**Immutable Object with Builder Pattern:**

To make a Java object **immutable**, it means that once the object is created, its state (i.e., its fields) cannot be changed. This is done by following a few key principles:

1. **Final Fields**: Fields should be declared as final, so they can only be assigned once, either in the constructor or in the initialization block.
2. **No Setters**: Avoid creating setter methods, which would allow the fields to be modified after the object is created.
3. **Private Constructor**: The constructor should be private to ensure that the object can only be created via the Builder class.
4. **Immutable Object**: Once the object is created, its state cannot change.

**Example of Immutable Person Class Using Builder Pattern:**

public class Person {

private final String name; // final fields to ensure immutability

private final int age; // final fields to ensure immutability

private final String address; // final fields to ensure immutability

// Private constructor, to be used only by the Builder class

private Person(Builder builder) {

this.name = builder.name;

this.age = builder.age;

this.address = builder.address;

}

// Getter methods (no setters, to keep the object immutable)

public String getName() {

return name;

}

public int getAge() {

return age;

}

public String getAddress() {

return address;

}

// Builder Class for creating Person objects

public static class Builder {

private String name;

private int age;

private String address;

// Methods to set values for the fields

public Builder name(String name) {

this.name = name;

return this;

}

public Builder age(int age) {

this.age = age;

return this;

}

public Builder address(String address) {

this.address = address;

return this;

}

// The build method creates and returns the immutable Person object

public Person build() {

return new Person(this); } } }

**Example Usage of Person class which implemented Builder design Pattern:**

public class Main {

public static void main(String[] args) {

// Using the Builder pattern to create an immutable Person object

Person person = new Person.Builder()

.name("John Doe")

.address("123 Main St")

.build();

// Accessing values using getter methods

System.out.println(person.getName()); // John Doe

System.out.println(person.getAddress()); // 123 Main St

}

}

**Explanation:**

1. **final Fields**:  
   The fields name, age, and address are declared as final. This means once these fields are assigned a value, they cannot be changed.
2. **Private Constructor**:  
   The constructor is private to ensure that the object can only be created through the Builder class.
3. **No Setters**:  
   There are no setter methods, so you cannot modify the fields after the object is created.
4. **Builder Class**:  
   The Builder class is used to construct the Person object. It provides methods to set the values of the fields (like name, age, and address). Once all required fields are set, the build() method is called to create the immutable Person object.
5. **Immutability**:  
   Once the Person object is built using the Builder, its state cannot be changed. The values of name, age, and address are fixed for that object.

* In this code, person will have name as "John Doe" and address as "123 Main St". The age field is not set, so it will have the default value (likely 0 for an int).

**Summary of the Correct Immutable Object with Builder Pattern:**

1. **Immutability** is achieved by:
   * Declaring fields as final.
   * Using a **private constructor** to prevent external modification.
   * Providing only **getter methods** to access field values.
   * Using a **Builder class** for object creation, ensuring fields are set only once.
2. **The Builder Pattern** helps in constructing complex objects step-by-step, while keeping them immutable once created.
3. **No setter methods** and **final fields** ensure that the object’s state cannot change after it is created, making it **immutable**.

Now, I hope this explanation is much clearer. Let me know if you need further clarification!

4o mini

**You said:**

And also is @Builder is a part of Lombok library?

**ChatGPT said:**

ChatGPT

Yes, **@Builder** is part of the **Lombok** library.

Lombok is a Java library that helps reduce boilerplate code by automatically generating common code such as getters, setters, constructors, and builders at compile-time, based on annotations you use in your code.

**@Builder Annotation:**

* **Purpose**: The @Builder annotation generates a **Builder pattern** for the annotated class automatically. This allows you to create objects step-by-step in a fluent and readable way without writing a separate builder class.

**How It Works:**

When you annotate a class with @Builder, Lombok automatically generates the necessary code to implement the builder pattern. It creates:

* A **static inner Builder class** with setters for all the fields in the original class.
* A **build()** method to construct the object.

**Example with @Builder:**

java

Copy code

import lombok.Builder;

@Builder

public class Person {

private final String name;

private final int age;

private final String address;

// Getters

public String getName() {

return name;

}

public int getAge() {

return age;

}

public String getAddress() {

return address;

}

}

**Using @Builder:**

java

Copy code

public class Main {

public static void main(String[] args) {

Person person = Person.builder() // Using the builder() method generated by Lombok

.name("John Doe")

.address("123 Main St")

.build(); // Building the Person object

System.out.println(person.getName()); // John Doe

System.out.println(person.getAddress()); // 123 Main St

}

}

**Advantages of @Builder:**

* **Reduces Boilerplate**: Lombok takes care of the repetitive code needed for the builder pattern.
* **Fluent API**: You can use the builder in a fluent style, making the code clean and easy to read.
* **Immutability**: You can still create immutable objects with Lombok's builder pattern, as long as the fields are final.

**In Conclusion:**

* Yes, @Builder is part of the **Lombok** library, which helps reduce the effort of writing boilerplate code, especially for patterns like **Builder**.
* It provides a more concise and readable way to construct objects without manually implementing the builder pattern.

Let me know if you'd like further details on Lombok!