<http://www.java-samples.com/showtutorial.php?tutorialid=145>

Garbage collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.

Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs

The finalize () Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the java run time calls the **finalize()** method on the object.

The **finalize()** method has this general form:

protected void finalize()

{

// finalization code here

}

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

It is important to understand that finalize() is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

<http://javarevisited.blogspot.in/2012/03/finalize-method-in-java-tutorial.html>

# [10 points on finalize method in Java – Tutorial Example](http://javarevisited.blogspot.in/2012/03/finalize-method-in-java-tutorial.html)

**finalize method in java** is a special method much like the [main method in java](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html). finalize() is called before Garbage collector reclaim the Object, its last chance for any object to perform cleanup activity i.e. releasing any system resources held, closing connection if open etc. Main issue with finalize method in Java is it's not guaranteed by JLS that it will be called by Garbage collector or exactly when it will be called, for example, an object may wait indefinitely after becoming [eligible for garbage collection](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) and before its finalize() method gets called. similarly even after finalize gets called it's not guaranteed it will be immediately collected. Because of above reason it makes no sense to finalize method for releasing critical resources or perform any time critical activity inside finalize. It may work in development in one JVM but may not work in other JVM. In this Java tutorial, we will see some **important points about finalize method in Java**, *How to use finalize method*, what to do and what not to do inside finalize in Java.  Btw, the best advice on finalize method is given by none other than Joshua Bloch on his time less classic book, [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20). Don't forget to read that after reading this article.

## What is finalize method in Java – Tutorial Example

1) finalize() method is defined in java.lang.Object class, which means it available to all the classes for the sake of overriding. finalize method is defined as protected which leads to a popular core java question "**Why finalize is declared protected instead of public**"? well, I don't know the exact reason its falls in the same category of questions like [why java doesn't support multiple inheritance](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html) which can only be answered accurately by designers of Java. any way making finalize protected looks good in terms of following rule of encapsulation which starts with least restrictive access modifier like private but making finalize private prevents it from being overridden in subclass as you can not override private methods, so making it protected is next obvious choice.

2) One of the most important points of finalize method is that it's *not automatically chained like* [*constructors*](http://javarevisited.blogspot.com/2012/01/what-is-constructor-overloading-in-java.html). If you are overriding finalize method then it's your responsibility to call finalize() method of the superclass, **if you forgot to call then finalize of super class will never be called**. so it becomes critical to remember this and provide an opportunity to finalize of super class to perform cleanup. The best way to call super class finalize method is to call them in the finally block as shown in below example. This will guarantee that finalize of the parent class will be called in all condition except when [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) exits:

 @[**Override**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)  
    **protected** **void** finalize() **throws** **Throwable** {  
        **try**{  
            **System**.out.println("Finalize of Sub Class");  
            *//release resources, perform cleanup ;*  
        }**catch**(**Throwable** t){  
            **throw** t;  
        }**finally**{  
            **System**.out.println("Calling finalize of Super Class");  
            **super**.finalize();  
        }  
        
    }

3) finalize method is called by **garbage collection thread** before collecting object and if not intended to be called like a normal method.

4) finalize gets called only once by GC thread if object revives itself from finalize method than **finalize will not be called again**.

5) Any [Exception](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html) is thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact, I doubt if you find any trace of it.

6) There is one way to increase the probability of running of finalize method by calling System.runFinalization() and

Runtime.getRuntime().runFinalization(). These methods put more effort that JVM call finalize() method of all object which are eligible for garbage collection and whose finalize has not yet called. It's not guaranteed, but JVM tries its best.

## Alternative of finalize method for cleanup.

[what is finalize method in Java – Tutorial Example](http://javarevisited.blogspot.com/2011/11/ldap-authentication-active-directory.html)So far its seems we are suggesting not to use finalize method because of its non-guaranteed behavior but than what is alternative of releasing resource, performing cleanup because there is no destructor in Java. Having a method like close() or destroy() make much sense for releasing resources held by classes. In fact, [JDK](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) library follows this. if you look at [java.io](http://docs.oracle.com/javase/1.5.0/docs/api/java/io/package-frame.html) package which is a great example of acquiring system resource like **file descriptor** for opening file, offers close() method for opening stream and close() for closing it. In fact its one of the best practice to call the close  method from the finally block in java. Only caveat with this approach is its not automatic, client has to do the cleanup and if client forgot to do cleanup there are chances of resources getting leaked, which again suggest us that we could probably give another chance to finalize method. You will be pleased to know that [Java 7 has added automatic resource management feature](http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html) which takes care of closing all resource opened inside try block automatically, leaving no chance of manual release and leakage.

When to use finalize method in JavaIn the last paragraph, I pointed out that there are certain cases where overriding finalize make sense like an ultimate last attempt to cleanup the resource. If a [Java class](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) is made to held resource like input-output devices, JDBC connection then you should override finalize and call its close() method from finalize. though there is no guarantee that it will run and release the resource timely best part is we are not relying on it. It just another last attempt to release the resource which most likely have been already released due to the client calling the close() method. This technique is heavily used inside Java Development library. look at below example of finalize method from [FileInputStream.java](http://docs.oracle.com/javase/1.5.0/docs/api/java/io/FileInputStream.html)

protected void finalize() throws IOException {

if ((fd != null) && (fd != FileDescriptor.in)) {

/\*

\* Finalize should not release the FileDescriptor if another

\* stream is still using it. If the user directly invokes

\* close() then the FileDescriptor is also released.

\*/

runningFinalize.set(Boolean.TRUE);

try {

close();

} finally {

runningFinalize.set(Boolean.FALSE);

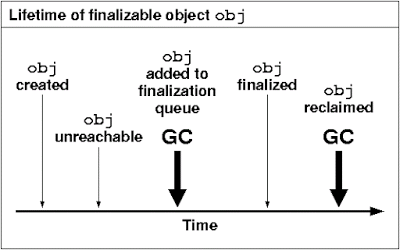
}

}

}

### What not to do in finalize method in Java

trusting finalize method for releasing critical resource is biggest [mistake java programmer can make](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html). suppose instead of relying on close() method to release file descriptor, you rely on finalize to relapse it for you. Since there is no guaranteed when finalize method will run you could effectively lock hundreds of file-descriptor of earlier opened file or socket and there is high chance that your application will run out of file-descriptor and not able to [open any new file](http://javarevisited.blogspot.com/2011/12/read-and-write-text-file-java.html). It's best to use finalize as the last attempt to do cleanup but **never use finalize as a first or only attempt**.

[](http://4.bp.blogspot.com/-gOrgSAxNSBU/VhUOrp9gObI/AAAAAAAAD6I/dgtaHmSH2dY/s1600/Finalize%2Bmethod%2Bin%2BJava.gif)

That's all on **finalize method in Java**. as you have seen there are quite a lot of *specifics about finalize method* which java programmer should remember before using finalize in java. In one liner don’t do time critical task on finalize method but use finalize with caution.