**How to Make any Java Class Immutable**

October 28, 2012 by Lokesh Gupta

An [immutable class](https://en.wikipedia.org/wiki/Immutable_object) is one whose state cannot be changed once created. There are certain guidelines to create an class immutable.

In this post, we will revisit these guidelines.

**Benefits of making a class immutable**

Lets first identify benefits of making a class immutable. Immutable classes are

1. are simple to construct, test, and use
2. are automatically thread-safe and have no synchronization issues
3. do not need a copy constructor
4. do not need an implementation of clone
5. allow [hashCode](https://howtodoinjava.com/core-java/related-concepts/working-with-hashcode-and-equals-methods-in-java/" \o "Working with hashCode and equals methods in java) to use lazy initialization, and to cache its return value
6. do not need to be copied defensively when used as a field
7. make good [Map keys and Set elements](https://howtodoinjava.com/core-java/collections/how-hashmap-works-in-java/) (these objects must not change state while in the collection)
8. have their class invariant established once upon construction, and it never needs to be checked again
9. always have “**failure atomicity**” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state

**Guidelines to make a class immutable**

Java documentation itself has some guidelines identified [in this link](http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html). We will understand what they mean actually:

**1) Don’t provide “setter” methods — methods that modify fields or objects referred to by fields.**

This principle says that for all mutable properties in your class, do not provide setter methods. Setter methods are meant to change the state of object and this is what we want to prevent here.

**2) Make all fields final and private**

This is another way to increase immutability. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.

**3) Don’t allow subclasses to override methods**

The simplest way to do this is to declare the class as ***final***. Final classes in java can not be overridden.

**4) Special attention when having mutable instance variables**

Always remember that your instance variables will be either **mutable or immutable**. Identify them and **return new objects with copied content for all mutable objects**. Immutable variables can be returned safely without extra effort.

A more sophisticated approach is to make the constructor ***private*** and construct instances in **factory methods**.

Lets add all above rules and make something concrete class implementation.

|  |
| --- |
| import java.util.Date;    /\*\*  \* Always remember that your instance variables will be either mutable or immutable.  \* Identify them and return new objects with copied content for all mutable objects.  \* Immutable variables can be returned safely without extra effort.  \* \*/  public final class ImmutableClass  {        /\*\*      \* Integer class is immutable as it does not provide any setter to change its content      \* \*/      private final Integer immutableField1;      /\*\*      \* String class is immutable as it also does not provide setter to change its content      \* \*/      private final String immutableField2;      /\*\*      \* Date class is mutable as it provide setters to change various date/time parts      \* \*/      private final Date mutableField;        //Default private constructor will ensure no unplanned construction of class      private ImmutableClass(Integer fld1, String fld2, Date date)      {          this.immutableField1 = fld1;          this.immutableField2 = fld2;          this.mutableField = new Date(date.getTime());      }        //Factory method to store object creation logic in single place      public static ImmutableClass createNewInstance(Integer fld1, String fld2, Date date)      {          return new ImmutableClass(fld1, fld2, date);      }        //Provide no setter methods        /\*\*      \* Integer class is immutable so we can return the instance variable as it is      \* \*/      public Integer getImmutableField1() {          return immutableField1;      }        /\*\*      \* String class is also immutable so we can return the instance variable as it is      \* \*/      public String getImmutableField2() {          return immutableField2;      }        /\*\*      \* Date class is mutable so we need a little care here.      \* We should not return the reference of original instance variable.      \* Instead a new Date object, with content copied to it, should be returned.      \* \*/      public Date getMutableField() {          return new Date(mutableField.getTime());      }        @Override      public String toString() {          return immutableField1 +" - "+ immutableField2 +" - "+ mutableField;      }  } |

Now its time to test our class:

|  |
| --- |
| class TestMain  {      public static void main(String[] args)      {          ImmutableClass im = ImmutableClass.createNewInstance(100,"test", new Date());          System.out.println(im);          tryModification(im.getImmutableField1(),im.getImmutableField2(),im.getMutableField());          System.out.println(im);      }        private static void tryModification(Integer immutableField1, String immutableField2, Date mutableField)      {          immutableField1 = 10000;          immutableField2 = "test changed";          mutableField.setDate(10);      }  }    Output: (content is unchanged)    100 - test - Tue Oct 30 21:34:08 IST 2012  100 - test - Tue Oct 30 21:34:08 IST 2012 |

As it can be seen that even changing the instance variables using their references does not change their value, so the class is immutable.