HowToDoInJava

Java finalize() – Why We Should Not Use It?

Last Updated: April 11, 2022 By: Lokesh Gupta Java Basics Java Basics

The ***finalize()* method in Java** is called by garbage collector thread before reclaiming the memory allocated to the object. It is considered as a *destructor* in Java – though It is not similar to what we have in C/C++.

Let's learn when to use *finalize()* in Java – and why we should avoid using it, if possible.

> The JEP-421 (*Java 18*) has marked the finalization deprecated. It will be removed in a future release. The use of Cleaners is recommended.

Table Of Contents

1. The *finalize()* in Java

1.1. Syntax

It is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the *finalize()* method to dispose of system resources or to perform other cleanups.

Notice that `finalize()` is the **deprecated** method.

finalize method syntax

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

1.2. Implementing *finalize()*

A general **syntax to override finalize() method** is given below.

How to override finalize method

```
public class MyClass
{

  //Other fields and methods

  //...

  @Override
  protected void finalize() throws Throwable
  {
    try
    {
      //release resources here
    }
    catch(Throwable t)
    {
      throw t;
    }
```

```
      finally
      {
        super.finalize();
      }
    }
  }
```

2. The *finalize()* Execution is Not Guaranteed

Let's prove it using a program. I have written a simple Java runnable with one print statement in each block i.e. **try-catch-finally-finalize**.

I have also created another class that will create 3 instances of this runnable and then we will see the execution path.

TryCatchFinallyTest.java

```
public class TryCatchFinallyTest implements Runnable {

  private void testMethod() throws InterruptedException
  {
    try
    {
      System.out.println("In try block");
      throw new NullPointerException();
    }
    catch(NullPointerException npe)
    {
      System.out.println("In catch block");
    }
    finally
    {
      System.out.println("In finally block");
    }
  }

  @Override
```

```
  protected void finalize() throws Throwable {
    System.out.println("In finalize block");
    super.finalize();
  }

  @Override
  public void run() {
    try {
      testMethod();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

TestMain.java

```
public class TestMain
{
  @SuppressWarnings("deprecation")
  public static void main(String[] args) {
  for(int i=1;i<=3;i++)
  {
    new Thread(new TryCatchFinallyTest()).start();
  }
  }
}
```

Program output.

Output

```
In try block
In catch block
In finally block

In try block
In catch block
```

```
In finally block

In try block
In catch block
In finally block
```

Amazing, **finalize()** method has not been executed at all for any thread. So this proves what I stated already.

The reason which I can think of is – that finalizers are executed by a separate thread from the garbage collector. If JVM exits too early then the garbage collector does not get enough time to create and execute finalizers.

2.1. Can we force `finalize()` to execute?

The answer is yes. Yes, we can. Using `Runtime.runFinalizersOnExit(true);`

TestMain.java

```
public class TestMain
{
  @SuppressWarnings("deprecation")
  public static void main(String[] args) {
    for(int i=1;i<=3;i++)
    {
      new Thread(new TryCatchFinallyTest()).start();
      Runtime.runFinalizersOnExit(true);
    }
  }
}
```

Program output.

Output

```
In try block
In catch block
In finally block

In try block
In try block
In catch block
In finally block
In catch block
In finally block

In finalize block
In finalize block
In finalize block
```

There is another method also: `Runtime.getRuntime().runFinalization();` but, it only guarantees that GC will make its best efforts. Even our program is not able to run finalize() method for all 3 threads.

Moving forward, we have used `Runtime.runFinalizersOnExit(true)`. Well, this is another pit. This method has already been deprecated in JDK, for the following reason –

> "This method is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock."

So, in one way we can not guarantee the execution and in another way we the system in danger. *Better, don't use this method.*

3. Other Reasons for Not Using *finalize()*

1. `finalize()` methods **do not work in chaining** like constructors. It means when you call a constructor then constructors of all superclasses will be invoked

implicitly. But, in the case of `finalize()` methods, this is not followed. Ideally, parent class's `finalize()` should be called explicitly but it does not happen.

2. Suppose, you created a class and wrote its finalize method with care. Someone comes and extends your class and does not call `super.finalize()` in subclass's `finalize()` block, then super class's `finalize()` will never be invoked anyhow.
3. Any exception which is thrown by *finalize()* method is ignored by GC thread and it will not be propagated further, in fact *it will not be logged in your log files*. So bad, isn't it?

4. *finalize()* adds Heavy Penalty on Performance

In Effective Java (2nd edition) Joshua Bloch says:

> "Oh, and one more thing: there is a severe performance penalty for using finalizers. On my machine, the time to create and destroy a simple object is about 5.6 ns. Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers."

I also tried the above analysis on my system but I was not able to get that much difference. Although, there is some difference for sure. But that was around 30% on the original time. On time-critical systems, it is also a big difference.

5. How to Use *finalize()* Correctly

After all the above arguments, if you still find a situation where using finalize() is essential, then cross-check below points:

- Always call `super.finalize()` in your `finalize()` method.
- Do not put time-critical application logic in `finalize()`, seeing its unpredictability.

- Do not use `Runtime.runFinalizersOnExit(true);` as it can put your system in danger.
- Try to follow the below template for *finalize()* method

finalize() method template

```
@Override
protected void finalize() throws Throwable
{
  try{
    //release resources here
  }catch(Throwable t){
    throw t;
  }finally{
    super.finalize();
  }
}
```

In this post, we discussed Java finalize best practice, how can we call finalize method manually in Java, and why not to use finalize in Java applications.

Happy Learning !!

Recommended Reading:

1. Difference between final, finally and finalize in Java
2. Java puzzle – Why try-catch-finally blocks require braces?
3. Why Strings are Immutable in Java?
4. Introduction to MongoDB: Why MongoDB?
5. Java Statements
6. What is Block Statement in Java
7. Java Classpath
8. Java Comments
9. Java Naming Conventions
0. Guide to Finalization with Java Cleaners

33 thoughts on “Java finalize() – Why We Should Not Use It?”

Pawel

January 12, 2020 at 1:46 pm

The C language does not implement destructors.

Reply

ushak

July 30, 2019 at 5:14 pm

At the time of garbage collection the memory of an unreferenced object is reclaimed. But reclaiming the object memory does not guarantee that the resources it holds will be released. That’s where finalize() method can help though using it is problematic and tha’t why it is deprecated in Java 9.

Reply

Ramakrishna

May 31, 2016 at 11:11 pm

Hi Mr.Lokesh. Great!!
I would say guys please stick to the title of the post "Why finalize() method should not be used ".
Yes, you can try in your sample code and non-critical dev code just to understand the concepts already explained by Mr.Lokesh in the article (invoke super class's finalize method, understanding constructor chaining vs finalize; understanding points explained in effective java book,).

If you interviewer (a crazy person or trying to check candidates response) asks; , the most suitable answer is confidently respond that "...I prefer not to use finalize() method.
after that if you have been asked to explain the reason ..
ii)
iii)avoid to understand the tricky code if you are not sure how finalize works. But you should know that finalize() is not called by GC as already explained in the article.

Reply

Shubham Gupta

May 7, 2016 at 8:05 pm

Hi Lokesh,

Recently, I was asked what would happen if I do something like :

```
@Override
  protected void finalize() throws Throwable {
    System.out.println("In Finalize Block");
    TimeUnit.MINUTES.sleep(INTEGER.MAX_VALUE);
    super.finalize();
  }
```

I said that gc collector thread will have to wait for very long time(almost indefinitely). Inshort object won't get collected by GC. Am I correct ? GC thread only invokes the finalize ?

Reply

Lokesh Gupta

May 7, 2016 at 10:59 pm

GC does not invoke `finalize()` methods. There is separate thread for it in JVM whose single responsibility is to execute `finalize()` methods for all objects. Objects with finalizers that become eligible for garbage collection are not collected immediately but, with the hotspot JVM, are first put onto a queue where a single finalizer thread calls the finalizers on each object in the queue in turn.

In my view, correct answer is that it can bring the whole application down due to memory leaks – because eventually all objects who have `finalize()` methods will not be garbage collected and wait for finalizer thread to execute there `finalize()` methods and thus causing memory leaks. If there are sufficient number of suck objects, memory will be full and application will crash.

Reply

Shubham Gupta

May 18, 2016 at 2:23 pm

Hello Lokesh,

Thank You for your previous response. Now as per your answer finalize is called by

```
is separate thread for it in JVM whose single responsibility is to execu
```

.

I stumbled upon this

```
https://stackoverflow.com/questions/28832/java-and-manually-executing-f
```

question in stackoverflow. Now here if you see finalize is being called twice. Two questions –

1- So does it means gc is also calling finalize ?
2- Is it possible to ask gc to collect one “particular object” at any point of time. When this question was asked to me I told that either by referencing that object to null or by calling its finalize method and then calling System.gc(). Something like this:

```
Employee emp = new Employee();
               emp = null;    // or by doing emp.finalize()
               System.gc();
```

One thing which I noticed while doing emp.finalize(). Even after doing emp.finalize(), i still had the reference to object which means it will not go for gc. so i guess by doing null referencing I can achieve this ?

Reply

Yousha

March 13, 2016 at 1:34 am

Dude fix your post…
Some contents are broken

Reply

Lokesh Gupta

March 13, 2016 at 3:21 pm

Thank you very much for pointing this out. Fixed it !!

Reply

Peng Huang

August 15, 2015 at 3:50 pm

Hello, for PhantomReference, One use is during a finalize method, which guarantees that the object is not resurrected during finalization and thus can be

garbage collected in a single cycle, rather than needing to wait for a second GC cycle to ensure that it has not been resurrected.

I don't understand it, can you help explain it?

Reply

Simon

January 6, 2015 at 7:43 am

Sometimes we need doing this. It cans happened when we need either free not managed resources or , for example inform application about the instance is not available. But it should not be a usual practice in any case.

Reply

Lokesh Gupta

January 6, 2015 at 8:04 am

True

Reply

Sumeet Saini

April 20, 2014 at 1:13 pm

Nice post and Great work lokesh sir, our post are really helpful for freshers and exp people……thanx please continue it

Reply

aditya

April 10, 2014 at 12:27 pm

Thanks for the nice post:)

Reply

meet

March 19, 2014 at 2:13 pm

nice article. I think finally construct is really required to release non memory resources in java & C#(C# Destructor will be translated in finalize() method by C# compiler). because it is not guarantee of the execution of finalize but finally clause will always execute so we can write code in finally to release non memory resources. C++ don't require finally because when the object is destroyed the destructor will be called and we can write code to release non-memory resources in destructor. Correct me if i wrong.

Reply

Lokesh Gupta

March 19, 2014 at 2:31 pm

You are right. 10 on 10.

Reply

Pravin

January 29, 2014 at 10:10 am

Dear Lokesh,
If object become abandon or eligible for gc, then why gc will call finalize() method before sweeping that obj? what will do finalize() method once called by gc?

Reply

Lokesh Gupta

January 29, 2014 at 10:51 am

