Java Inheritance

Table of contents

Subclasses and superclasses 1
`extends` 3
The `super` keyword 3
Method overriding 4

Introduced in 2026-03-05-object-oriented-programming-i[1] (Lecture, Week 2) — "things you have because your parents have them".

Subclasses and superclasses

Inheritance allows creating a new class from an existing class:

- The new class is the **subclass** (child/derived class).
- The existing class it's derived from is the **superclass** (parent/base class).

The running example is one `Employee` superclass with two subclasses:

This form is not a course-wide standard. The course draws inheritance in **three** ways: this UML three-compartment form with an unfilled triangle, a labelled box with a plain arrowhead, and a diagonal form with every edge labelled "Is a". **None of them is declared canonical.** See csse2002-diagram-notation[2] for the compartment, visibility and arrowhead conventions this figure uses.

Two things to read off the diagram before looking at any code:

- `name` and `address` are declared **`private` in `Employee`** and are *not* repeated in either subclass. Every `ContractEmployee` carries them, but the subclass's own code cannot name them — `private` members are not accessible in a subclass — which is what forces the `super(...)` call below.

[1] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf
[2] courses/csse2002/csse2002-diagram-notation.pdf

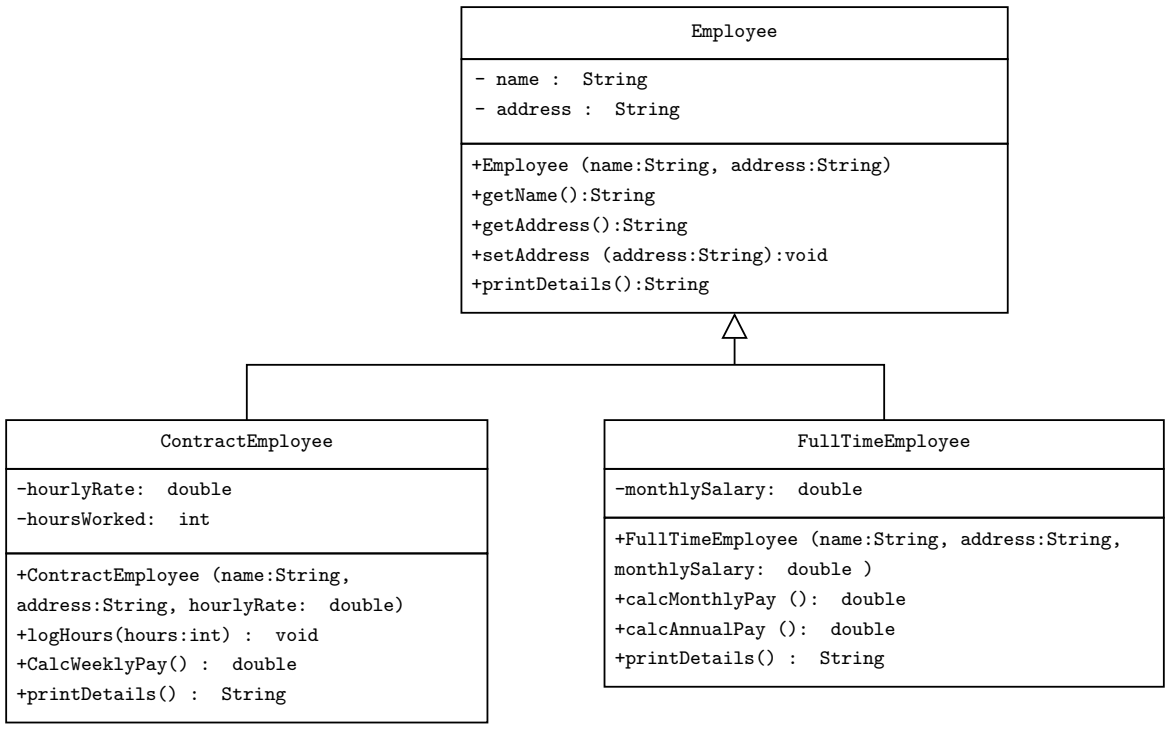

Figure 1: The `Employee` inheritance example as a UML class diagram. Each box has three compartments (name / attributes / operations), with - for private and + for public, and constructors listed in the operations compartment with no return type. Generalisation is a solid line with a single unfilled triangle arrowhead entering the superclass.

- `printDetails()` appears in **all three** boxes. That repetition is the setup for method overriding.

(The diagram writes `CalcWeeklyPay()` with a capital C; the code calls the same method `calculateWeeklyPay()`. The code spelling is the one to trust.)

`extends`

`extends` is the keyword used to implement inheritance in Java:

```
public class Employee {
    private String name;
    private String address;

    public Employee(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public String getName() { return name; }
    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
    public String printDetails() { return name + " " + address; }
}

public class ContractEmployee extends Employee {
    private double hourlyRate;
    private int hoursWorked;

    public ContractEmployee(String name, String address, double hourlyRate) {
        super(name, address);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = 0;
    }
    public void logHours(int hours) { this.hoursWorked += hours; }
    public double calculateWeeklyPay() { return hourlyRate * hoursWorked; }
}
```

From the diagram, the shape of `FullTimeEmployee` is `private double monthlySalary`, a constructor `FullTimeEmployee(String name, String address, double monthlySalary)`, and `calcMonthlyPay()` / `calcAnnualPay()` both returning `double`.

The `super` keyword

`super` is used in a subclass to access superclass **members — attributes, constructors and methods**.

As a constructor call, `super(name, address)` runs `Employee`'s constructor first, so the inherited `name` and `address` are initialised before the subclass's own constructor body runs. This is not merely tidy: `name` and `address` are `private` in `Employee`, so `ContractEmployee` has no other way to set them.

Method overriding

The subclass inherits the attributes and methods of its superclass. If the **same method** is defined in both the superclass and the subclass, the subclass's version **overrides** the superclass's version — and both **MUST** have the same method signature (see java-encapsulation[3]; a different signature gives you an overload, not an override).

```
public class Employee {
    ...
    public String printDetails() {
        return name + " " + address;
    }
}

public class ContractEmployee extends Employee {
    ...
    @Override
    public String printDetails() {
        return "Role: Contract Employee, Hourly Rate: $" + hourlyRate
             + ", Hours Worked: " + hoursWorked;
    }
}

public class FullTimeEmployee extends Employee {
    @Override
    public String printDetails() {
        return "Role: Full-Time Employee, Monthly Salary: $" + monthlySalary;
    }
}
```

`super` in an override

The superclass's method is still reachable after overriding it. `super.methodName(...)` calls the superclass's version from inside the override, so the subclass can **extend** the inherited behaviour rather than replace it:

[3] courses/csse2002/java-encapsulation.pdf

```java
public class FullTimeEmployee extends Employee {
    ...
    @Override
    public String printDetails() {
        return super.printDetails()
             + ", Role: Full-Time Employee, Monthly Salary: $" + monthlySalary;
    }
}
```

With `Employee.printDetails()` returning `name + " " + address`, this version prints the name and address *and* the role and salary, where the previous version threw the inherited half away.

Choosing whether `Employee`'s `printDetails()` or a subclass's override actually runs for a given object is **dynamic dispatch**, taken up with polymorphism — see java-polymorphism[4] and week4-tutorial-method-dispatch-and-casting[5].

[4] courses/csse2002/java-polymorphism.pdf
[5] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf