**📌 Objective:**

To understand whether changes made to an Address object after it is passed to a Person class **affect the internal state of Person**, based on how the object is stored and whether it is **copied or shared**.

**🧱 1. Scenario: Mutable Address Passed by Reference**

**Description:**

* Address is a mutable class (has setters).
* It is passed **directly** to the Person constructor without making a copy.

**🔧 Code:**

java

CopyEdit

class Address {

private String city;

public Address(String city) { this.city = city; }

public String getCity() { return city; }

public void setCity(String city) { this.city = city; }

}

class Person {

private final Address address;

public Person(Address address) {

this.address = address; // direct reference

}

public Address getAddress() { return address; }

}

**🧪 Main:**

java

CopyEdit

Address address = new Address("New York");

Person person = new Person(address);

address.setCity("Los Angeles"); // modifies original object

System.out.println(person.getAddress().getCity()); // Output: Los Angeles ✅

**✅ Conclusion:**

Changes to the original Address **WILL reflect** in the Person object — they both share the **same object reference**.

**🧱 2. Scenario: Mutable Address with Defensive Copy in Person**

**Description:**

* Address is mutable.
* Person makes a **defensive copy** in its constructor.
* Only values (not references) are copied.

**🔧 Code:**

java

CopyEdit

class Person {

private final Address address;

public Person(Address address) {

this.address = new Address(address.getCity()); // defensive copy

}

public Address getAddress() {

return address;

}

}

**🧪 Main:**

java

CopyEdit

Address address = new Address("New York");

Person person = new Person(address);

address.setCity("Los Angeles"); // changes original only

System.out.println(address.getCity()); // Output: Los Angeles

System.out.println(person.getAddress().getCity()); // Output: New York ❌

**✅ Conclusion:**

Person holds a **separate copy**, so changes to the original Address **do NOT affect** the Person object.

**🧱 3. Scenario: Immutable Address Class**

**Description:**

* Address is **immutable** — no setters, all fields final.
* Once created, its state cannot be changed.

**🔧 Code:**

java

CopyEdit

final class Address {

private final String city;

public Address(String city) {

this.city = city;

}

public String getCity() {

return city;

}

}

java

CopyEdit

class Person {

private final Address address;

public Person(Address address) {

this.address = address; // safe since Address is immutable

}

public Address getAddress() {

return address;

}

}

**🧪 Main:**

java

CopyEdit

Address address = new Address("New York");

Person person = new Person(address);

// address.setCity("Los Angeles"); // ❌ Compile error — no setter

System.out.println(person.getAddress().getCity()); // Output: New York

**✅ Conclusion:**

Address cannot be modified at all. No matter how it’s passed, it stays unchanged — **true immutability**.

**🧾 Final Summary Table:**

| **Scenario** | **Is Address mutable?** | **Does Person make a copy?** | **Will changes affect Person?** |
| --- | --- | --- | --- |
| 1. Mutable, passed by reference | ✅ Yes | ❌ No | ✅ Yes |
| 2. Mutable, defensive copy made | ✅ Yes | ✅ Yes | ❌ No |
| 3. Immutable class | ❌ No | ❌ Not needed | ❌ No |

**💡 Best Practice:**

* ✅ Use **defensive copies** if you need to **protect internal state** from external changes.
* ✅ Prefer **immutable objects** (no setters + final fields) for clean, thread-safe, and bug-resistant code.

### Scenario: ****Address is Immutable****

When Address is **immutable**, it means:

1. **No setters**: Fields cannot be changed after the object is created.
2. **Final fields**: All fields in the class are final and initialized only in the constructor.
3. **Object state is fixed**: Once an instance of Address is created, it cannot be modified.

### 🧱 1. ****Immutable Address Class****

#### **Key Points**:

* The Address class is **immutable**, meaning once an instance is created, its state cannot be modified.
* The constructor sets the values of the fields, and there are no setter methods.
* This ensures that **no changes** can be made to Address objects after they are created, regardless of how they are passed around.

#### **Code Example**:

java

CopyEdit

final class Address { // Immutable class

private final String city;

// Constructor sets the city

public Address(String city) {

this.city = city;

}

// Getter for city

public String getCity() {

return city;

}

}

class Person {

private final Address address; // Immutable reference to Address

public Person(Address address) {

this.address = address; // No copy, just reference (safe as Address is immutable)

}

public Address getAddress() {

return address; // Returns reference to the immutable Address object

}

}

#### **Explanation**:

* **No setters** in Address means the object is **immutable**.
* The Person class holds an immutable reference to the Address object. Even though the reference to Address is passed to Person, no one can modify the contents of the Address object after it's created.

#### **Main Method Example**:

java

CopyEdit

public class Main {

public static void main(String[] args) {

// Creating an immutable Address object

Address address = new Address("New York");

// Creating a Person object with the immutable Address

Person person = new Person(address);

// No way to change the address since it's immutable

// address.setCity("Los Angeles"); // Compile error — no setter

System.out.println(person.getAddress().getCity()); // Output: New York

}

}

### ✅ ****How Changes Work for Immutable Objects:****

* **No Setter Methods**: Since Address is immutable, you cannot modify its properties after it's created. Any attempt to change it will result in a **compile-time error** (e.g., trying to call a non-existent setter method).
* **Passing the Object**: When an Address is passed to a Person, the Person class holds a **reference** to the Address, not a copy. However, since Address is immutable, this **does not matter** because no one can modify it anyway.

### ✅ ****Summary for Immutable Address****:

* **Changes to Address after passing it to Person**: **Not possible** (since the object cannot be modified).
* **What happens if Person holds a reference to Address**: The reference is **safe** because Address cannot be modified.
* **Is it thread-safe?**: Yes, since its state cannot be changed, it is inherently thread-safe.