Java Polymorphism

Table of contents

What is polymorphism? 1
Compile-time polymorphism (method overloading) 1
Runtime polymorphism (dynamic method dispatch) 2
Subtype polymorphism 3
How it connects to casting 5

Introduced in 2026-03-12-object-oriented-programming-ii[1] (Lecture, Week 3). Practised in week4-tutorial-method-dispatch-and-casting[2].

What is polymorphism?

From the Greek *"poly"* (many) and *"morphe"* (form). In programming, it allows the same method call to produce different behaviour depending on the object that executes it.

There are three named kinds. The first two differ in **when** the method body is chosen; the third is about **what types a call site will accept**.

Compile-time polymorphism (method overloading)

> **Method overloading** occurs when multiple methods have the same name but different parameter lists (different type, number, or both). The method to be executed is determined at compile time based on the method signature.

Which one runs is decided from the signature (see java-encapsulation[3]) before the program ever runs:

[1] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[2] courses/csse2002/week4-tutorial-method-dispatch-and-casting.pdf
[3] courses/csse2002/java-encapsulation.pdf

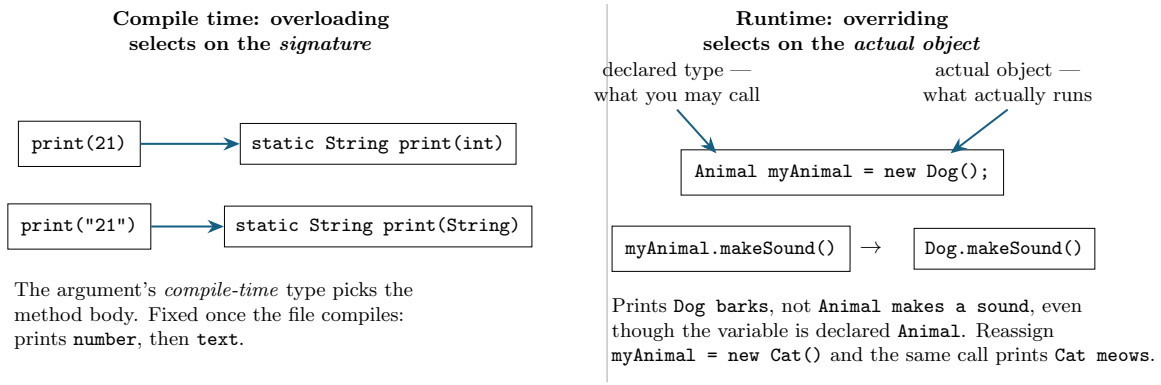

Figure 1: The two selection mechanisms. Left: overloading is resolved by the compiler from the argument's compile-time type, so the choice is fixed once the file compiles. Right: overriding is resolved by the JVM from the object actually on the other end of the reference; the arrow from `myAnimal.makeSound()` to `Dog.makeSound()` means the call dispatches to that method.

```java
public class CompPolymorphism {
    static String print(int value) {
        return "number";
    }
    static String print(String value) {
        return "text";
    }

    public static void main(String[] args) {
        System.out.println(print(21));    // number
        System.out.println(print("21"));  // text
    }
}
```

Runtime polymorphism (dynamic method dispatch)

- Runtime polymorphism is also known as **dynamic method dispatch**.
- It occurs when the method that gets executed is determined **at runtime based on the actual object type, not the variable type**.
- It is implemented through **method overriding** — when a subclass provides a specific implementation for a method already defined in its superclass, the subclass's method is the *overridden* method (see java-inheritance[4]).

[4] courses/csse2002/java-inheritance.pdf

The rule:

> The subclass's overriding method executes, regardless of the compile-time type of the subclass instance. — *Effective Java*

The example below **reassigns the same variable**: nothing about the variable changed, only the object it points at.

```
public class Animal {
    public void makeSound() { System.out.println("Animal makes a sound"); }
}
public class Cat extends Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
}
public class Dog extends Animal {
    @Override public void makeSound() { System.out.println("Dog barks"); }
}
```

```
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound();   // Dog barks

        myAnimal = new Cat();
        myAnimal.makeSound();   // Cat meows
    }
}
```

Even though the object type is `Animal`, the `Dog` version of `makeSound()` runs because the actual object is `Dog`.

Subtype polymorphism

- Subtype polymorphism occurs when objects of different subclasses can be treated as objects of a common superclass **or interface**.
- This is possible because subclasses are *subtypes* of their superclass.
- It is achieved through inheritance (`extends`) or interface implementation (`implements`).
- It allows code to work with the abstraction (supertype) instead of specific implementations.

The rule the whole idea rests on:

> A subclass object can be used anywhere a superclass is expected.

Where runtime polymorphism is about *which body runs*, subtype polymorphism is about *what a call site will accept*: `performAnimalSound` is written once against `Animal` and never mentions `Dog` or `Cat`.

```
public class TestSubtypePolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();

        performAnimalSound(myAnimal);   // a Dog object
    }

    public static void performAnimalSound(Animal animal) {
        animal.makeSound();             // Outputs: Dog barks
    }
}
```

The same method works unchanged when the supertype is an **interface** rather than a superclass. Any class that implements the interface can be passed:

```
public interface Animal {
    void makeSound();
}

public class Cat implements Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
}
public class Dog implements Animal {
    @Override public void makeSound() { System.out.println("Dog barks"); }
}
```

```
public class TestSubtypePolymorphism {
    public static void main(String[] args) {
        performAnimalSound(new Dog());
        performAnimalSound(new Cat());
    }

    public static void performAnimalSound(Animal animal) {
        animal.makeSound();
    }
}
```

Choosing between the two supertypes is the interface-vs-abstract-class question — see java-abstraction[5].

[5] courses/csse2002/java-abstraction.pdf

How it connects to casting

Assigning `new Dog()` to an `Animal` variable is an implicit **upcast** (java-casting[6]). That is why the declared type restricts what you may call (`myAnimal.climb()` will not compile even when the object is a `Cat`) while the actual object still decides what runs — the left and right halves of Figure 1 are the same reference seen from the compiler's side and the JVM's side.

[6] courses/csse2002/java-casting.pdf