

Creating an Immutable Class in Java

📅 Last Updated: January 29, 2022 👤 By: Lokesh Gupta 📁 Java Basics 🔖 Java Immutability

An [immutable class](#) is one whose state can not be changed once created. There are certain guidelines to **create a class immutable in Java**.

In this post, we will revisit these guidelines.

Table of Contents

- 1. Rules to create immutable classes
- 2. Java immutable class example
- 3. Benefits of making a class immutable
- 5. Summary

1. Rules to create immutable classes

Java documentation itself has some guidelines identified to [write immutable classes](#) in this link. We will understand what these guidelines mean actually by creating an immutable class with mutable object with **Date** field.

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 an 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 extended.

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**.

2. Java immutable class example

Lets apply all above rules for immutable classes and make a concrete class implementation for **immutable class in Java**.

```
ImmutableClass.java
```

```
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 con
```

```
* */
private final Integer immutableField1;

/**
 * String class is immutable as it also does not provide setter to change its con
 * */
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 d
{
    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:

TestMain.java

```
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.getMutableF
        System.out.println(im);
    }

    private static void tryModification(Integer immutableField1, String immutableFie
    {
        immutableField1 = 10000;
        immutableField2 = "test changed";
        mutableField.setDate(10);
    }
}
```

Program output:

Console

```
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.

Immutable classes in JDK

Apart from your written classes, JDK itself has lots of immutable classes. Given is such a list of immutable classes in Java.

1. String
2. Wrapper classes such as Integer, Long, Double etc.
3. Immutable collection classes such as Collections.singletonMap() etc.
4. java.lang.StackTraceElement
5. Java enums (ideally they should be)
6. java.util.Locale
7. java.util.UUID

3. Benefits of making a class immutable

Lets first identify **advantages of immutable class**. In Java, 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()` 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** (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

4. Summary

In this tutorial, we learned to **create immutable java class with mutable objects** as well as immutable fields as well. We also saw the benefits which immutable classes bring in an application.

As a design best practice, always aim to make your application Java classes to be immutable. In this way, you can always worry less about [concurrency](#) related defects in your program.

How to write an immutable class? This could be an [interview question](#) as well.

Happy Learning!!

Read More:

[Why string class is immutable in Java?](#)

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. [\[Solved\]: javax.xml.bind.JAXBException: class java.util.ArrayList nor any of its super class is known to this context](#)
2. [Why Strings are Immutable in Java?](#)
3. [Immutable Collections with Factory Methods in Java 9](#)
4. [Creating a Temporary File in Java](#)

5. [Creating Threads Using java.util.concurrent.ThreadFactory](#)
6. [Creating a New File in Java](#)
7. [Creating Streams in Java](#)
8. [Creating Infinite Streams in Java](#)
9. [Creating New Directories in Java](#)
0. [Creating Password Protected Zip with Zip4J](#)

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

108 thoughts on “Creating an Immutable Class in Java”

Sudhakar

July 12, 2019 at 2:22 pm

Why do you make constructor private in ImmutableClass.java.

[Reply](#)

Lokesh Gupta

July 13, 2019 at 6:46 am

It is optional if you do not provide factory methods/builder pattern.

[Reply](#)

vallimayil

November 1, 2018 at 11:27 pm

Hi,

In code snippet, method is invoked which is pass by value and not pass by reference, sotryModification should return an object and it should be assigned back to variable im to check if class is mutable.

```
im = sotryModification( im.getImmutableField1(),  
                        im.getImmutableField2(),  
                        im.getMutableField());  
System.out.println(im);
```

[Reply](#)

jeet

February 20, 2019 at 10:50 pm

Exactly.. 😊

[Reply](#)

Nishant

September 19, 2018 at 1:00 am

I have two doubts regarding this immutability example.

(1) Why you are creating new Date objects at two places: one in constructor and other one in getMutableField() method? If I simply do this.mutableField = date.getTime() in constructor and in getMutableField() return new Date() instance , then the output remains same . Means still object is immutable

(2) Also the hashCode of all new Date object in your code is same if you print it. Means any change in the object is going to be reflected everywhere. So does it ensures immutability?

[Reply](#)

Lokesh Gupta

September 19, 2018 at 5:11 am

Hi Nishant, thanks for sharing your thoughts.

- 1) Any implementation is good as far as you don't break the immutability. I really don't understand your first point. Probably you can share the code.
- 2) Hash code only comes into scene when you start using it as key in maps. And this is never my intention.

[Reply](#)

Chirag

May 19, 2017 at 4:31 pm

```
final class Immutable {  
    private final int a;  
    private final String s;  
    private final Chirag ss;  
    public Immutable (int a1, String s1, Chirag ss1){  
        this.a = a1;  
        this.s = s1;  
        this.ss = ss1;  
    }  
    public Integer geta() {  
        return a;  
    }  
}
```

```
}  
public String gets() {  
    return s;  
}  
  
public String toString() {  
    return a + " - " + s + " - " + ss.age;  
}  
private static void modification(int a2, String s2, Chirag ss2){  
    a2 = 20;  
    ss2.age = 11;  
    s2 = "changed";  
}  
public static void main(String[] args){  
    Immutable im = new Immutable(14, "age", new Chirag(20));  
    System.out.println(im);  
    modification(im.a, im.s, im.ss);  
    System.out.println(im);  
}  
}  
  
class Chirag  
{  
    public int age;  
  
    Chirag(int s)  
    {  
        this.age = s;  
    }  
}
```

Output: 14 - age - 20
 14 - age - 11

What do i do so that the ss.age value does not change?

