How to Create an Immutable Class in Java

by Hussein Terek MVB · **Dec. 12, 17 · Java Zone · Research**

An object is immutable if its state cannot change after construction. Immutable objects don't expose any way for other objects to modify their state; the object's fields are initialized only once inside the constructor and never change again.

In this article, we'll define the typical steps for creating an immutable class in Java and also shed light on the common mistakes which are made by developers while creating immutable classes.

1. Usage of Immutable Classes

Nowadays, the *"must-have"* specification for every software application is to be distributed and multi-threaded—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the *Synchronized* blocks whenever they modify the state of an object.

With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn't worry about concurrent modifications.

2. Some Popular Immutable Classes

String is the most popular immutable class in Java. Once initialized its value cannot be modified. Operations like ***trim(), substring(), replace()*** always return a new instance and don't affect the current instance, that's why we usually call ***trim()*** as the following:

```
1  String alex = "Alex";
2  alex = alex.trim();
```

Another example from JDK is the wrapper classes like: ***Integer, Float, Boolean*** ... these classes don't modify their state, however they create a new instance each time you try to modify them.

```
1  Integer a =3;
```

```
a += 3;
```

After calling ***a += 3,*** a new instance is created holding the value: 6 and the first instance is lost.

3. How Do We Create an Immutable Class

In order to create an immutable class, you should follow the below steps:

1. Make your class ***final,*** so that no other classes can extend it.
2. Make all your fields ***final,*** so that they're initialized only once inside the constructor and never modified afterward.
3. Don't expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.
5. If the class holds a mutable object:
    - Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.
    - Make sure to always return a clone copy of the field and never return the real object instance.

3.1. Simple Immutable Class

Let's follow the above steps and create our own immutable class (***ImmutableStudent.java***).

```
package com.programmer.gate.beans;

public final class ImmutableStudent {

    private final int id;
    private final String name;

    public ImmutableStudent(int id, String name) {
        this.name = name;
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

The above class is a very simple immutable class which doesn't hold any mutable object and never expose its fields in any way; these type of classes are normally used for caching purposes.

3.2. Passing Mutable Objects to Immutable Class

Now, let's complicate our example a bit, we create a mutable class called ***Age*** and add it as a field to ***ImmutableStudent:***

```
package com.programmer.gate.beans;

public class Age {

    private int day;
    private int month;
    private int year;

    public int getDay() {
        return day;
    }

    public void setDay(int day) {
    this.day = day;
    }

    public int getMonth() {
    return month;
    }

    public void setMonth(int month) {
    this.month = month;
    }

    public int getYear() {
    return year;
    }

    public void setYear(int year) {
    this.year = year;
    }

}
```

```
package com.programmer.gate.beans;

public final class ImmutableStudent {

```

```
    private final int id;
    private final String name;
    private final Age age;

    public ImmutableStudent(int id, String name, Age age) {
    this.name = name;
    this.id = id;
    this.age = age;
    }

    public int getId() {
    return id;
    }

    public String getName() {
    return name;
    }

    public Age getAge() {
    return age;
    }
}
```

So, we added a new mutable field of type ***Age*** to our immutable class and assign it as normal inside the constructor.

Let's create a simple test class and verify that ***ImmutableStudent*** is no more immutable:

```
public static void main(String[] args) {

    Age age = new Age();
    age.setDay(1);
    age.setMonth(1);
    age.setYear(1992);
    ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

    System.out.println("Alex age year before modification = " + student.getAge().getYear());
    age.setYear(1993);
    System.out.println("Alex age year after modification = " + student.getAge().getYear());
}
```

After running the above test, we get the following output:

```
Alex age year before modification = 1992
Alex age year after modification = 1993
```

We claim that ***ImmutableStudent*** is an immutable class whose state is never modified after construction, however in the above example we are able to modify the age of ***Alex*** even after constructing ***Alex*** object. If we go back to the implementation of ***ImmutableStudent*** constructor, we find that *age* field is being assigned to the instance of the **Age** argument, so whenever the referenced ***Age*** is modified outside the class, the change is reflected directly on the state of ***Alex.*** Check out my Pass by value OR pass by reference article to more deeply understand this concept.

In order to fix this and make our class again immutable, we follow step **#5** from the steps that we mention above for creating an immutable class. So we modify the constructor in order to clone the passed argument of ***Age*** and use a clone instance of it.

```
public ImmutableStudent(int id, String name, Age age) {
    this.name = name;
    this.id = id;
    Age cloneAge = new Age();
    cloneAge.setDay(age.getDay());
    cloneAge.setMonth(age.getMonth());
    cloneAge.setYear(age.getYear());
    this.age = cloneAge;
}
```

Now, if we run our test, we get the following output:

```
Alex age year before modification = 1992
Alex age year after modification = 1992
```

As you see now, the age of ***Alex*** is never affected after construction and our class is back to immutable.

3.3. Returning Mutable Objects From Immutable Class

However, our class still has a leak and is not fully immutable, let's take the following test scenario:

```
public static void main(String[] args) {

    Age age = new Age();
    age.setDay(1);
    age.setMonth(1);
    age.setYear(1992);
    ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

    System.out.println("Alex age year before modification = " + student.getAge().getYear());
    student.getAge().setYear(1993);
    System.out.println("Alex age year after modification = " + student.getAge().getYear());
}
```

Output:

```
Alex age year before modification = 1992
```

```
Alex age year after modification = 1993
```

Again according to step **#4**, when returning mutable fields from immutable object, you should return a clone instance of them and not the real instance of the field.

So we modify ***getAge()*** in order to return a clone of the object's age:

```
public Age getAge() {
    Age cloneAge = new Age();
    cloneAge.setDay(this.age.getDay());
    cloneAge.setMonth(this.age.getMonth());
    cloneAge.setYear(this.age.getYear());

    return cloneAge;
}
```

Now the class becomes fully immutable and provides no way or method for other objects to modify its state.

```
Alex age year before modification = 1992
Alex age year after modification = 1992
```

4. Conclusion

Immutable classes provide a lot of advantages especially when used correctly in a multi-threaded environment. The only disadvantage is that they consume more memory than the traditional class since upon each modification of them a new object is created in the memory... but, a developer should not overestimate the memory consumption as its negligible compared to the advantages provided by these type of classes.

Finally, an object is immutable if it can present only one state to the other objects, no matter how and when they call its methods. If so it's thread safe by any definition of thread-safe.

Like This Article? Read More From DZone

Encapsulation: I Don't Think it Means What You Think it Means

Protect Your Immutable Object Invariants in More Complex Java Objects

The Importance of Immutability in Java

Free DZone Refcard
Getting Started With Kotlin

Topics: JAVA , IMMUTABLE , JAVA CORE

Published at DZone with permission of Hussein Terek , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

Java Partner Resources