Immutable classes in Java

Mykola Shumyn

Apr 4 · 3 min read ★

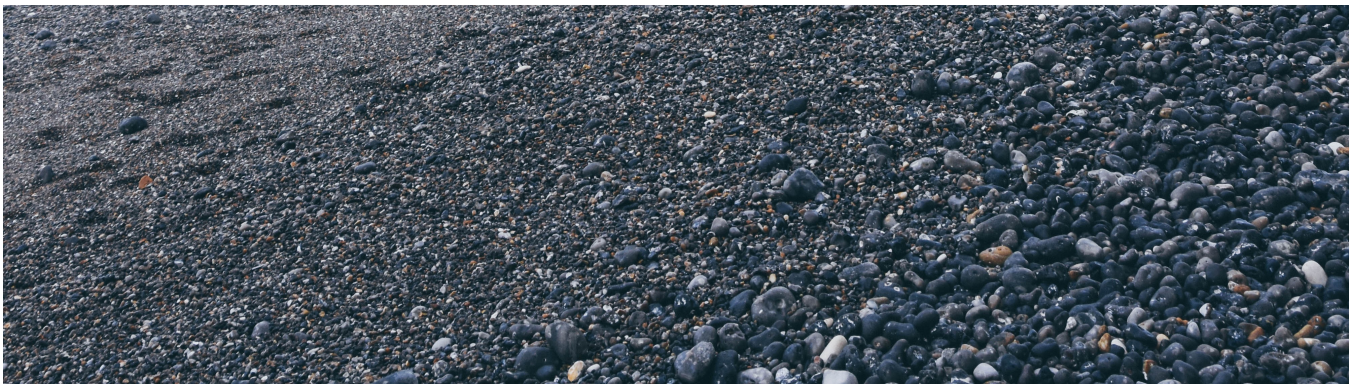

Photo by Emile Guillemot on Unsplash

Hi there!

Now we are going to talk about immutable classes in Java and show how to make the class immutable.

The class is called immutable if it is impossible to change its state and content after the initialization (creation).

The benefits of such classes are:

- **safety:** you, as a developer, can be sure that no one is able to change their state;
- **thread-safety:** the same as mentioned above is also actual for the multithreaded environment;
- **cacheable:** instances could be easily cached by VM cache or custom implementation, as we are 100% sure, that their values are not going to be changed;
- **hashable:** such classes could be safely put inside the hash collections (like `HashMap`, `HashSet` etc) - of course, if `equals()` & `hashCode()` methods are overridden in a proper way.

So, let's try to implement the immutable class by taking the mutable one and changing its implementation to the immutable step by step.

Consider, we have the following typical POJO class `Hobbit`:

```
1  import java.util.List;
2
3  public class Hobbit {
4      private String name;
```

```
    private Address address;
    private List<String> stuff;

    public String getName() {
        return name;
    }

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

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public List<String> getStuff() {
        return stuff;
    }

    public void setStuff(List<String> stuff) {
        this.stuff = stuff;
    }
}
```

Hobbit.java hosted with ❤ by GitHub view raw

`Hobbit` class has a field `address` of type `Address`:

```
public class Address {
    private String country;
    private String city;

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

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
```

```
    }

    public String getCity() {
        return city;
    }

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

Address.java hosted with ❤ by GitHub view raw

`Address` class is not going to be changed in this article, so, in case of any questions, please reference to the embedded code above.

The first issue to be noticed is setters, that allow changing the value of each class field. So let's remove those methods:

1. removing setters

```
public class Hobbit {
    private String name;
    private Address address;
    private List<String> stuff;

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }

    public List<String> getStuff() {
        return stuff;
    }
}
```

Hobbit.java hosted with ❤ by GitHub view raw

Looks better, but now the user of our class has no chances to set the values to class fields. So, let's provide the appropriate constructor:

2. adding all args constructor

```java
public class Hobbit {
    private String name;
    private Address address;
    private List<String> stuff;

    // all args constructor:
    public Hobbit(String name, Address address, List<String> stuff) {
        this.name = name;
        this.address = address;
        this.stuff = stuff;
    }