[Reply](#)

Rahul Singh

July 13, 2017 at 12:45 pm

```
package com.cic.testcase;

public class MyImmutable {
    private final Integer a;
    private final String s;
    private final Demo ss;

    private MyImmutable(int a1, String s1, Demo ss1) {
        this.a = a1;
        this.s = s1;
        this.ss = new Demo(ss1.getAge());
    }

    // factory method for obj creation
    public static MyImmutable newInstance(int a1, String s1, Demo ss1) {
        return new MyImmutable(a1, s1, ss1);
    }

    public Integer getA() {
        return a;
    }

    public String getS() {
        return s;
    }

    public Demo getSs() {
        return new Demo(ss.getAge());
    }

    @Override
    public String toString() {
        return "MyImmutable [a=" + a + ", s=" + s + ", ss=" + ss + "];"
    }
}

package com.cic.testcase;

public class Demo {
    public int age;

    Demo(int s)
    {
        this.age = s;
    }

    int getAge() {
        return age;
    }
}
```

```
@Override
public String toString() {
    return "Demo [age=" + age + "]";
}

}

package com.cic.testcase;

public class Test {

    public static void main(String[] args){
        MyImmutable im = MyImmutable.newInstance(14, "age", new Demo(20));
        System.out.println(im);
        modification(im.getA(), im.getS(), im.getSs());
        System.out.println(im);
    }

    private static void modification(int a2, String s2, Demo ss2){
        a2 = 20;
        ss2.age = 11;
        s2 = "changed";
    }

}
```

OUTPUT:

MyImmutable [a=14, s=age, ss=Demo [age=20]]

MyImmutable [a=14, s=age, ss=Demo [age=20]]

[Reply](#)

avinash

November 27, 2016 at 12:32 am

Hello Lokesh

please explain memory management in java and java class loading system
these topics are so confusing.

[Reply](#)

Sradha

[November 5, 2016 at 3:18 pm](#)

Could please paste some sample code to make a list field immutable

```
private List mutataList;
```

[Reply](#)

Lokesh Gupta

[November 7, 2016 at 11:57 am](#)

```
List unmodifiableList = Collections.unmodifiableList(list);  
Return unmodifiableList;
```

[Reply](#)

anket

[October 31, 2016 at 8:48 pm](#)

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());  
}
```

Why we need new Date object for mutable field?

[Reply](#)

anket

[October 31, 2016 at 10:33 pm](#)

How new Date(mutableField.getTime()) is different from mutableField.getTime();

[Reply](#)

Lokesh Gupta

[November 1, 2016 at 1:34 pm](#)

It's about new Date(mutableField.getTime()).setTime() and mutableField.setTime(). You see the difference? In first way, mutableField will not change, second way it will change.

[Reply](#)

Lokesh Gupta

[November 1, 2016 at 1:32 pm](#)

Because we don't want to "leak" the reference of original date object to prevent any modification on it.

[Reply](#)

Karan

[July 29, 2016 at 11:17 am](#)

Hi

Can't i create a class immutable by not using final at all ?

For eg :-

1. make the constructor private & all all variables private
2. Now make a public method createInstance() which returns new object of the class

Is this class not immutable ?

correct me if i am wrong ...

Thanks

Karan

[Reply](#)

Lokesh Gupta

[July 29, 2016 at 12:15 pm](#)

Hi Karan, yes it's possible to achieve immutability by your approach as well. In fact, there is no MUST DO guidelines for this.

[Reply](#)

Venkatesh

September 12, 2016 at 3:09 am

Think after get the final Date instance, if I call setTime to change value.

[Reply](#)

Cuong

June 13, 2016 at 7:50 pm

Hi guys,

First of all, this is interesting article, but I get stuck at one point, it is: "A more sophisticated approach is to make the constructor private and construct instances in factory methods.". If I declare the constructor with 3 parameters is public (don't have createNewInstance method), I think it's still immutable class, if not, could you give me an example to prove my fault, please?

Thanks in advance.

Cuong.

[Reply](#)

Lokesh Gupta

June 14, 2016 at 12:27 pm

Yes, it will be immutable class. "Making constructor private" is not mandatory requirement.

[Reply](#)

Shubham Gupta

[April 22, 2016 at 12:24 pm](#)

```
public final class ImmutableClass {

    private final Integer immutableField1;

    private final String 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.
     */
    private Date mutableField; // I did not put final here

    public ImmutableClass(Integer field1, String field2, Date date){
        this.immutableField1 = field1;
        this.immutableField2 = field2;
        //this is very important else what would happen is if we change
        this.mutableField = new Date(date.getTime());
    }

    public Integer getIntImmutableField(){
        return immutableField1;
    }

    public String getStringImmutableField(){
        return immutableField2;
    }

    public Date getMutableField(){
        return mutableField;
    }
}
```

MAIN class

```
public class App {  
  
    public static void main(String[] args) throws ParseException {  
        ImmutableClass immutableClass = new ImmutableClass(100,"test", new  
        Integer intTest = immutableClass.getIntImmutableField();  
        Date date = immutableClass.getMutableField();  
        System.out.println("Date associated with object:"+date);  
        SimpleDateFormat sdf = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");  
        String dateInString = "31-08-1982 10:20:56";  
        date = sdf.parse(dateInString);  
        System.out.println("New Date:"+date);  
        System.out.println("Date associated with object:"+immutableClass.ge  
    }  
}
```

Output:

```
Date associated with object:Fri Apr 22 12:18:24 IST 2016  
New Date:Tue Aug 31 10:20:56 IST 1982  
Date associated with object:Fri Apr 22 12:18:24 IST 2016
```

As you can see that I have removed final modifier from Date object which is mutable by default. But even after that when I try to get the reference of date from object and modify the date, I am not able to do it. Any idea why this is happening ?

[Reply](#)

Lokesh Gupta

April 22, 2016 at 12:51 pm

You are not changing the date. Rather you are assigning the reference of new date object to a local date reference (not of date object inside immutable class). Try this:

```
//date = sdf.parse(dateInString);  
date.setTime(sdf.parse(dateInString).getTime()); //Directly use the referenc
```

Integer is immutable because it does not provide any setter method – so even if you have it's reference you cannot change the content. Date class provide setter methods – so it's mutable.

[Reply](#)

Shubham Gupta

[April 23, 2016 at 1:19 am](#)

Thank You :). final & immutability is very clear now !!!

[Reply](#)

Devendra

[May 7, 2016 at 9:22 pm](#)

Thank you !!!!

[Reply](#)

Shafali

March 12, 2016 at 3:06 pm

Hi lokesh,

I have couple of question in regards to the Immutability. Please see if you can help out-

If I don't follow the sophisticated approach of writing a private constructor/any constructor and just write the code using first 4 points (I mean not at all writing any constructor) and an object of Immutable class is made by using the default constructor, Then

A. How will Immutability of the class be maintained as the values of properties are not set anywhere?

[Reply](#)

subose

February 25, 2016 at 2:18 am

Can you explain the following code.

