Immutable Object

Dr Kafi Rahman

Assistant Professor @CS

Truman State University

Background

- 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.

Usage

- 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.

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:
  - String alex = "Alex";
  - alex = alex.trim();
- Another example from JDK is the wrapper classes like: Integer, Float, Boolean …
  - these classes don't modify their state
  - they create a new instance each time you try to modify them.
    - Integer a =3;
    - a += 3;
  - After calling a += 3, a new instance is created holding the value: 6 and the first instance is lost.

Steps to Create an Immutable Class

- Make your class final, so that no other classes can extend it.
- Make all your fields final, so that they're initialized only once inside the constructor and never modified afterward.
- Don't expose setter methods.
- When exposing methods which modify the state of the class, you must always return a new instance of the class.
- 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.

Simple Immutable Class

```
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;
    }
}
```

Passing Mutable Objects to Immutable Class

- We create a mutable class called Age and add it as a field to ImmutableStudent:

```
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;
   }

}
```

Passing Mutable Objects to Immutable Class

- We added a new mutable field of type Age to our immutable class and assign it as normal inside the constructor

```java
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;
   }
}
```

Passing Mutable Objects to Immutable Class

- 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());
}
```

Passing Mutable Objects to Immutable Class

- 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.
- 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.
- 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.
    - Make sure to always return a clone copy of the field and never return the real object instance.
    - 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

Passing Mutable Objects to 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;
}
```

Returning Mutable Objects From Immutable Class

- The 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());
}
```

Returning Mutable Objects From Immutable Class

- The 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());
}
```

Returning Mutable Objects From Immutable Class

- 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.
    - When exposing methods which modify the state of the class, you must always return a new instance of the class.

Returning Mutable Objects From Immutable Class

- 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;
}
```

Conclusion

- 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.
- 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
  - however, its negligible compared to the advantages provided by these type of classes.

Thank you

Please let me know if you have any questions.

Questions?