A Guide to the finalize Method in Java

Last modified: January 25, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

In this tutorial, we'll focus on a core aspect of the Java language – the *finalize* method provided by the root *Object* class.

Simply put, this is called before the garbage collection for a particular object.

2. Using Finalizers

The *finalize()* method is called the finalizer.

Finalizers get invoked when JVM figures out that this particular instance should be garbage collected. Such a finalizer may perform any operations, including bringing the object back to life.

The main purpose of a finalizer is, however, to release resources used by objects before they're removed from the memory. A finalizer can work as the primary mechanism for clean-up operations, or as a safety net when other methods fail.

To understand how a finalizer works, let's take a look at a class declaration:

```
public class Finalizable {
    private BufferedReader reader;

    public Finalizable() {
        InputStream input = this.getClass()
          .getClassLoader()
          .getResourceAsStream("file.txt");
        this.reader = new BufferedReader(new InputStreamReader(input));
    }

    public String readFirstLine() throws IOException {
        String firstLine = reader.readLine();
        return firstLine;
    }

    // other class members
}
```

The class *Finalizable* has a field *reader*, which references a closeable resource. When an object is created from this class, it constructs a new *BufferedReader* instance reading from a file in the classpath.

Such an instance is used in the *readFirstLine* method to extract the first line in the given file. **Notice that the reader isn't closed in the given code.**

We can do that using a finalizer:

```
@Override
public void finalize() {
    try {
        reader.close();
        System.out.println("Closed BufferedReader in the finalizer");
    } catch (IOException e) {
        // ...
    }
}
```

It's easy to see that a finalizer is declared just like any normal instance method.

In reality, **the time at which the garbage collector calls finalizers is dependent on the JVM's implementation and the system's conditions, which are out of our control.**

To make garbage collection happen on the spot, we'll take advantage of the *System.gc* method. In real-world systems, we should never invoke that explicitly, for a number of reasons:

1. It's costly
2. It doesn't trigger the garbage collection immediately – it's just a hint for the JVM to start GC
3. JVM knows better when GC needs to be called

If we need to force GC, we can use *jconsole* for that.

The following is a test case demonstrating the operation of a finalizer:

```
@Test
public void whenGC_thenFinalizerExecuted() throws IOException {
    String firstLine = new Finalizable().readFirstLine();
    assertEquals("baeldung.com", firstLine);
    System.gc();
}
```

In the first statement, a *Finalizable* object is created, then its *readFirstLine* method is called. This object isn't assigned to any variable, hence it's eligible for garbage collection when the *System.gc* method is invoked.

The assertion in the test verifies the content of the input file and is used just to prove that our custom class works as expected.

When we run the provided test, a message will be printed on the console about the buffered reader being closed in the finalizer. This implies the *finalize* method was called and it has cleaned up the resource.

Up to this point, finalizers look like a great way for pre-destroy operations. However, that's not quite true.

In the next section, we'll see why using them should be avoided.

3. Avoiding Finalizers

Let's have a look at several problems we'll be facing when using finalizers to perform critical actions.

The first noticeable issue associated with finalizers is the lack of promptness. We cannot know when a finalizer is executed since garbage collection may occur anytime.

By itself, this isn't a problem because the most important thing is that the finalizer is still invoked, sooner or later. However, system resources are limited. **Thus, we may run out of those resources before they get a chance to be**

cleaned up, potentially resulting in system crashes.

Finalizers also have an impact on the program's portability. Since the garbage collection algorithm is JVM implementation dependent, a program may run very well on one system while behaving differently at runtime on another.

Another significant issue coming with finalizers is the performance cost. Specifically, **JVM must perform much more operations when constructing and destroying objects containing a non-empty finalizer**.

The details are implementation-specific, but the general ideas are the same across all JVMs: additional steps must be taken to ensure finalizers are executed before the objects are discarded. Those steps can make the duration of object creation and destruction increase by hundreds or even thousands of times.

The last problem we'll be talking about is the lack of exception handling during finalization. **If a finalizer throws an exception, the finalization process is canceled, and the exception is ignored, leaving the object in a corrupted state** without any notification.

4. No-Finalizer Example

Let's explore a solution providing the same functionality but without the use of *finalize()* method. Notice that the example below isn't the only way to replace finalizers.

Instead, it's used to demonstrate an important point: there are always options that help us to avoid finalizers.

