established **before** the subclass's
- Method overriding is polymorphic (resolved by the actual runtime type); field hiding is **not** (resolved by the variable's static type) — these two are frequently confused
- A `final` class can't be extended, a `final` method can't be overridden
- Every class implicitly extends `Object` — `toString()`, `equals()`, and `hashCode()` are the most commonly overridden `Object` methods
- Upcasting is implicit and always safe; downcasting requires an explicit cast and should be guarded with `instanceof`
- Java doesn't support multiple inheritance for classes because of the Diamond Problem; if a similar clash happens with interfaces' `default` methods, the compiler forces the developer to resolve it explicitly
- When the "is-a" relationship isn't genuinely meaningful, prefer composition ("has-a") over inheritance

Quick reference:

```
// Basic definition and terminology
class Animal {                    // superclass / parent / base class
    String name;
}

class Dog extends Animal {        // subclass / child / derived class
    Dog(String name) {
        super(name);              // constructor chaining -- always the first line
    }
}

// Method overriding vs field hiding
class Base {
    String label = "Base";
    String describe() { return "Base"; }
}

class Sub extends Base {
    String label = "Sub";               // field hiding -- resolved by STATIC type
    @Override
    String describe() { return "Sub"; } // overriding -- resolved by RUNTIME type
}

// super keyword -- three forms
class Child extends Base {
    Child() {
        super();                  // 1) call superclass constructor
    }
    @Override
    String describe() {
        return super.describe();  // 2) call superclass method
    }
    void show() {
        System.out.println(super.label); // 3) access superclass field
    }
}

// Upcasting / downcasting
Animal a = new Dog("Rex");        // upcasting -- implicit, always safe
if (a instanceof Dog d) {         // downcasting -- explicit, needs instanceof
    // use d as a Dog
}

// final
final class CannotBeExtended { }
class HasFinalMethod {
    final void fixedBehavior() { }
}
```

Glossary

Inheritance — The mechanism that lets a class (the subclass) acquire the fields and methods of another class (the superclass).

Superclass / parent class / base class — The class being `extends`ed, which passes down its fields and behavior.

Subclass / child class / derived class — The class that `extends`, inheriting the superclass's members.

"is-a" relationship — The kind of relationship inheritance represents ("Dog is an Animal"); distinct from the "has-a" relationship (composition).

Method overriding — A subclass redefining an inherited method with the same signature; resolved at runtime based on the actual type (dynamic dispatch).

Field hiding — A subclass declaring a new field with the same name as one in the superclass; resolved at compile time based on the static type, not polymorphic.

`super` — The keyword used to call a superclass's constructor (`super(...)`), call its overridden method (`super.method()`), or access one of its fields (`super.field`).

Upcasting — Assigning a subclass object to a variable of the superclass type; implicit and always safe.

Downcasting — Converting a superclass-typed variable back to a subclass type; requires an explicit cast, should be guarded with `instanceof`, and risks a `ClassCastException` otherwise.

Diamond Problem — In multiple inheritance, the ambiguity of which conflicting member inherited from the same superclass/interface should be used; Java avoids this for classes by banning multiple inheritance outright.

Composition — A class holding another class as a field instead of extending it (a "has-a" relationship); the subject of Joshua Bloch's "favor composition over inheritance" advice.

Appendix: Mini Project — Employee Hierarchy (Manager and Developer)

Let's put what we've learned into a real-world scenario: the `Employee`s at a company share common fields (`name`, `baseSalary`) and behavior (`describe()`), but salary calculation logic varies by role — a `Manager` gets a team bonus, a `Developer` gets extra pay based on overtime hours:

```
class Employee {
    String name;
    double baseSalary;

    Employee(String name, double baseSalary) {
        this.name = name;
        this.baseSalary = baseSalary;
    }

    double calculateSalary() {
        return baseSalary;
    }
```

```
    String describe() {
        return name + ": base salary " + baseSalary;
    }
}

class Manager extends Employee {
    double teamBonus;

    Manager(String name, double baseSalary, double teamBonus) {
        super(name, baseSalary);
        this.teamBonus = teamBonus;
    }

    @Override
    double calculateSalary() {
        return baseSalary + teamBonus;
    }

    @Override
    String describe() {
        return super.describe() + ", team bonus " + teamBonus + ", total " +
calculateSalary();
    }
}

class Developer extends Employee {
    int overtimeHours;
    static final double OVERTIME_RATE = 50.0;

    Developer(String name, double baseSalary, int overtimeHours) {
        super(name, baseSalary);
        this.overtimeHours = overtimeHours;
    }

    @Override
    double calculateSalary() {
        return baseSalary + (overtimeHours * OVERTIME_RATE);
    }

    @Override
    String describe() {
        return super.describe() + ", overtime " + overtimeHours + "h, total " +
calculateSalary();
    }
}
```

```
class EmployeeHierarchyDemo {
    public static void main(String[] args) {
        Employee[] employees = {
            new Employee("Ada", 4000),
            new Manager("Grace", 6000, 1500),
            new Developer("Linus", 5000, 10)
        };

        for (Employee e : employees) {
            System.out.println(e.describe());
        }
        // Ada: base salary 4000.0
        // Grace: base salary 6000.0, team bonus 1500.0, total 7500.0
        // Linus: base salary 5000.0, overtime 10h, total 5500.0
    }
}
```