```
public final class TestImmutable {  
    private final String test;  
    private final String test1;  
    private final Date dd;  
    TestImmutable(String s, String t, Date dd){  
        this.test=s;  
        this.test1=t;  
        this.dd=dd;  
    }  
}
```

```
}  
  
public String getTest() {  
    return test;  
}  
  
public String getTest1() {  
    return test1;  
}  
  
public Date getDd() {  
    return dd;  
}  
  
@Override  
public String toString(){  
    return test+"-"+test1+"-"+dd;  
}  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        TestImmutable ti=new TestImmutable("subose", "chandra", new Date());  
        System.out.println(ti);  
        Date dd=ti.getDd();  
        String s=ti.getTest1();  
        s="Gaddala";  
        dd.setDate(10);  
        System.out.println(ti); // Date get Changed  
  
    }  
  
}
```

[Reply](#)

Lokesh Gupta

February 25, 2016 at 6:22 am

String is immutable class but Date is mutable class. You should return a new date object from `getDd()` method, that will prevent the original date being changed.

[Reply](#)**Memosha Sinha**

January 12, 2016 at 5:28 am

Hi Lokesh, I have a question – As you mention, I made my constructor as private and provided a factory method which gives me instance of class

```
private final String name;
```

```
private final int id;
```

```
private final Integer doorId;
```

```
private final Date dateOfJoining;
```

```
public static ImmutablePOJO getObject(String name,int id, Integer doorId,Date dateOfJoining) {
```

```
// TODO Auto-generated method stub
```

```
return new ImmutablePOJO(name, id, doorId, dateOfJoining);
```

```
}
```

```
private ImmutablePOJO( String name,int id, Integer doorId,Date dateOfJoining){
```

```
this.name=name;
```

```
this.id=id;
```

```
this.doorId=doorId;
```

```
//this.dateOfJoining=dateOfJoining;
```

```
this.dateOfJoining=new Date(dateOfJoining.getTime());
}

public Date getDateOfJoining() {
//Below line of code will allow to modify the mutable fields
return dateOfJoining;
//This piece of code helps us in maintaining the same value for mutable field
//return new Date(dateOfJoining.getTime());
}

public static void main(String[] args) {
// TODO Auto-generated method stub
ImmutablePOJO immutablePOJO= ImmutablePOJO.getObject("memosha", 100,
100, new Date());
System.out.println("Before calling to the method to change the value ");
System.out.println(immutablePOJO.toString());
//Call the method to change the vale and see the effect
changeValue(immutablePOJO.getDateOfJoining());
System.out.println(immutablePOJO.toString());

}

public static void changeValue(Date dateOfJoining){
//So the value of mutable field is changing if we pass the direct reference
dateOfJoining.setDate(10);

}

Still I am able to change the value of Date dateOfJoining
```

[Reply](#)

Lokesh Gupta

January 12, 2016 at 7:03 am

Comment out – return dateOfJoining;

Uncomment this – //return new Date(dateOfJoining.getTime());

[Reply](#)

Memosha Sinha

January 14, 2016 at 7:04 am

```
private final String name;
private final int id;
private final Integer doorId;
private final Date dateOfJoining;
private Student(String name,int id, Integer doorId,Date dateOfJoining){
this.name=name;
this.id=id;
this.doorId=doorId;
//this.dateOfJoining=dateOfJoining;
this.dateOfJoining=new Date(dateOfJoining.getTime());
}
```

```
public static Student getObject(String name,int id, Integer doorId,Date
dateOfJoining) {
// TODO Auto-generated method stub
return new Student(name, id, doorId, dateOfJoining);
}
```

When I call the below statements still I am able to change the value

```
public static void main(String[] args) {
// TODO Auto-generated method stub
Student student= Student.getObject("memosha", 100, 100, new Date());
```

```
System.out.println("Before calling to the method to change the value ");
System.out.println(student.toString());
//Call the method to change the vale and see the effect
changeValue(student.getDateOfJoining());
System.out.println(student.toString());

}

public static void changeValue(Date dateOfJoining){
//So the value of mutable field is changing if we pass the direct reference
dateOfJoining.setDate(1000);

}
```

Before calling to the method to change the value

name =memosha ,id =100, doorId100, dateOfJoiningThu Jan 14 12:31:43 IST
2016

After calling the method

name =memosha ,id =100, doorId100, dateOfJoiningWed Sep 26 12:31:43 IST
2018.

Please verify and explain me this.

[Reply](#)

yogesh

November 24, 2015 at 2:59 pm

Why class is declared as final.

If all variables are private, even if class is extended , variables are not visible to in
derived class.

Also no need to declare variables are final, private is enough.

[Reply](#)**Lokesh Gupta**

November 25, 2015 at 3:26 pm

Making class final is necessary because one can create a mutable subclass and inject its reference to main immutable class. Application will be compromised in this way.

Next thing, even if variables are not visible in subclass, yet getter methods are. You can easily override the getter method, inject its reference in super class and blow up whole immutability based design.

Making variables final is optional but certainly highly recommended. You do not want to change the state of object in any way, after it's fully initialized. private prevent changing them in other classes, but final ensures that you will not be able to change even inside same class also.

[Reply](#)**Rajesh**

November 29, 2015 at 5:50 am

Dear Lokesh,

In first point here you said that "Making class final is necessary because one can create a mutable subclass and inject its reference to main immutable class. Application will be compromised in this way."

My point is if you have made constructor private in Parent class then how would you create an object of subclass. Can you please give an example.

Thanks in advance!!

[Reply](#)

Lokesh Gupta

November 29, 2015 at 8:03 am

Oops, I must be thinking something else at that time. You are right !! 😊

[Reply](#)

Nitin

July 19, 2017 at 9:53 pm

Hi,

I have doubt on this. If variables in class are final, and we mark all the methods also final, what's the need of marking class as final. Please tell.

[Reply](#)

shruti

October 17, 2015 at 10:43 am

Hi,

You can add in this is there is there is alist member in class
List list . How to make it immutable

Thanks,
Shruti

[Reply](#)

Lokesh Gupta

October 17, 2015 at 6:28 pm

Please your question more clear. I couldn't understand it 😞

[Reply](#)

Sanjay

February 3, 2016 at 12:05 pm

Actually @Shruti want to say that if a Class contain a List.
Then how you make it immutable ?

Ex :

```
public final class ImmutableClass  
{  
    Private final List c1;  
  
}
```

[Reply](#)

Lokesh Gupta

February 3, 2016 at 12:38 pm

So in this case, you must return the list instance wrapped in `Collections.unmodifiableList()` from it's getter method.

[Reply](#)

Ayush

[August 13, 2015 at 3:54 pm](#)

why new objects are returned with copied contents for mutable objects??

[Reply](#)

Rakesh sahu

[July 5, 2015 at 10:19 am](#)

Hi Lokesh,

How we will handle the situation when class is have a instance type of it own type.

for examle

```
public final class ImmutableClass  
{
```

```
private final ImmutableClass immutable;
```

[Reply](#)

Ravi

November 20, 2014 at 11:26 am

Hi Lokesh,

In TestMain.tryModification(...) this has local variables whose scope is within in this method, any fool will understand that you are modifying content of local parameters, how would your test case confirms you that you are trying to modify ImmutableClass instances???

[Reply](#)

Lokesh Gupta

November 20, 2014 at 6:08 pm

Hi Ravi, Could you please try some modifications and share with all of us your findings?? Let me know if something interesting you find.

[Reply](#)

Rahul Kumar Gupta

November 8, 2014 at 1:47 pm

Why is it we need a "Private Constructor" in an immutable class? What is the specific need of it?

[Reply](#)

Lokesh Gupta

November 11, 2014 at 6:36 am

Absolutely no need. As far as you are able to create a fully initialized instance of class, and does not let modify users to change it after, you are free to make it public as well.

[Reply](#)**Rahul Kumar Gupta**

November 11, 2014 at 6:42 am

let's say we have a constructor for immutable class, which takes in 2 parameters: String1 and String2. Now after calling the constructor from main method, we modify those strings that were initially passed in the constructor, then wouldn't it impact our immutable class object instance?

[Reply](#)**vinaykallat**

December 3, 2014 at 11:51 am

Nope String is itself immutable so changing string1 and string2 would not change the values in the Immutable class Instance variable say immutableString1 and immutableString2. Even more to say it would be pointing to a different location in the String pool.

That is : let us say:

string1 = "hi"; //lets us say in memmory where ultimately it would be stored

– 33450

string2 = "Vinay"; // lets us say in memmory where ultimately it would be stored – 33451

now let us say you pass this values to the constructor using new Immutable(string1, string2);

now lets say after assigning in the constructor:

immutableString1 and string1 point to 33450 location in pool with Value as hi
immutableString2 and string2 point to 33451 location in pool with Value as Vinay

Now if you change

string1 = "bye"; //would point to a new location in memmory 33452 as bye

string2 = "Rahul"; //would point to a new location in memmory 33453 as Rahul

The values in memmory 33450 and 33451 remains untouched because of which immutableString1 and immutableString2 would not be changed. That is why cloning was not necessary for Immutable instance variable but if you try the same with date without cloning you would run into problems.

So even I feel private constructor is not actually needed.

[Reply](#)

agpt

[August 4, 2014 at 6:22 am](#)

I don't get it...!! How can one assign value to final declared variable/object ?? for example. private final Integer int; it give error here itself !!

[Reply](#)

agpt

August 4, 2014 at 6:38 am

ohh sorry, I got it.. we are initializing it from constructor. Can you please explain about following scenario:

```
//..... Case 1....  
private final int num;  
public Constructor(int num)  
{  
    this.num =num;  // this is working  
}  
//.....Case 2...  
private final static int num;  
public Constructor(int num)  
{  
    this.num = num; // not working  
}
```

[Reply](#)**Lokesh Gupta**

August 4, 2014 at 7:05 am

static variables/blocks are initialized even before constructor is called. Also, static variables can be accesses only inside static blocks.

[Reply](#)**Siva Kumar**

October 30, 2014 at 12:37 pm

Hi,

The keyword 'this' refers to current instance. but 'static' is not specific to any instance, so that the below code never works.

```
private final static int num;  
public Constructor(int num)  
{  
    this.num = num; // not working  
}
```

[Reply](#)

rajat kumar

[June 20, 2015 at 9:03 am](#)

if you create a variable static final it means your variable is constant it means you can not change the value of your static final variable.

[Reply](#)

@ndee

[July 17, 2014 at 2:32 pm](#)

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

[Reply](#)

@ndee

July 17, 2014 at 2:38 pm

i meanthe correct thing should be final classes cannot be subclasses and final methods cannot be overridden in a subclass

[Reply](#)**Lokesh Gupta**

July 17, 2014 at 4:18 pm

OK. But how it make a class immutable? I can still change the state of class by changing the values of fields it contain.

[Reply](#)**Raveesh Verma**

September 10, 2014 at 1:47 pm

Hi Lokesh,

When you change the values of the fields the state does not change but, the new object of that class will be created and its reference points to the new object (while old object is still in memory).

Thanks,
Raveesh

[Reply](#)

Vinay Kallat

December 3, 2014 at 3:47 pm

Hi Raveesh,

I am not sure if I get your question. But instance variable define the state of the object. So when somebody says a class is immutable it means once defined nobody can change its state which means that nobody can change its instance variables.

So If an Raveesh object of class Employee is created then its instance variable name cannot be changed to Vinay.

Unless a new object with Vinay is created. So if multiple threads are accessing Raveesh object, no one can come in and suddenly change its state to Vinay shocking all the other threads.

Hope it is clear now.

udaykiranreddy desam

July 15, 2014 at 1:13 pm

Hi, in below code if i have 100 variables and getter and setter for this am i supposed to add them manually in getT() method is there any shortcut

```
package Immutable2;
```

```
import java.util.Date;
```

```
/**
```

```
 * Always remember that your instance variables will be either mutable or in  
 * Identify them and return new objects with copied content for all mutable  
 * 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
     * */
    private final Integer immutableField1;
    /**
     * String class is immutable as it also does not provide setter to change
     * */
    private final String immutableField2;
    /**
     * Date class is mutable as it provide setters to change various date/tim
     * */
    private final Date mutableField;
    private final Test t;

    //Default private constructor will ensure no unplanned construction of c
    private ImmutableClass(Integer fld1, String fld2, Date date, Test t)
    {
        this.immutableField1 = fld1;
        this.immutableField2 = fld2;
        Date d = new Date();
        d.setTime(date.getTime());
        //this.mutableField = new Date(date.getTime());
        this.mutableField = d;
        Test tt = new Test();
        tt.setName(t.getName());
        this.t = tt;
    }

    //Factory method to store object creation logic in single place
    public static ImmutableClass createNewInstance(Integer fld1, String fld2
    {
        return new ImmutableClass(fld1, fld2, date, t);
    }

    //Provide no setter methods
    /**
     * Integer class is immutable so we can return the instance variable as i
     * */
    public Integer getImmutableField1()
    {
        return immutableField1;
    }

    /**
     * String class is also immutable so we can return the instance variable
     * */
    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());
    }

    public Test getT()
    {
        Test t1 = new Test();
        t1.setName(t.getName());
        return t1;
    }
    // return new Test();
}

@Override
public String toString()
{
    return immutableField1 + " - " + immutableField2 + " - ";
}

public static void main(String[] args)
{
    Test t = new Test();
    t.setName("kiran");
    Date d = new Date();
    d.setTime(1000000000);
    ImmutableClass im = ImmutableClass.createNewInstance(100, "test");
    System.out.println(im);
    tryModification(im.getImmutableField1(), im.getImmutableField2(), im);
    System.out.println(im.getT().getName());
    System.out.println(im);
}

private static void tryModification(Integer immutableField1, String immutableField2)
{
    immutableField1 = 10000;
    immutableField2 = "test changed";
    mutableField.setDate(10);
    t.setName("uday");
}

}

class Test
{
    private String name;

```

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

[Reply](#)

Lokesh Gupta

July 15, 2014 at 1:35 pm

You can use [BeanUtils.copyProperties\(\)](#) method

[Reply](#)

Deepak

January 28, 2019 at 6:24 pm

Can you post the code for this example without using
BeanUtils.copyProperties

BeanUtils.copyProperties is a 3rd party lib

[Reply](#)

Deepak

January 28, 2019 at 10:26 pm

Post the working example

[Reply](#)**barun**

June 26, 2014 at 8:32 am

Hi Lokesh,

Could you please explain me in brief? why the below code is not showing immutable behaviour as you explain above....

infact i have modified your code and included one mutable class Test with an attribute name with setter and getter method for name...I did it in the same way as you did for mutable class Date but output you can verify below.

```
package com.barun;

import java.util.Date;

/**
 * Always remember that your instance variables will be either mutable or im
 * Identify them and return new objects with copied content for all mutable
 * 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
     */
    private final Integer immutableField1;
    /**
     * String class is immutable as it also does not provide setter to change
     */
}
```

```
private final String immutableField2;
/**
 * Date class is mutable as it provide setters to change various date/tir
 * */
private final Date mutableField;
private final Test t;

//Default private constructor will ensure no unplanned construction of c
private ImmutableClass(Integer fld1, String fld2, Date date, Test t)
{
    this.immutableField1 = fld1;
    this.immutableField2 = fld2;
    Date d = new Date();
    d.setTime(date.getTime());
    //this.mutableField = new Date(date.getTime());
    this.mutableField = d;
    Test tt = new Test();
    tt.setName(t.getName());
    this.t = tt;
}

//Factory method to store object creation logic in single place
public static ImmutableClass createNewInstance(Integer fld1, String fld2
{
    return new ImmutableClass(fld1, fld2, date, t);
}

//Provide no setter methods
/**
 * Integer class is immutable so we can return the instance variable as i
 * */
public Integer getImmutableField1()
{
    return immutableField1;
}

/**
 * String class is also immutable so we can return the instance variable
 * */
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 return
 * */
public Date getMutableField()
{

```

```
        return new Date(mutableField.getTime());
    }

    public Test getT()
    {
        return t;
    }

    @Override
    public String toString()
    {
        return immutableField1 + " - " + immutableField2 + " - " + mutableFie
    }

    public static void main(String[] args)
    {
        Test t = new Test();
        t.setName("barun");
        Date d = new Date();
        d.setTime(1000000000);
        ImmutableClass im = ImmutableClass.createNewInstance(100, "test", d,
        System.out.println(im);
        tryModification(im.getImmutableField1(), im.getImmutableField2(), im.
        System.out.println(im);
    }

    private static void tryModification(Integer immutableField1, String immu
    {
        immutableField1 = 10000;
        immutableField2 = "test changed";
        mutableField.setDate(10);
        t.setName("tarun");
    }
}

class Test
{
    private String name;

    public String getName()
    {
        return name;
    }

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

[Reply](#)**Lokesh Gupta**

June 26, 2014 at 9:27 am

For other's information, the output of above code is:

```
100 - test - Fri Jan 02 09:16:40 IST 1970-barun
```

```
100 - test - Fri Jan 02 09:16:40 IST 1970-tarun
```

Barun, as you see the class is not immutable as you are able to change the name from "barun" to "tarun". To understand this, you must know the concept that **Java is pass by value** [please read this post]. When you pass reference of t to method tryModification(), then there are two reference variables in program referring to same instance of Test. You can say e.g. t1 = t2 = new Test(); This also has two reference variable t1 and t2; both pointing to single instance of Test.

If you change anything in instance of Test by using reference of t1, then t2 will also be able to see it. Both referring to same instance, remember. Now what if you make t2 'null' or assign any new instance of Test(). Will t1 impact?? NO. Assigning null or another instance make t2 to point somewhere else; but t1 is still pointing to original instance.

```
package test.core;

public final class ImmutableClass
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.setName("&&&&&quot;barun&&&&&quot;");
        Test t2 = t1;

        //Change the name
        t2.setName("&&&&&quot;tarun&&&&&quot;");
        System.out.println(t1.getName());
    }
}
```

```
//Mak2 t2 null
t2 = null;
System.out.println(t1.getName());

//Mak2 t2 point to another instance
t2 = new Test();
t2.setName("&&&&&quot;lokesh&&&&&quot;");
System.out.println(t1.getName());
}
}

class Test
{
    private String name;

    public String getName()
    {
        return name;
    }

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

Output:

```
tarun
tarun
tarun
```

You see why name value got changed, because reference of Test() got skipped outside ImmutableClass. Once you have reference of an object, you can always change its state if it's mutable. Mutability of Test class, make ImmutableClass mutable as well.

To prevent this, you need to return a new instance of Test from getT(). This will ensure that original reference of Test() does not skip outside the ImmutableClass, so that nobody can change it.

Hope it helps.

[Reply](#)

Rahul

June 24, 2014 at 7:15 am

Sir, I am a bit confused and would like to ask that whatever we are changing in the tryModification method, we are changing it locally to this method..then anyway its not going to reflect the changes in the class...so if I take any int variable in the class without keeping it final then also it won't change as that int variable is changed locally....

Following is your code with little modification

package com.immutable.test;