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }

    public List<String> getStuff() {
        return stuff;
    }
}
```

Hobbit.java hosted with ❤️ by GitHub view raw

Are we done now? Not yet, there are several issues left. First of all, our immutability could be hacked by using one of the OOP principles — inheritance. Let me demonstrate this with a simple example:

```java
import java.util.List;

public class Gandalf extends Hobbit {
    private String hackedName;

    public Gandalf(String name, Address address, List<String> stuff) {
        super(name, address, stuff);
        hackedName = name;
    }

    public void hackTheImmutability(String newNameValue) {
        hackedName = newNameValue;
```

```
        hackedName = newNameValue;
        System.out.println("Immutability has been hacked!");
    }

    @Override
    public String getName() {
        return hackedName;
    }
}
```

Gandalf.java hosted with ❤️ by GitHub view raw

```
import java.util.Collections;

public class Hack {
    public static void main(String[] args) {
        Gandalf gandalf = new Gandalf("Frodo Baggins", new Address("Hobitton", "Shire")
        Hobbit hobbit = (Hobbit) gandalf;

        System.out.println(hobbit.getName());

        System.out.println();
        gandalf.hackTheImmutability("Mr. Underhill");
        System.out.println();

        System.out.println(hobbit.getName());
    }
}
```

Hack.java hosted with ❤️ by GitHub view raw

If we run the `Hack` class, we'll get the following output:

```
Frodo Baggins

Immutability has been hacked!

Mr. Underhill
```

To fix this issue we have to:

3. marking class as final to protect it from being extended

```
public final class Hobbit {
```

```
    private String name;
    private Address address;
    private List<String> stuff;

    // all args constructor:
    public Hobbit(String name, Address address, List<String> stuff) {
        this.name = name;
        this.address = address;
        this.stuff = stuff;
    }

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }

    public List<String> getStuff() {
        return stuff;
    }
}
```

Hobbit.java hosted with ❤ by GitHub view raw

Now if we run `Hack` class one more time, the output will be correct:

```
Frodo Baggins

Immutability has been hacked!

Frodo Baggins
```

There is one more interesting issue left: as we know, in Java when we pass the value of an object, we are passing the reference to it. So the state of the objects could be changed by the side effects. Let's get back to the code to see this:

```
import java.util.List;
import java.util.ArrayList;

public class Hack {
    public static void main(String[] args) {
        Address address = new Address("Hobitton", "Shire");
```

```
        Address address = new Address("Hobitton", "Shire");
        List<String> stuff = new ArrayList<>();
        stuff.add("Sword");
        stuff.add("Ring of Power");

        Hobbit hobbit = new Hobbit("Frodo Baggins", address, stuff);

        System.out.println("Hobbit country: " + hobbit.getAddress().getCountry());
        System.out.println("Hobbit city: " + hobbit.getAddress().getCity());
        System.out.println("Hobbit stuff: " + hobbit.getStuff());

        address.setCountry("Isengard");
        address.setCity("Saruman tower");
        stuff.remove("Ring of Power");
        stuff.remove("Sword");

        System.out.println();
        System.out.println("Immutability has been hacked!");
        System.out.println();

        System.out.println("Hobbit country: " + hobbit.getAddress().getCountry());
        System.out.println("Hobbit city: " + hobbit.getAddress().getCity());
        System.out.println("Hobbit stuff: " + hobbit.getStuff());
    }
}
```

Hack.java hosted with ❤️ by GitHub view raw

The output of this code is:

```
Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]

Immutability has been hacked!

Hobbit country: Isengard
Hobbit city: Saruman tower
Hobbit stuff: []
```

This issue could be fixed by:

4. initializing all non-primitive mutable fields via constructor by performing a deep copy

```java
import java.util.List;
import java.util.ArrayList;

public final class Hobbit {
    private String name;
    private Address address;
    private List<String> stuff;

    // all args constructor:
    public Hobbit(String name, Address address, List<String> stuff) {
        this.name = name;
        this.address = new Address(address.getCountry(), address.getCity());
        this.stuff = new ArrayList<>(stuff);
    }

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }

    public List<String> getStuff() {
        return stuff;
    }
}
```

Hobbit.java hosted with ❤ by GitHub view raw

Now the output of `Hack` class is more predictable:

```
Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]

Immutability has been hacked!

Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]
```

But the same side effect issue could be produced through getter methods:

```java
import java.util.ArrayList;
```

```java
import java.util.List;

public class Hack {
    public static void main(String[] args) {
        Address address = new Address("Hobitton", "Shire");

        List<String> stuff = new ArrayList<>();
        stuff.add("Sword");
        stuff.add("Ring of Power");

        Hobbit hobbit = new Hobbit("Frodo Baggins", address, stuff);

        System.out.println("Hobbit country: " + hobbit.getAddress().getCountry());
        System.out.println("Hobbit city: " + hobbit.getAddress().getCity());
        System.out.println("Hobbit stuff: " + hobbit.getStuff());

        Address hobbitAddress = hobbit.getAddress();
        hobbitAddress.setCountry("Isengard");
        hobbitAddress.setCity("Saruman tower");

        List<String> hobbitStuff = hobbit.getStuff();
        hobbitStuff.remove("Ring of Power");
        hobbitStuff.remove("Sword");

        System.out.println();
        System.out.println("Immutability has been hacked!");
        System.out.println();

        System.out.println("Hobbit country: " + hobbit.getAddress().getCountry());
        System.out.println("Hobbit city: " + hobbit.getAddress().getCity());
        System.out.println("Hobbit stuff: " + hobbit.getStuff());
    }
}
```


Output:

```
Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]

Immutability has been hacked!

Hobbit country: Isengard
Hobbit city: Saruman tower
Hobbit stuff: []
```

So, it goes with another step to be done:

5. performing cloning of the returned non-primitive mutable object in getter methods

```java
import java.util.ArrayList;
import java.util.List;

public final class Hobbit {
    private String name;
    private Address address;
    private List<String> stuff;

    // all args constructor:
    public Hobbit(String name, Address address, List<String> stuff) {
        this.name = name;
        this.address = new Address(address.getCountry(), address.getCity());
        this.stuff = new ArrayList<>(stuff);
    }

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return new Address(address.getCountry(), address.getCity());
    }

    public List<String> getStuff() {
        return new ArrayList<>(stuff);
    }
}
```

Hobbit.java hosted with ❤️ by GitHub view raw

Now the `Hack` output is:

```
Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]

Immutability has been hacked!
```

```
Hobbit country: Hobitton
Hobbit city: Shire
Hobbit stuff: [Sword, Ring of Power]
```

There is one more optional step to be done, but if all other prerequisites are met, it is not necessary. Anyway, let's mentioned it:

6. marking all class fields as final (optional)

```java
import java.util.ArrayList;
import java.util.List;

public final class Hobbit {
    private final String name;
    private final Address address;
    private final List<String> stuff;

    // all args constructor:
    public Hobbit(String name, Address address, List<String> stuff) {
        this.name = name;
        this.address = new Address(address.getCountry(), address.getCity());
        this.stuff = new ArrayList<>(stuff);
    }

    // getters:
    public String getName() {
        return name;
    }

    public Address getAddress() {
        return new Address(address.getCountry(), address.getCity());
    }

    public List<String> getStuff() {
        return new ArrayList<>(stuff);
    }
}
```

Hobbit.java hosted with ❤ by GitHub view raw

And that's it, we are done — our `Hobbit` class is immutable now.

Let's summarize out steps:

1. *removing setters;*
2. *adding all args constructor;*
3. *marking the class as `final` to protect it from being extended;*
4. *initializing all non-primitive mutable fields via constructor by performing a deep copy;*
5. *performing cloning of the returned non-primitive mutable object in getter methods;*
6. *marking all class fields as `final` (optional step).*

Of course, there are other ways to make class immutable (for example, using Builder pattern), but it is out of the scope of the current story.

Have fun!