How to Create Immutable Class in java?

PANKAJ — 76 COMMENTS

Today we will learn how to create an immutable class in Java. **Immutable objects** are instances whose state doesn't change after it has been initialized. For example, String is an immutable class and once instantiated its value never changes.

Read: Why String in immutable in Java

Immutable Class in Java

An immutable class is good for caching purpose because you don't need to worry about the value changes. Other

➕ Advanced Java Concepts

Recommended Tutorials

➕ Java Tutorials

➕ Java EE Tutorials

benefit of immutable class is that it is inherently **thread-safe**, so you don't need to worry about thread safety in case of multi-threaded environment.

Read: Java Thread Tutorial and Java Multi-Threading Interview Questions.

Here I am providing a way to create an immutable class in Java via an example for better understanding.

To create an immutable class in java, you have to do following steps.

1. Declare the class as final so it can't be extended.
2. Make all fields private so that direct access is not allowed.
3. Don't provide setter methods for variables
4. Make all **mutable fields final** so that it's value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

To understand points 4 and 5, let's run the sample Final class that works well and values don't get altered after instantiation.

FinalClassExample.java

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

```
                HashMap<String, String> h1 =
new HashMap<String,String>();
                h1.put("1", "first");
                h1.put("2", "second");

                String s = "original";

                int i=10;

                FinalClassExample ce = new
FinalClassExample(i,s,h1);

                //Lets see whether its copy
by field or reference

System.out.println(s==ce.getName());
                System.out.println(h1 ==
ce.getTestMap());
                //print the ce values
                System.out.println("ce
```

Output of the above immutable class in java example program is:

```
Performing Deep Copy for Object initialization
true
false
ce id:10
ce name:original
ce testMap:{2=second, 1=first}
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:
{2=second, 1=first}
ce testMap after changing variable from
accessor methods:{2=second, 1=first}
```

Now let's comment the constructor providing deep copy and uncomment the constructor providing a shallow copy. Also uncomment the return statement in `getTestMap()` method that returns the actual object reference and then execute the program once again.

```
Performing Shallow Copy for Object
initialization
true
```

```
true
ce id:10
ce name:original
ce testMap:{2=second, 1=first}
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:
{3=third, 2=second, 1=first}
ce testMap after changing variable from
accessor methods:{3=third, 2=second, 1=first,
4=new}
```

As you can see from the output, HashMap values got changed because of shallow copy in the constructor and providing a direct reference to the original object in the getter function.

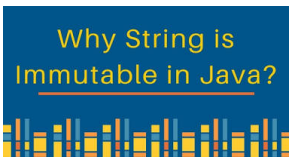
That's all for how to create an immutable class in java. If I have missed something here, feel free to comment.

Further Reading: If the immutable class has a lot of attributes and some of them are optional, we can use builder pattern to create immutable classes.

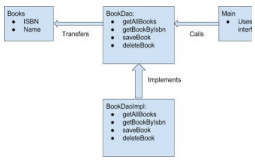
> You can check out more Java examples from our GitHub Repository.

RESTful Web Services Interview Questions

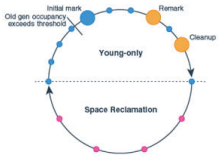
Java Tricky Interview Questions



Why String is Immutable in Java

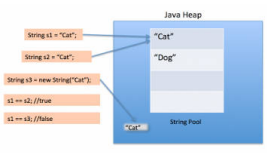


DAO Design Patterr



Garbage Collection in Java

Java 11 Features



What is Java String Pool

Java Mode Mana

« PREVIOUS
ConcurrentHashMap in Java

NEXT »
StringBuffer vs StringBuilder

About Pankaj

I love Open Source technologies and writing about my experience about them is my passion. You can connect with me directly on **Facebook**, **Twitter**, and **YouTube**.

FILED UNDER: JAVA

Comments

Sreerag says

May 18, 2019 at 11:16 pm

But when we trying to change a String object a new object would be created with those changes. Is that possible here ?. Please explain

Reply

Ashwani Pratap says

April 12, 2019 at 9:23 am

What's the significance of declaring class as final here?

Reply

Pankaj says

April 14, 2019 at 4:21 am

So that it can't be extended.

Reply

Ravi Beli says
March 11, 2019 at 6:22 pm
Even within this constructor if you assign cloned version of input “hm” to testMap, it should still work, no need of deep copy.
public FinalClassExample(int i, String n, HashMap hm)
{
System.out.println(“Performing Shallow Copy for Object initialization”);
this.id=i;
this.name=n;
this.testMap=hm.clone();
}
Reply

Satish says
March 18, 2019 at 2:15 pm
Perfect answer with a small change. You need to type cast when calling the clone method as it return object.
Reply

Pandidurai says
January 31, 2019 at 4:26 am
Hi,
First i want to thank you for sharing your knowledge with us.
here i have one question with this
if i am getting HashMap reference through getter method then i can avoid changing Immutable class value by sending deep copy. But what if user accessed it directly? i mean
Map local = ce.testMap;
local.put(“local” , “local”);
in this case ce.testMap will have the above added object as well.
How can we avoid this?
thanks!
Reply

Rishabh says
February 11, 2019 at 6:10 am
while returning map return new Object.
So the getter method of testMap will look like
getTestMap{
return new HashMap(testMap);
}
Reply

vkas says
February 21, 2019 at 3:05 am
Hi Pandidurai , testMap access is private as rule, so you can not access outside class directly, without getter you can not access directly
Reply

online java training says
January 3, 2019 at 1:15 am
It was worth a read about immutable class in Java! Really learned a lot form this article. I can imagine the effort you put into this & especially appreciate you sharing it. keep up the good work.
Reply

Deepak says
December 18, 2018 at 9:12 am
Hi,
please post your Java code of connection pooling in java without using Spring and hibernate.
Reply

Parvise says
October 11, 2018 at 11:02 pm
Not required map iteration in constructor and return statement clone which you used, may be any other

reason please provide that.

```
class Student {
private final int sid;
private final String sname;
private final Map map;
public Student(int sid, String sname, Map map) {
super();
this.sid = sid;
this.sname = sname;
this.map = new HashMap(map);
//this.list=list;
}
public int getSid() {
return sid;
}
public String getSname() {
return sname;
}
public Map getList() {
return map;
}
@Override
public String toString() {
return “Student [sid=” + sid + “, sname=” + sname + “]”;
}
}
```

Reply

Pankaj says

October 11, 2018 at 11:09 pm

Yes, we can use the HashMap constructor too. However, the idea here was to showcase how to perform the deep copy. Not every object constructor provides this feature, in that case, we need to perform field by field copy so that the original object and the cloned object have different references.

Reply

u.mang says

September 24, 2018 at 7:18 pm

This is how we create an immutable class. But can you please describe how the immutable object is created?

or is that happens in java or not??
Reply

Pankaj says
September 24, 2018 at 8:56 pm
A class is a design and Object is the actual implementation. When we say immutable class, it means immutable objects too.
Reply

Vijay Kumar says
September 22, 2018 at 3:29 am
Article is very nice and easy to understand. I have gone through the comments and got deeper understanding of this concept. Requesting everyone else to go through the discussions done in comments then you will get more understanding about this topic.
Thanks to Pankaj.
Reply

Omkar J says
August 28, 2018 at 2:24 am
Great Article…… very helpful in cracking interviews ….
Thank you so much !!
Reply

Jitendra Singh says
June 24, 2018 at 10:00 am
Hi Pankaj,
Thanks for writing such an informative article.
I would like to know what if my all member variables are immutable like all Strings or Wrappers?, Do I still need to follow above steps?
Thanks,
Reply

Pankaj says

June 24, 2018 at 10:42 am

Yes, most of them. You can get rid of deep-copy logic if the variables are immutable.

Reply

Prahlad Kumar says

May 19, 2018 at 4:19 am

Hi Pankaj,

What will happen if we do not declare class as final, since member variables are privare so can not be extendable.

I am here trying to understand why we need final for class and its member variable.

Is there any way state of a class can be modified if we don't declare class and its variables final?

Thanks,

Prahlad.

Reply

Gunjal Shrivastava says

August 16, 2018 at 4:44 am

You need the class as final so, that it cannot be extended further and it's implememtation should not be changed by overriding the methods and by creating new variables.

Making variables as final will make sure two things,

1. Compiler will flag error if any reassignment will take place

2. It will give readability to the code to other people who are working on the same class that these variables are not supposed to be changed.

Basically it is because of IMHO

Reply

Hagos haile says

April 27, 2018 at 8:01 am

correct output is:

Performing Shallow Copy for Object initialization
true
false
ce id:10
ce name:original
ce testMap:{1=first, 2=second}
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:{1=first, 2=second, 3=third}
ce testMap after changing variable from accessor methods:{1=first, 2=second, 3=third}
Reply

Vigilante says
May 31, 2019 at 2:55 am
Yeah Thats what i was thinking
.ce testMap after changing variable from accessor methods:{1=first, 2=second, 3=third}
4=new wont be get added
Reply

TEJENDRA PRATAP SINGH says
January 29, 2018 at 11:11 am
Your getter method for HashMap can be in that way
public HashMap getTestMap() {
//return testMap;
//HashMap tempMap=new HashMap();
return new HashMap(testMap);
}
Reply

Vaibhav Jetly says
September 23, 2017 at 2:03 am
What if we remove the final access modifier from class as we are independently handling all the fields or methods of this class. And if some one extend this class then they doesn't impact this class instance. Please suggest.
Reply

Mike New says
November 23, 2018 at 6:22 am
But then it is circumventing the intent of the class. If you want additional behavior from a different class, create a different class and have the immutable one as an instance member of that class.
Reply

Sameera says
September 12, 2017 at 9:32 pm
I also wrote an article with a complete different view and you may have a look,
http://www.codedjava.com/2017/09/immutable-objects-in-java_50.html
Thanks,
Reply

kuldeep patil says
July 6, 2017 at 12:09 am
I am not clear on Point 5 initialization
swallow or Deep are comparison not initialization.
swallow comparison is done by == and Deep comparison by equal / equalignorecase
Did you mean initialization should be done at constructor with safer way without exposing identity of the fields?
Reply

Rushabh says
June 24, 2017 at 10:47 pm
Correct output is::
Performing Shallow Copy for Object initialization
true
false
ce id:10
ce name:original
ce testMap:{2=second, 1=first}

ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:{3=third, 2=second, 1=first}
ce testMap after changing variable from accessor methods:{3=third, 2=second, 1=f
irst}
Reply

Tanmai says
May 13, 2017 at 8:05 pm
Which Java version ? or forgot the Final variables initialisation?
Reply

Manoj Kumar Vohra says
May 22, 2017 at 8:00 am
Final variables can be left uninitialized in declaration if initialization is provided by constructor.
Reply

Anbu says
April 13, 2017 at 12:28 am
Hi setter method for map provides shallow copy only though clone method so that we can change the value later. How come you can say its immutable ?
Reply

Prakash says
November 30, 2017 at 3:14 am
I believe, you are talking about getter method.
public HashMap getTestMap() {
//return testMap;
return (HashMap) testMap.clone();
}
I believe, It should return a copy by deep cloning.
Reply

Prakash says
November 30, 2017 at 3:17 am
I believe, In this example, Shallow copy would also work fine as long as we are storing immutable String object in HashMap.
In case, if we need to store mutable object in HashMap. We should do deep cloning
Reply

Pratyush says
April 6, 2017 at 6:27 am
I it necessary to have variable as final??
We can achive it without it also,
there is no statement to change varable.
Reply

Rahul says
April 1, 2017 at 11:18 am
When you make the field final, Why making the variable private is mandate?
Reply

tabish says
March 1, 2017 at 3:53 am
hi pankaj,
i love to read your blog. here i found a hack at main() using a reflection how to prevent it.
Class mc = ce.getClass();
Field fs = mc.getDeclaredField(“name”);
fs.setAccessible(true);
fs.set(ce, “hacked”);
System.out.println(“ce name after reflection hack:”+ce.getName());
Reply

shashank says

March 22, 2017 at 7:39 am
You cannot do this since name is final
Reply

Umang Gupta says
August 13, 2018 at 12:55 am
It does work.
Reply

WAA says
January 12, 2017 at 9:01 pm
Hi Pankaj
I didn't Understood the System.out.println(h1 == ce.getTestMap()) answer is False.
Can you Please explain why it is false.
Reply

Ankush K says
April 20, 2017 at 12:23 am
On this statement we are just checking the reference of h1 is pointing to the one that of our final class TestMap reference, which in this case is no, because we have made a new copy of h1 hashmap and copied it in TempHashMap which is an completely new Object, & then the reference of this temp map is assign to TestHashMap.
Hence this reference are pointing to, two different Object all together.
Reply

Bektur Toktosunov says
December 28, 2015 at 10:44 am
Thanks for great article!
Can we use testMap.clone() in the constructor instead of going through all of the map items with Iterator?
Reply

Vineet kaushik says
December 22, 2015 at 12:00 am
Very easy to understand and useful post!!
Reply

Vijay Nandwana says
December 18, 2015 at 3:02 am
Thank you Pankaj. I'm a big fan of your writing skills. You cover every details and explain concepts in easy to understand language. Thanks again.
Reply

Ajaz says
September 14, 2015 at 12:07 pm
private final String name; …Why this is final …String is already final …do we need to declare it again..???
Reply

Vinay says
February 5, 2017 at 10:40 pm
"String is already final." – yes , it is , but that final is at class level which means you can't extend the String class, while that "private final String " is for variable , so that we cant change the value of that object once initialized.
Reply

Bijoy says
March 14, 2015 at 9:11 am
I can modify the object using ce.testMap.put("10", "ten");
Output:
Performing Deep Copy for Object initialization
true
false
ce id:10

ce name:original
ce testMap:{2=second, 1=first}
ce id after local variable change:10
ce name after local variable change:original
ce testMap after local variable change:{2=second, 1=first, 10=ten}
ce testMap after changing variable from accessor methods:{2=second, 1=first, 10=ten}
Reply

parasuram tanguturu says
October 16, 2016 at 8:45 am
how to overcome this case;
ce.testMap.put(“10”, “ten”);
//ce testMap after local variable change:
{2=second, 1=first, 10=ten}
//ce testMap after changing variable from accessor methods:{2=second, 1=first, 10=ten}
Reply

JavaRocker says
April 27, 2017 at 12:04 am
testMap is private variable and out side class it wont be available.
Its just a Example code that is why in main you are able to do ce.testMap but in real application you wont be as generally you dont do such operations in POJO class.
Reply

ahmed says
March 7, 2015 at 1:34 am
thanx
i disable ABP for u
Reply

Pankaj says
March 7, 2015 at 9:50 am
Thanks Ahmed, I appreciate it.

Reply

S says
February 12, 2015 at 3:26 am
Excellent post!!
Reply

mahi says
February 1, 2015 at 7:34 pm
Can you please describe no 6. more deeply.I am not able to understand it.
Reply

Rais Alam says
November 22, 2014 at 6:45 am
Hi Pankaj,
It was a great learning about creating Immutable objects.If you are performing step 5 and 6 then step 4 is not required I guess. You are not storing or returning original reference of HashMap, You are using clone concept for that, Hence as a result client application have no way to reassign new object to declared HaspMap.
Please correct me If I am missing some thing
Thanks& Regards
Rais Alam
Reply

Ramakant says
November 10, 2014 at 12:05 pm
Should not be String name declared as a not final? Its not mutable anyway.
Reply

Marwen says
August 18, 2014 at 2:40 am
Thanks for the detailed tutorial, well written and the flow goes exactely to showing up almost the need of every instruction in the code ☺
One side question, even if I know we are talking about Objects immutability,but what about the other instance variables you introucted in the `FinalClassExample` (*id*, *name*)?
Is there any way to make them immutable?
Reply

Pankaj says
August 18, 2014 at 5:04 am
int and String both are already immutable, since there are no setter methods for them. For any other class variables, you should return a deep copy of the variable to avoid mutability.
Reply

Mirey says
September 19, 2013 at 6:38 pm
Thanks, you know it and you know how to **explain** it too! I will definitely read more of your articles ☺
Reply

Anish Sneh says
June 2, 2012 at 9:59 am
Thanks mate, great details..
— Anish Sneh
Reply

Pankaj says
October 17, 2012 at 6:47 pm
Thanks for the kind words.
Reply

Łomża Zuhlke says
May 5, 2011 at 5:43 pm
It's super webpage, I was looking for something like this
Reply

whaley says
December 23, 2010 at 12:35 pm
Out of curiosity, why the requirement to have the class be marked as final so as not to be extended? What does disallowing subclasses actually provide in terms of allowing objects of this type to be immutable?
Further, you don't have to mark fields as private only just so long as you can guarantee that all constructor's of the class properly initialize all of the fields.
As a side note, you *can* have setters, but with the nuance that instead of changing an internal field, what the setter really does is specify a return type of the class the method is on, and then behind the scenes creates a new object using a constructor that accepts all internal fields, using the internally held state in for all params with the exception of the field represented by the setter called since you want the new object to have that field updated.
Reply

Pankaj says
December 23, 2010 at 3:52 pm
If the class is not marked as final then its function can be overridden in the subclass either accidentally or intentionally. So its more related to keep the object secure. For this either all the getter methods can be made final or the class itself – this is again a design decision and depends on the requirement.
Again if the fields wont be private then client application can override the value. Make the HashMap as public in the code and run the below code to see yourself.

```
FinalClassExample fce = new
FinalClassExample(1,"", new HashMap());
```

```
System.out.println(fce.testMap);
HashMap hm = fce.testMap;
hm.put("1", "1");
System.out.println(fce.testMap);
```

Having a setter function will give the feeling that the actual object has been modified whereas internally creating a new object. Its better to client application know that its immutable (like String).

Reply

Hamlet D'Arcy says

December 23, 2010 at 9:02 am

In Groovy you can annotate the class as @Immutable and get /almost/ similar results to the scala example without all the boilerplate. IMHO Scala is better for it's immutable support though.

Also, don't forget that Java Date, Dimension, and other JDK classes are not immutable as well, so you need to make defensive copies of those classes as well.

Reply

Pankaj says

December 23, 2010 at 9:48 am

Exactly, for all the mutable objects we need to return the defensive copy rather than same object reference.

Have to dig into Scala now.

Reply

John says

December 23, 2010 at 8:30 am

why don't you just do this:

```
import static
java.util.Collections.unmodifiableMap;
public final class FinalClassExample {
...
private final Map testMap;
```

```
public FinalClassExample(int i, String n,
Map m){
id = i;
name = n;
testMap = unmodifiableMap(new HashMap (m));
}
public Map getTestMap() {
return testMap;
}
...
}
```

Reply

Pankaj says

December 23, 2010 at 9:30 am

In this case, when we will get the testMap from getTestMap() function, we will not be allowed to modify it and it will throw exception. Also in that case again we are passing the reference and the values will change accordingly.