```
import java.util.Date;

public final class ImmutableClass {

    private final String name;
    private final Integer num;
    private final Date date;
    private int i;

    public ImmutableClass(String name,Integer num, Date date,int i){
        this.name = name;
        this.num = num;
        this.date = new Date(date.getTime());
        this.i = i;
    }

    public int getI(){
        return i;
    }

    public String getName(){
        return name;
    }

    public Integer getNum(){
        return num;
    }

    public Date getDate() {
```

```
        return new Date(date.getTime());
    }

    public String toString(){
        return name+" "+num+" "+date+" and int number is-> "+i;
    }
}

package com.immutable.test;

import java.util.Date;

public class TestMain {

    public static void main(String[] args) {
        ImmutableClass immutableClass = new ImmutableClass("Rahul", 1012516, new Date());
        System.out.println(immutableClass);
        modify(immutableClass.getName(), immutableClass.getNum(), immutableClass.getDate(), 21);
        System.out.println(immutableClass);
    }

    @SuppressWarnings("deprecation")
    private static void modify(String name, Integer num, Date date, int i){
        name = "Rocky";
        num= 1012517;
        date.setDate(101);
        i = 11;
    }
}
```

Output : Rahul 1012516 Tue Jun 24 12:45:18 IST 2014 and int number is-> 21
Rahul 1012516 Tue Jun 24 12:45:18 IST 2014 and int number is-> 21

Sir, please clarify

[Reply](#)

Lokesh Gupta

June 24, 2014 at 7:38 am

Hi Rahul, Thanks for asking this question. Asking a question to clear doubt is always better, then remaining in doubt.

When you talk about immutability, always separate fields in class into two sections: i.e. mutable fields and immutable fields.

All primitives (e.g. int, long), wrapper classes (e.g. Integer, Double) and String class are implicitly immutable in java so you do not need to much worry about it.

Date class is mutable field so you need to safeguard it with some extra code e.g. we wrote getDate like this:

```
public Date getDate()  
{  
    return new Date(date.getTime());  
}
```

What we are actually returning a new copy of Date object, so if somebody manipulate it then original copy is unchanged.

Date class provides you setXXX() methods which allow changing its content. Primitives and Wrapper classes do not provide any set methods, so you can't change anything inside them.

You are right that methods receive a local copy to work on, but that copy is actually a copy of reference pointing to original object. You can not change the reference, but once you have the reference, you can always change the content inside object that reference is pointing to.

[Reply](#)

Rahul

[June 24, 2014 at 9:12 am](#)

Thanks for a quick response sir.... And thats what my point is if we get the reference then we can change the content but see the following code...

```
package com.immutable.test;
```

```
public class Test1 {
```

```
    private int num;
```

```
    public Test1(int num) {  
        this.num = num;  
    }
```

```
    public int getNum(){  
        return num;  
    }
```

```
    public String toString(){  
        return num+"";  
    }  
}
```

```
package com.immutable.test;
```

```
public class Test2 {
```

```
    public static void main(String[] args){  
        Test1 test1 = new Test1(5);  
        System.out.println(test1);  
        modify(test1.getNum());  
        System.out.println(test1);  
    }
```

```
    public static void modify(int num){  
        num = 15;
```

```
}
```

```
}
```

Output : 5

5

As here also I'm passing the variable num by getting it through Test1 class's instance and changing the value in modify method..but its not reflecting...

I'm confused here why so..please explain me and so sorry for bothering you again

Thanks!

[Reply](#)

Rahul

[June 24, 2014 at 9:47 am](#)

And one more thing Sir, java supports call by value and not call by reference..so in modify method whatever we are passing we are just passing the value and not the variable itself..so any change in the method for that variable is not reflecting there....Please clarify my doubts

[Reply](#)

Rahul

[June 24, 2014 at 10:10 am](#)

Sir, I misunderstood one of your point, so am clarify with what you were telling as int is just a primitive data type, its neither mutable or immutable rather it depends upon our implementation...

Thanks for your patience and I personally like your blog as it clarifies many things..Thanks Once again!!!

[Reply](#)

Lokesh Gupta

June 24, 2014 at 10:20 am

Your welcome. In between, I would like you to go through this post, as it will help you in clearly defining the term "call by value" in context of java, only in case if you have any doubt.

<https://howtodoinjava.com/java/basics/java-is-pass-by-value-lets-see-how/>

[Reply](#)

Sanket Bhalerao

June 17, 2014 at 10:22 am

below is a simple code that i wrote taking inspiration from your code obviously, now can you explain how the code is you posted is immutable and the code i posted is mutable

////////////////////////////////

```
import java.util.Date;

public class ImmutableDemo
{
    Date date;
    String string;
```

```
public ImmutableDemo(Date date,String string)
{
    this.date = date;
    this.string = string;
}

public static void modify(Date dateNew,String stringNew)
{
    dateNew = new Date(dateNew.getTime() + 86400000);
    stringNew = "SOMETHING";
}
@Override
public String toString()
{
    return date + " %%% " + string;
}

public static void main(String[] args)
{
    ImmutableDemo id = new ImmutableDemo(new Date(), "NOT SOMETHING");
    System.out.println(id);
    modify(id.date, id.string);
    System.out.println(id);
}
}
```

//////////

Output: (content is unchanged)

Tue Jun 17 15:51:21 IST 2014 %%% NOT SOMETHING

Tue Jun 17 15:51:21 IST 2014 %%% NOT SOMETHING

[Reply](#)

Lokesh Gupta

June 17, 2014 at 11:01 am

I changed your modify method to below:

```
public static void modify(Date dateNew,String stringNew)
{
    dateNew.setYear(2009); //Updated the year
```

