Java Casting

Table of contents

Primitive casting 1
Reference casting 2
Every class implicitly extends `Object` 4

Introduced in 2026-03-12-object-oriented-programming-ii[1] (Lecture, Week 3). See also java-primitive-and-reference-types[2] and week4-tutorial-method-dispatch-and-casting[3].

Casting is the process of converting a variable from one data type to another. There are two kinds: **primitive casting**, between the built-in numeric types, and **reference casting**, up and down an inheritance hierarchy.

Primitive casting

Set as self-study (homework). The reading is w3schools' type casting page.

- **Widening** — converting a smaller type to a larger type size: `byte -> short -> char -> int -> long -> float -> double`.
- **Narrowing** — converting a larger type to a smaller size type: `double -> float -> long -> int -> char -> short -> byte`.

An example of a narrowing cast: `long floatToLong = (long) inputFloat;`.

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

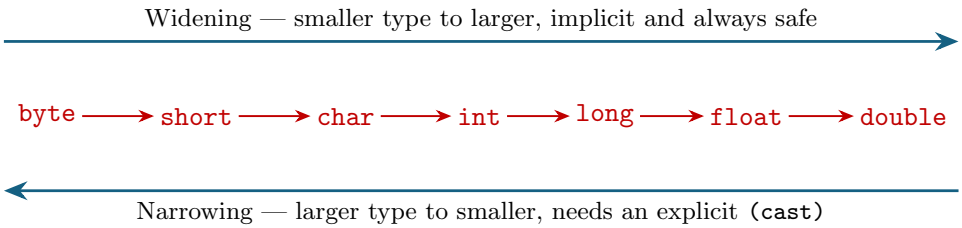

Figure 1: The two primitive conversion chains on a single axis: the narrowing chain is exactly the widening chain reversed. Widening is implicit; narrowing needs an explicit cast.

Reference casting

- **Upcasting** (implicit) — casting a subclass reference to a superclass reference.
- **Downcasting** (explicit) — casting a superclass reference back to a subclass reference.

The two directions are defined against a class hierarchy: up the tree is upcasting, down the tree is downcasting.

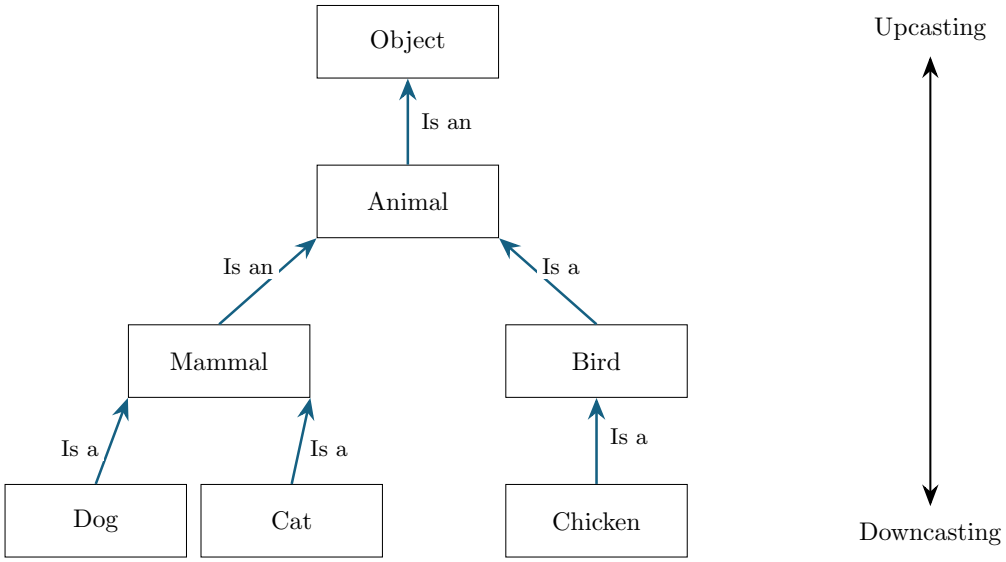

Figure 2: The class hierarchy that upcasting and downcasting move along, with 'Is a' / 'Is an' edge labels and the up/down casting axis at the right. Object sits at the root because every class implicitly extends it.

Upcasting

Upcasting is done implicitly and is always safe: every `Cat` really is an `Animal`, so no check and no cast syntax is needed. In the assignment below the `Cat` is treated as an `Animal`:

```
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"); }
}

Animal myAnimal = new Cat(); // upcasting: Cat is treated as an Animal
myAnimal.makeSound();        // Outputs: Cat meows
```

After the upcast, the *declared* type narrows what you may call, but the *object* still decides which body runs. That is java-polymorphism[4]'s runtime dispatch, and it is why the line above prints `Cat meows` rather than `Animal makes a sound`.

Downcasting

Before downcasting, you must ensure the object is an instance of the subclass using the `instanceof` operator. The cast is written explicitly, and it is the only way to reach members that exist on the subclass but not the superclass:

```
public class Cat extends Animal {
    @Override public void makeSound() { System.out.println("Cat meows"); }
    public void climb() { System.out.println("I can climb trees"); }
}

public class DownCasting {
    public static void main(String[] args) {
        Animal myAnimal = new Cat();     // upcasting

        if (myAnimal instanceof Cat) {
            Cat myCat = (Cat) myAnimal;  // Downcasting
            myCat.climb();               // access methods specific to cat
        }
    }
}
```

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

The `instanceof` guard is mandatory because an unguarded bad downcast is not a compile error: it compiles and throws `ClassCastException` at run time (see java-exceptions[5]).

Additional reading: upcasting and downcasting in Java.

Every class implicitly extends `Object`

Worked question: what is the parent of class `Student`, a class with no `extends` clause at all?

```
public class Student {
    public void study() {
        System.out.println("CSSE2002");
    }
}
```

The answer is `Object`, the root of the Java class hierarchy and the top box of Figure 2. So every reference type can be upcast to `Object`, and a container declared over `Object` will hold anything. Getting a specific type back out of one means downcasting, which is the problem java-generics[6] exists to remove.

[5] courses/csse2002/java-exceptions.pdf
[6] courses/csse2002/java-generics.pdf