`Manager` and `Developer` each override `Employee`'s `calculateSalary()` with their own formula, and in `describe()` they call `super.describe()` (recall "The super Keyword") to build on top of `Employee`'s shared text instead of rewriting it. The array in `EmployeeHierarchyDemo` holds every element as an `Employee` (see "Upcasting"), yet each one runs its own `calculateSalary()` implementation.

> We deliberately didn't make `Employee` `abstract` here — a concrete superclass, not just an abstract base like we saw in the Abstract Class lesson, is a perfectly valid design for inheritance. Making a superclass `abstract` is only necessary when you specifically want to prevent it from being instantiated directly.

Appendix: Mini Project — Vehicle Hierarchy (Multi-Level Inheritance)

Our last mini project builds a **multi-level** inheritance chain instead of a single level — `Vehicle → MotorVehicle → Car` / `Motorcycle` — combining several ideas from this lesson (the constructor chain, overriding `Object` methods, upcasting/downcasting) into one example:

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }

    @Override
    public String toString() {
        return brand;
    }
}

class MotorVehicle extends Vehicle {
```

```
    int horsePower;

    MotorVehicle(String brand, int horsePower) {
        super(brand);
        this.horsePower = horsePower;
    }

    void revEngine() {
        System.out.println(brand + " engine revving at " + horsePower + " hp");
    }

    @Override
    public String toString() {
        return super.toString() + " (" + horsePower + " hp)";
    }
}

class Car extends MotorVehicle {
    int doors;

    Car(String brand, int horsePower, int doors) {
        super(brand, horsePower);
        this.doors = doors;
    }

    @Override
    public String toString() {
        return super.toString() + ", " + doors + " doors";
    }
}

class Motorcycle extends MotorVehicle {
    boolean hasSidecar;

    Motorcycle(String brand, int horsePower, boolean hasSidecar) {
        super(brand, horsePower);
        this.hasSidecar = hasSidecar;
    }

    @Override
    public String toString() {
        return super.toString() + (hasSidecar ? " with sidecar" : " no sidecar");
    }
}
```

```
class VehicleHierarchyDemo {
    public static void main(String[] args) {
        Vehicle[] vehicles = {
            new Car("Toyota", 150, 4),
            new Motorcycle("Harley-Davidson", 90, false)
        };

        for (Vehicle v : vehicles) {
            System.out.println(v); // uses each subclass's overridden toString()

            if (v instanceof MotorVehicle motorVehicle) {
                motorVehicle.revEngine(); // safe downcast, all vehicles here are motorized
            }

            if (v instanceof Car car) {
                System.out.println("It's a car with " + car.doors + " doors");
            } else if (v instanceof Motorcycle motorcycle) {
                System.out.println("It's a motorcycle, sidecar: " + motorcycle.hasSidecar);
            }
        }
    }
}
```

`Car`'s constructor calls `MotorVehicle` via `super(...)`, which in turn calls `Vehicle` via its own `super(...)` — a three-level constructor chain that always runs top-down, starting from `Vehicle` (recall "Constructors and super()"). `VehicleHierarchyDemo` demonstrates both upcasting (the `Vehicle` array) and safe downcasting with `instanceof` (see "Downcasting and instanceof") together.

> A three-level chain (`Vehicle → MotorVehicle → Car`) stays readable in this example, but as we warned in "Common Mistakes," every additional level makes things harder to follow — before building a hierarchy that goes past four or five levels in a real codebase, always ask whether composition would be a better fit.