```
stringNew = "SOMETHING";  
}
```

Now output is:

Tue Jun 17 16:30:30 IST 2014 %%% NOT SOMETHING

Thu Jun 17 16:30:30 IST 3909 %%% NOT SOMETHING

[Reply](#)

Deepak Kumar

June 11, 2014 at 6:23 am

Hello Lokesh,

This explanation is somewhat full of confusion. In the Class you have used 3 variables, out of them two are String and Integer which you are trying to modify (to show that these cannot be changed as the class is immutable). But these two variables are itself immutable so these cannot be changed whether the class is mutable or immutable.

[Reply](#)

Lokesh Gupta

June 11, 2014 at 7:27 am

3 variables are: String, Integer and Date. Date is mutable.

[Reply](#)

Sowmi

June 10, 2014 at 12:41 pm

@Lokesh: Can you clarify a point ?

What is exactly the need of having a static method with the name `createNewInstance` ? We can directly call the constructor when we create a object with new operator.

To be more precise what will be the difference between creating an object directly calling the constructor (making it public) and having a method to create an instance of the immutable class ?

According to me an immutable class is a class for which if an instance is created , there should be no means/method associated with the class which can change the state of the object. For eg String class has no method which returns void and changes the String. All functions of String class will always return a new String

Similar to that , I suppose having an instance creation method is not needed for making a class Immutable

Please correct me if I'm wrong anywhere

[Reply](#)**Lokesh Gupta**

June 10, 2014 at 12:56 pm

Yes, you are right. The method `createNewInstance()` is not mandatory and completely optional "ONLY IF" the constructor return you a fully prepared object. This method has absolutely nothing to do with immutability of class. As

you can notice that I put a comment above method "Factory method to store object creation logic in single place". So essentially it was an attempt to have object creation logic in one place. The same thing you can achieve with having multiple overloaded constructors as well.

[Reply](#)

Piyush Vj

[April 11, 2014 at 7:24 am](#)

Above example goes good !!

but for modification you are using tryModification mehod, here the modification limits up to this method only due to local variable scope.

[Reply](#)

venkata

[March 13, 2014 at 3:00 am](#)

HI Lokesh,

Your blogs are very useful which clears all my doubts. But seeing about code, had a confusion on difference between Singleton and Immutable class?

[Reply](#)

Lokesh Gupta

[March 13, 2014 at 9:14 am](#)

Singleton means only one instance per JVM. There is no restriction that you cannot change the state of instance.

Immutable means you cannot change the state of instance once created. Though you can have multiple instances with same state data.

[Reply](#)

asheesh

[May 27, 2014 at 4:20 pm](#)

In case of singleton,if there are more then one server/cluster in production environment for an application,usually it happens,so in that case there would be multiple singleton class instance across all JVMs.

In this way there would be more than one instance per application?

[Reply](#)

Lokesh Gupta

[May 27, 2014 at 4:23 pm](#)

Singleton ensures one instance per JVM only, not per application. Yes there will be as many instances as many JVMs. That's why it is advised to use immutable classes as singleton which will have same data in all JVM nodes.

[Reply](#)

Pallav Rajput

January 20, 2014 at 9:19 am

Hi Lokesh,

According to your example, the output (content will be unchanged), if our class is immutable.

but the line

`tryModification(im.getImmutableField1(),im.getImmutableField2(),im.getMutableField());` is not going to change the content of instance `im` in all cases, even our class and attribute is final or not. we are just passing the value via parameter in `tryModification` method, we are not changing the values of `im` in any ways.

correct me if i am wrong.

Thanks,

pallav

[Reply](#)

Lokesh Gupta

January 20, 2014 at 5:31 pm

Inside method I am trying to change state in `"mutableField.setDate(10);"`

[Reply](#)

Java StringBuilder

November 30, 2013 at 1:49 am

Simple post for java but nicely written.

[Reply](#)

Ali

[November 27, 2013 at 7:38 am](#)

Would be beneficial if you elaborate each benefit mentioned in your post.

[Reply](#)

varun

[October 25, 2013 at 11:30 pm](#)

You have not talked about map immutable and also reflection effect

[Reply](#)

Lokesh Gupta

[October 25, 2013 at 11:38 pm](#)

Would you like to add something here? I will appreciate if you do so.

[Reply](#)

Madhuri

October 23, 2013 at 9:02 pm

Hi Lokesh,

Your articles are really good and helpful. I want to understand the real advantage of making the constructor private and providing the factory method to create the instance while making a class Immutable. In your example, if you make the constructor as public and remove the factory method, how is it going to affect the immutability of this class. Can you please explain it in detail?

Thanks

[Reply](#)

Lokesh Gupta

October 23, 2013 at 11:05 pm

There is no absolute requirement for making constructor private. We have public constructor in String class and it's good example of immutability. Factory method gives a meaningful name to the process of object creation, specially when class has multiple constructors with different number of parameters. In above example also, there will not be any behavioral change. It's about only good practice.

[Reply](#)

Colton Brooks

October 15, 2013 at 4:38 pm

I respectfully disagree.

You point out at the very beginning of your article: "An immutable class is one whose state can not be changed once created." That means if I instantiate an object (im) of type ImmutableClass with initial values of 100, "test", and "2013-10-14", those values should NEVER, EVER change.

Well, guess what? If I can manipulate an "old reference" to an object (the Date d) stored INSIDE the object (im), then I've not only changed d, I've changed the object im as well! You **do not** want to allow somebody to be able to do that! That is precisely the whole point of immutable classes- to prevent this kind of chicanery!

In your original article, you included a link to Java documentation. From that link:

"Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies."

Unfortunately, that is **exactly** what your example does! It stores a reference (in mutableField) to an external, mutable object (Date d) passed to the constructor!

The good news is that there is an easy solution. What you need to do is create a defensive copy in your constructor:

```
this.mutableField = new Date(date.getTime());
```

Then, I can manipulate the original Date (d) that I pass in to the constructor all I want and it won't affect my instance of ImmutableClass at all, since the internal date in im is now a completely different Date object (that just happens to be set to the same time as d initially was).

[Reply](#)

Lokesh Gupta

October 15, 2013 at 10:49 pm

Perfect. I see what you meant in previous comment. Agree. I will update the example with defensive copy so that others can also refer to same technique. Thanks for your contribution.

[Reply](#)

Deepak

April 30, 2022 at 11:24 am

Code please to explain this

[Reply](#)

Colton Brooks

October 14, 2013 at 5:54 pm

I think there might be a problem in your example, as you do not make a defensive copy of the Date in the constructor.

For example, try this for your test program:

```
Date d = new Date();
ImmutableClass im = ImmutableClass.createNewInstance(100,"test", d);
System.out.println(im);
```

```
d.setMonth(1);  
System.out.println(im); // date is changed.
```

[Reply](#)**Lokesh Gupta**[October 14, 2013 at 10:06 pm](#)

Code you tried is flawed. You are not getting the date object from Immutable class, you are using an old reference which you passed to immutable class. Try asking the date object from immutable class and then change it.

[Reply](#)**Muralidhar**[September 26, 2013 at 11:47 pm](#)

Very Good points you mentioned above..Thanks..

[Reply](#)**naman kapoor**[September 21, 2013 at 12:55 am](#)

I think this example is WRONG...

Immutability means you CAN CHANGE the content, but if you change then a new copy of the object will be created in memory.

Ex.