Here's the declaration of our new class:

```
public class CloseableResource implements AutoCloseable {
    private BufferedReader reader;

    public CloseableResource() {
        InputStream input = this.getClass()
          .getClassLoader()
          .getResourceAsStream("file.txt");
        reader = new BufferedReader(new InputStreamReader(input));
    }

    public String readFirstLine() throws IOException {
        String firstLine = reader.readLine();
        return firstLine;
    }

    @Override
    public void close() {
        try {
            reader.close();
            System.out.println("Closed BufferedReader in the close metho
        } catch (IOException e) {
            // handle exception
        }
    }
}
```

It's not hard to see that the only difference between the new *CloseableResource* class and our previous *Finalizable* class is the implementation of the *AutoCloseable* interface instead of a finalizer definition.

Notice that the body of the *close* method of *CloseableResource* is almost the same as the body of the finalizer in class *Finalizable*.

The following is a test method, which reads an input file and releases the resource after finishing its job:

```
@Test
public void whenTryWResourcesExits_thenResourceClosed() throws IOExceptio
    try (CloseableResource resource = new CloseableResource()) {
        String firstLine = resource.readFirstLine();
        assertEquals("baeldung.com", firstLine);
    }
}
```

In the above test, a *CloseableResource* instance is created in the *try* block of a try-with-resources statement, hence that resource is automatically closed when the try-with-resources block completes execution.

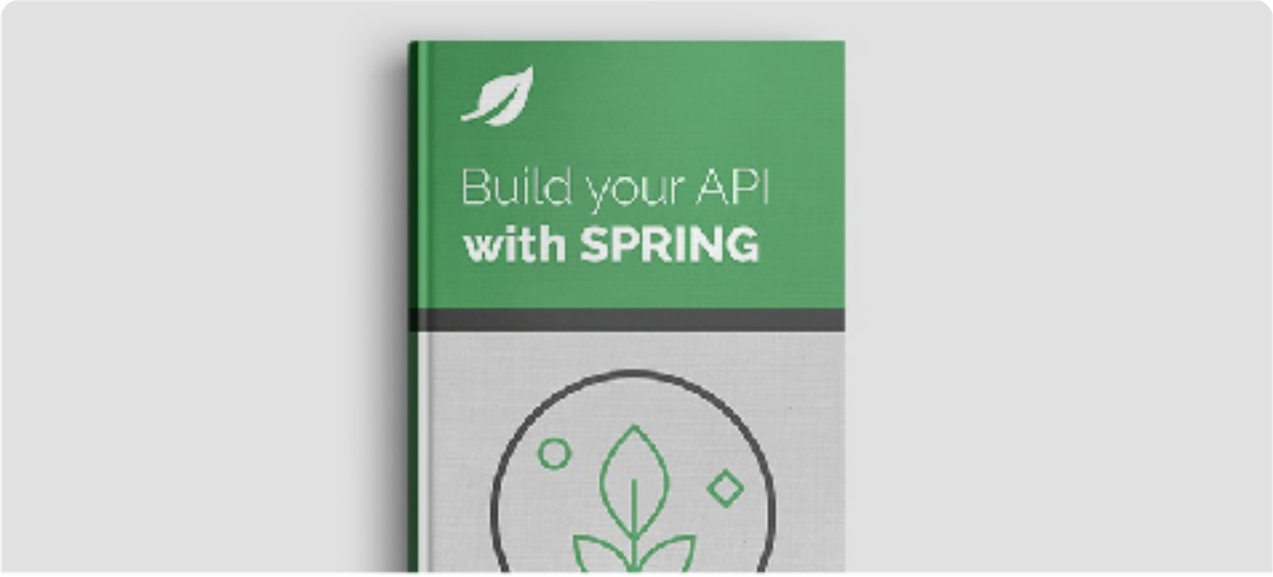
Running the given test method, we'll see a message printed out from the *close* method of the *CloseableResource* class.

5. Conclusion

In this tutorial, we focused on a core concept in Java – the *finalize* method. This looks useful on paper but can have ugly side effects at runtime. And, more importantly, there's always an alternative solution to using a finalizer.

One critical point to notice is that *finalize* has been deprecated starting with Java 9 – and will eventually be removed.

As always, the source code for this tutorial can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java/).



(http://www.baeldung.com/wp-content/uploads/2018/04/baeldung-rest-post-footer-main-1.2.0-newcover-1.jpg)


(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

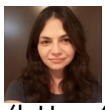
Subscribe ▾ ▲ newest ▲ oldest ▲ most voted

Guest

Bruno

Thanks. What about the final keyword ? Other usecase but another article it would be nice.

✚ 0 ━ 4 months ago ^



(http://www.baeldung.com/author/loredana-crusoveanu/)

Editor

Loredana Crusoveanu (http://www.baeldung.com/author/loredana-crusoveanu/)

Hey Bruno,

Thanks, an interesting suggestion. We'll add that to our content calendar.

Cheers.

✚ 0 ━ 3 months ago

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)

HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)
MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)