How do we achieve immutability in Java?

Immutability means that once an object is created, its state (data) **cannot be changed**. To achieve this, we must ensure that no "setter" methods exist and that the object is fully initialized via the constructor. If the object holds references to other mutable objects (like a `List` or `Date`), we must protect them using **defensive copies**.[1] [2]

How to write your own Immutable Class? (The Rules)

To create a custom immutable class (like `String`), follow these **5 strict rules**:[3] [1]

1. **Class must be `final`:** So no one can extend it and override methods to change behavior.
2. **Fields must be `private` and `final`:** So they are initialized once and cannot be accessed directly.
3. **No Setter Methods:** Do not provide any methods that modify fields.
4. **Initialize all fields in the Constructor:** Pass all data when creating the object.
5. **Defensive Copying (Crucial Step):** If a field is a mutable object (like `List`, `Date`, or a custom mutable class), do **not** assign it directly. Instead, create a **new copy** (clone) of it in the constructor. Similarly, return a **copy** in the getter, not the original reference.[1]

Code Example: `ImmutableEmployee`

Here is a perfect example to show an interviewer, specifically highlighting the "Defensive Copy" part which is often the catch.

```
import java.util.*;

// Rule 1: Class is final
public final class ImmutableEmployee {

    // Rule 2: Fields are private and final
    private final int id;
    private final String name;
    private final List<String> skills; // Mutable field!

    // Rule 4: Initialize in Constructor
    public ImmutableEmployee(int id, String name, List<String> skills) {
        this.id = id;
        this.name = name;

        // Rule 5: Defensive Copy (Deep Copy) for mutable fields
        // If we did this.skills = skills; the caller could change the list later!
        this.skills = new ArrayList<>(skills);
    }

    // Rule 3: No Setters. Only Getters.

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public List<String> getSkills() {
        // Rule 5: Return a copy (or Unmodifiable view)
        // If we returned this.skills, the caller could .add() new skills!
        return Collections.unmodifiableList(skills);
    }
}
```

Interview Follow-up: Why use `record`?

"In modern Java (14+), we can use `record` types (e.g., `record Employee(String name) {}`). Records are **immutable by default**. They automatically provide final fields, a canonical constructor, and no setters, saving us from writing all this boilerplate code." [3]

⁂

1. https://www.digitalocean.com/community/tutorials/how-to-create-immutable-class-in-java
2. https://www.geekster.in/articles/immutable-class-in-java/
3. https://reflectoring.io/java-immutables/
4. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg
5. https://dzone.com/articles/java-and-immutability-avoid
6. https://www.linkedin.com/posts/er-arman-siddiqui_java-programming-cleancode-activity-7349451834735935489-jGNK
7. http://www.javapractices.com/topic/TopicAction.do?Id=29
8. https://www.codiga.io/blog/java-immutable-lists/
9. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg
10. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg
11. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg
12. https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html
13. https://stackoverflow.com/questions/29470658/best-design-approach-for-creating-immutable-class

14. https://www.youtube.com/watch?v=lO5r8EBdvYo