```
String a="hi";
```

```
a="hello";
```

```
System.out.println(a);
```

O/P

hello

that is changes are made to the a object

But if we go by your example, then we can never change the value

Hence according to your version of immutability, value of 'a' could never be changed from hi to hello

[Reply](#)

Lokesh Gupta

September 21, 2013 at 11:35 am

Hey Naman, That's something new definition to me but I disagree with it. Creating a new copy of object is part of application logic how you want to handle such calls to make program more readable and more flexible, just like design patterns.

And forget about programming, search it's dictionary meaning, it also will make your definition weak.

[Reply](#)

Bibhu

October 3, 2013 at 11:26 am

Hi naman,

Here you are not changing the content of object instead making a new object and referencing to the older one. The first string object "hi" will be unused and cleared by the garbage collector latter.

[Reply](#)

Vijayakumar Ramdoss

June 6, 2013 at 7:41 pm

Excellent example

[Reply](#)

Stefan Richter

October 30, 2012 at 10:31 am

Just remove "= null" in line 13, 17 and 21. Then you are able to make your fields final and set them in your constructor. Setting field "= null" the way you do it, does not have any additional effect, because the virtual machine sets all fields to thier defaults on creation of instances (Objects = null, numbers = 0, boolean = false) before any constructor or initialisation block is executed.

[Reply](#)

Admin

October 30, 2012 at 10:37 am

I was really unaware of this. My bad. Made the corrections you suggested. In between, thanks for valuable contribution in this post.

Thanks !!

[Reply](#)

Stefan Richter

October 30, 2012 at 9:35 am

Your article is good until you try to explain immutable classes with code examples.

First: your class does not follow your own rules, because mutableField1 and mutableField2 are not final.

Second: even if you would return the original field instead of a copy of the field (lines 21 and 26) your test would run with same output, because line 8 and 11 in the test alter the value of the local variable and not the value of the fields.

Besides both fields are immutable classes and can not be altered. So you can't even alter the internal value of the fields.

[Reply](#)

Admin

October 30, 2012 at 10:19 am

Hi Stefan,

I appreciate your patience to figure out the mistake and responded with reasons. I agree that previous example was not correct on second point. Which i have updated now. Please review it.

First point: fields are not final because i want to set them in constructor which it commonly seen in all JDK supplied immutable classes. Making them final will give compile time error.

Thanks

[Reply](#)

abhi

[June 24, 2013 at 12:39 am](#)

Date is getting change as per above example,if class and variable is immutable(it means we can not change)

BUT.....Date is get change....

THIS EXAMPLE IS TOTALLY WRONG...

[Reply](#)

Lokesh Gupta

[June 24, 2013 at 9:44 am](#)

Probably I am not able to see what you want me to see. I am sorry. Can you please be more specific what is that you are pointing at. I can't see why the

date has been changed?

[Reply](#)

Gaurav

[October 13, 2013 at 9:09 am](#)

Hi Lokesh, I have little problem to understand the conversation between you and Stefan. According to Stefan- "First: your class does not follow your own rules, because mutableField1 and mutableField2 are not final."

And in your reply you said like "First point: fields are not final because i want to set them in constructor which it commonly seen in all JDK supplied immutable classes. Making them final will give compile time error."

But whatever I have seen above, I have not find any non-final field. So it will be very kind of your side if you elaborate this more. Please!!!

Thankyou

Lokesh Gupta

[October 13, 2013 at 11:12 am](#)

Gaurav, I tried hard to remember and it is what I can recall so far. Previously field declaration was like :

```
private Integer immutableField1 = null; //Initialized with null
```

Putting final in above statement was causing error while setting value because of obvious reasons. [You cannot change value once initialized for final variables]. So what I did later, I declared the variables like:

```
private final Integer immutableField1;
```

Now I am able to initialize anywhere I want, but obviously before object is fully constructed.

Eric Jablow (@DrJayVA)

October 29, 2012 at 2:10 pm

Also taken from "Effective Java,"

An immutable class copies mutable data, both on input and on output. You see this most often with `java.util.Date`. Imagine a `DateRange` class. Its constructor should look like:

```
public DateRange(final Date start, final Date end) {  
    this.start = new Date(start.getTime());  
    this.end = new Date(end.getTime());  
    if (!this.start.before(this.end)) {  
        throw new DateRangeException(this.start, this.end); //Write this, or use an IAE  
        instead.  
    }  
}
```

An immutable class either is not `Serializable`, or it uses `readResolve` to maintain class invariants.

```
public Object readResolve() {  
    return new DateRange(start, end);  
}
```

This way, no one can falsify a serialized stream.

[Reply](#)

Admin


October 29, 2012 at 6:57 pm

Thanks for pointing out serialization issue. Yes, i agree. A class should maintain its state even in case of serialization/ de-serialization. In between, in my example, i have made class serializable . So, if it had been, i would have added readResolve() like this;

```
//pseudo code  
readResolve()  
{  
    return this;  
}
```

[Reply](#)

Leave a Comment



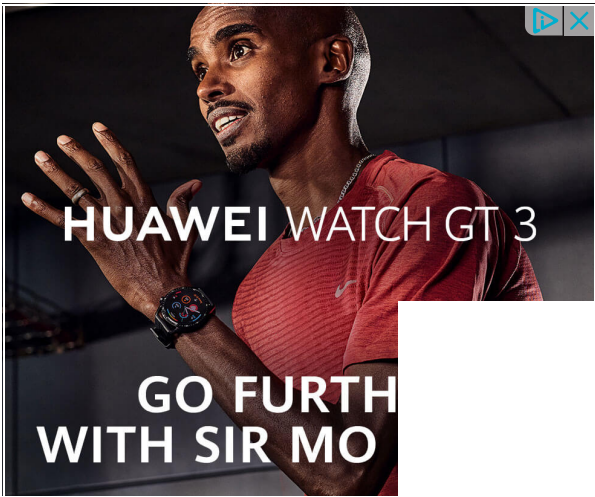
Name *

Email *

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment





A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- [About Me](#)
- [Contact Us](#)
- [Privacy policy](#)
- [Advertise](#)
- [Guest Posts](#)

Blogs

[REST API Tutorial](#)



Copyright © 2022 · Hosted on [Cloudways](#) · [Sitemap](#)