GC will not call finalize. It's responsibility of JVM. The idea behind finalizes is that application can perform some clean up before objects using that resource garbage collected and leave resource open without any use. If the object responsible for cleanup is garbage collected before even cleaning up resource, then it is against the desired use case.

Reply

Gobbly

March 17, 2014 at 6:54 pm

This is correct. Finalizing has nothing really to do with GC in an under-the-hood sense. Each object that overrides the Object finalize method will have a Finalizer object associated with it. This is a reference, and as such GC will not recognize the object as unreferenced and will not consider it for GC. Once it has been finalized then GC can clean it up as an unreferenced object.

Reply

maheswari

January 3, 2014 at 10:53 am

what happend if we didn't call super.finalize method.without implementation of finalize method we should release non java resurces by using finally block.why should we implement finalize method.

Reply

Lokesh Gupta

January 3, 2014 at 11:16 pm

Best approach is that don;t use finalize(). Use finally instead to clean up resources in normal routine and attach shutdown hooks for application

shutdown.

If we didn't call super.finalize method in child class, then it will not be invoked.

Reply

Alvin P. Reyes

December 18, 2013 at 5:54 pm

For the record, the garbage collector is the one that is not guaranteed – this, in effect makes the finalize() method not guaranteed as well. Yes, even if you set the object to null. I would agree though that its not a good practice / recommendation to use the finalize() method to perform critical post-processing (closing DB connections or IO etc).

Reply

Gobbly

March 17, 2014 at 6:58 pm

To clarify, you likely mean that you set the reference to null. Also, as stated before, GC does not perform finalization.

Reply

Jvanaa

December 16, 2013 at 5:13 pm

Very useful article. Thanks.

Reply

Raam

November 28, 2013 at 8:24 pm

Good and interesting topic , Learning some very important things.

Reply

mun

October 28, 2013 at 2:52 pm

Why are you catching Throwable t and then rethrowing it? You can just leave it uncaught.

Reply

Lokesh Gupta

October 28, 2013 at 10:59 pm

Fair enough. But, This is only template and emphasis is on calling "super.finalize();" in finalize block.

Reply

Manoj

October 25, 2013 at 12:07 pm

Good post, thanks

Reply

JAI

October 18, 2013 at 10:33 am

Really helpful .. carry on.....

Reply

abc

August 27, 2013 at 2:02 pm

Good post

Reply

Nitya

August 15, 2013 at 10:29 pm

what happens when we call finalize()..

Reply

Lokesh Gupta

August 15, 2013 at 11:16 pm

Nothing special. It is regular method call. Just like your own cleanUp() method.

Reply

Brijesh

July 7, 2013 at 12:49 am

Now I am really scared of using finalize.. anyways thanks for the post.

Reply

Tomek

July 4, 2013 at 7:24 pm

Nice article, thanks.

Reply

Leave a Comment

Name *

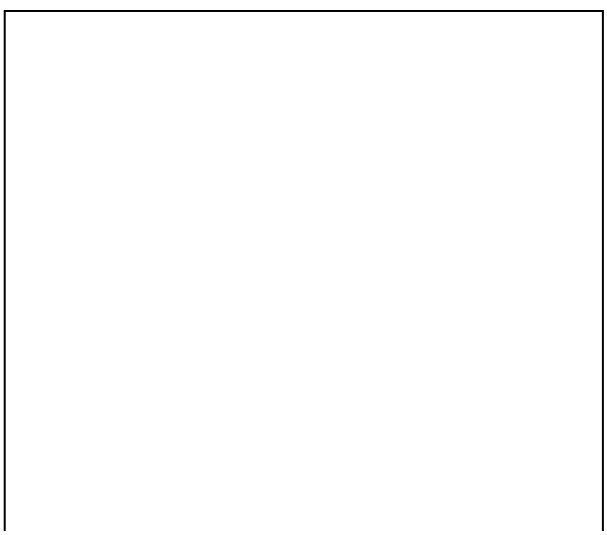
Email *

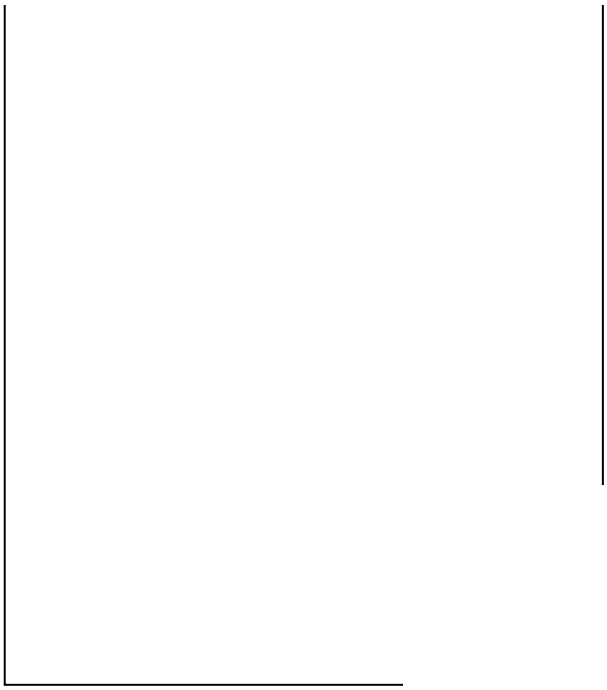
Website

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

Post Comment

Search ...





HowToDoInJava

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