Java Mutability

Table of contents

Introduced in 2026-03-12-object-oriented-programming-ii[1] (Lecture, Week 3).

Mutable vs immutable classes

Objects in Java can either change their state after creation or remain fixed forever:

- **Mutable class** — object state can change after creation.
- **Immutable class** — object state cannot change after creation.

Mutable

```
public class Person {
    private String name;
    private int age;

    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
}

Person p = new Person();
p.setName("Alice");
p.setAge(20);
p.setAge(30); // state changed
```

[1] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf

Immutable

Made `final` (fields set once in the constructor, no setters):

```
public final class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}
```

A subclass can reintroduce mutability into an otherwise-immutable design (e.g. adding a setter, or exposing mutable internal state) — worth reading up on (Effective Java, Item 16) before relying on "final" alone as a guarantee.

Arrays and lists break naive immutability

`final` on an array/list field only prevents **reassigning the reference** — it does nothing to protect the array's *contents*:

```
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String name) {
        this.name = name;
        this.phoneNumbers = new String[]{"111", "222"};
    }

    public String[] getPhoneNumbers() { return phoneNumbers; }
}

Person person = new Person("John");
person.getPhoneNumbers()[0] = "999999"; // mutates the "immutable" object!
```

```
this.phoneNumbers = new String[]{"123"}; // NOT allowed - final prevents reassignment
```

Even though the class looks immutable (all fields `final`, no setters), the caller can still reach into the array returned by the getter and mutate it in place.

Defensive copying

To actually protect mutable internal state, return (and store) **copies** rather than the original reference (Effective Java, Item 50):

```
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String name, String[] phoneNumbers) {
        this.name = name;
        this.phoneNumbers = phoneNumbers.clone(); // defensive copy
    }

    public String[] getPhoneNumbers() {
        return phoneNumbers.clone(); // defensive copy
    }
}
```