Java Mutability

Table of contents

Mutable vs immutable classes 1
Arrays and lists break naive immutability 3
Defensive copying 4

Introduced in 2026-03-12-object-oriented-programming-ii[1] (Lecture, Week 3). Builds on java-encapsulation[2].

Mutable vs immutable classes

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

Two types of class design:

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

Mutable

State is reachable through setters, so it can change at any point in the object's life:

```
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; }
}
```

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

```java
class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        p.setAge(20);

        p.setAge(30);   // state changed
    }
}
```

This object can change at any time.

Immutable

The immutable version changes four things at once: the class is `final`, both fields are `final`, the values arrive through the constructor, and there are no setters — only getters.

```java
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; }
}
```

```java
class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 20);
        Person p2 = new Person("Bob", 30);
        System.out.println(p1.getName() + " " + p1.getAge());
    }
}
```

Changing a value now means constructing a new object, not mutating the old one.

Why `final` on the class

A subclass can introduce mutability (**Effective Java (2nd ed.), Item 16, page 78**). If the class is subclassable, a subclass can add mutable state or override methods to return values that vary, so callers can no longer rely on "an instance of this type never changes". Marking the class `final` is what forecloses that.

Arrays and lists break naive immutability

Even if a class looks immutable, it may not be.

`final` on an array field only prevents **reassigning the reference**. It does nothing to protect the array's *contents*, and a getter that returns the field hands the caller a live handle on the object's own state:

```
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 getName() { return name; }

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

```
class Main {
    public static void main(String[] args) {
        Person person = new Person("John");
        person.getPhoneNumbers()[0] = "999999";

        System.out.println(person.getPhoneNumbers()[0]);   // 999999
    }
}
```

Every field is `private final` and there is not a setter in sight, yet the object's state changed. The rule:

- `final` for arrays means the **reference** cannot change, not the contents.
- `final` prevents *reassigning* the array:

```java
this.phoneNumbers = new String[]{"123"}; // NOT allowed if already assigned
```

Defensive copying

To protect mutable objects, create copies — on the way in *and* on the way out, so neither the constructor's caller nor the getter's caller keeps a handle on the field (**Effective Java, Item 50: "Make defensive copies when needed"**):

```java
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
    }
}
```

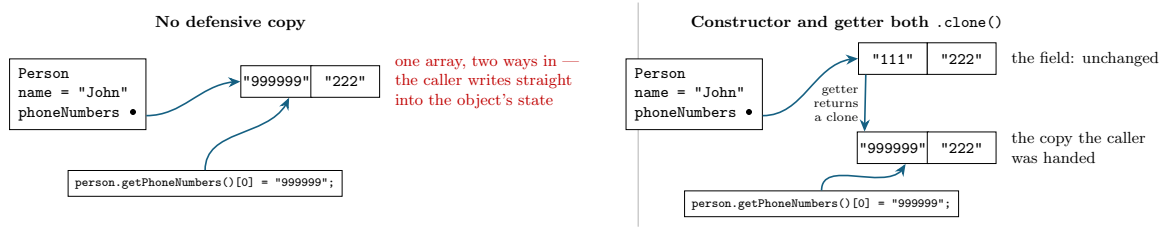

Figure 1: Why a class whose getter returns its array field is not immutable, and what `.clone()` changes. Left: the field and the getter's return value are the same array, so the caller's write lands in the object's state. Right: the getter hands out a copy, so the write lands somewhere the object does not care about.

`clone()` on an array is a shallow copy — it duplicates the row of references, not the objects they point at. That is enough here because `String` is itself immutable, but a `Person[]` of mutable `Person` objects would still leak.