Try executing this class:

```
package com.journeldev.java;
import static
java.util.Collections.unmodifiableMap;
import java.util.HashMap;
import java.util.Map;
public final class FinalClassExample1 {
private final Map testMap;
public Map getTestMap() {
return testMap;
}
public FinalClassExample1(Map hm) {
this.testMap = unmodifiableMap(hm);
}
public static void main(String[] args) {
HashMap h1 = new HashMap();
h1.put("1", "first");
h1.put("2", "second");
FinalClassExample1 ce = new
FinalClassExample1(h1);
System.out.println("ce testMap:" +
ce.getTestMap());
h1.put("3", "third");
System.out.println("ce testMap after
local variable change:"+
ce.getTestMap());
```

```
Map hmTest = ce.getTestMap();
hmTest.put("4", "new");
System.out.println("ce testMap after
changing variable from accessor
methods:"+ ce.getTestMap());
}
}
```

Output will be:
ce testMap:{2=second, 1=first}
ce testMap after local variable change:{3=third, 2=second, 1=first}
Exception in thread “main”
java.lang.UnsupportedOperationException
at
java.util.Collections$UnmodifiableMap.put(Collections.java:1285)
at
com.journeldev.java.FinalClassExample1.main(FinalClassExample1.java:33)
Reply

Steve says
December 23, 2010 at 6:25 pm
Pankaj,
You should not be allowed to modify a map you get from getTestMap. Objects coming from an immutable object, I would expect to also be immutable so you can not change the internals of the object after it is created. A mutable map may imply that the objects map is changed. Sure you can document that the object returned is new, but being defensive upfront seems better.
Another alternative is to never return collections. Instead, define a forEach method that takes a function:
public void forEach(Function fn) {
for(Thing t : Iterable collection) {
fn.apply(t);
}
}
Reply

Pankaj says
December 24, 2010 at 6:23 am

I am not sure why the map returned from getTestMap method should not be allowed to modify. Suppose the list contains some 1000 items and client wants to add 5 more to make their own immutable object instance, in this case its better to let them modify the returned object and do whatever they want on it. Usually, we have setter methods to change the state of object, so just by changing the returned object you should not think that the object state has been changed because the internal implementation is not known to us(as a client).

Reply

John says

December 24, 2010 at 9:38 am

you didn't notice that I wrote:

```
this.testMap = modifiableMap(new
HashMap (m))

so you couldn't change the inner map
after construction.
and I agree with Steve that the map
returned by getTestMap() should not
be modifiable as this would give the
false impression that you're actually
changing the immutable object.
in that case I would prefer doing
something like this:

FinalClassExample example = new
FinalClassExample(...);
Map newMap = new
HashMap(example.getTestMap());
newMap.put("hello", "world");
```

Reply

Pankaj says

December 24, 2010 at 3:27 pm

Ok agreed that this will work but then its not generic. What if its some generic mutable class like Date or may be user defined class where you don't have this feature. This example is intended to provide a generic way to create immutable classes.
As for changing the object state, that is why we have setter methods and I still prefer it doing like that… but again it depends on your application requirements and design.
Reply

Shantanu Kumar says
December 23, 2010 at 4:11 am
Shouldn't it be "shallow copy" instead of "swallow copy" unless I am missing something?
Reply

Pankaj says
December 23, 2010 at 6:35 am
Thanks for pointing out the typo error. Corrected now to shallow copy.
Reply

Ken says
December 23, 2010 at 12:11 am
… and here is a translation of the entire exercise into Scala, except without the wasteful copying and cloning, and with correct equals and hashCode methods:
case class FinalClassExample(id: Int, name: String, testMap: Map[String,String])
Reply

Pankaj says
December 23, 2010 at 6:42 am
I didn't understood what are you trying to point here?
Reply

Paweł Chłopek says
December 23, 2010 at 8:03 am
I belive that he was trying to point out that Scala lets you create immutable classes in more concise way. I'm new to Scala myself and already I love and often use these one liners. Less code less opportunities to make mistakes ☺
Reply

Pankaj says
December 23, 2010 at 8:45 am
Thats a completely new thing to me… Got something new to learn now… will dig into this soon ☺
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *
Comment

Name *

Email *

POST COMMENT