RAII in Java... is resource disposal always so ugly?

Asked 16 years, 2 months ago Modified 13 years, 2 months ago Viewed 5k times

17

I just played with Java file system API, and came down with the following function, used to copy binary files. The original source came from the Web, but I added try/catch/finally clauses to be sure that, should something wrong happen, the Buffer Streams would be closed (and thus, my OS ressources freed) before quiting the function.

I trimmed down the function to show the pattern:

```
public static void copyFile(FileOutputStream oDStream, FileInputStream
oSStream) throw etc...
{
   BufferedInputStream oSBuffer = new BufferedInputStream(oSStream, 4096);
   BufferedOutputStream oDBuffer = new BufferedOutputStream(oDStream, 4096);

   try
   {
      try
      {
         int c;

         while((c = oSBuffer.read()) != -1)  // could throw a IOException
         {
            oDBuffer.write(c);  // could throw a IOException
         }
      }
      finally
      {
         oDBuffer.close(); // could throw a IOException
      }
   }
   finally
   {
      oSBuffer.close(); // could throw a IOException
   }
}
```

As far as I understand it, I cannot put the two `close()` in the finally clause because the first `close()` could well throw, and then, the second would not be executed.

I know C# has the *Dispose* pattern that would have handled this with the `using` keyword.

I even know better a C++ code would have been something like (using a Java-like API):

```
void copyFile(FileOutputStream & oDStream, FileInputStream & oSStream)
{
   BufferedInputStream oSBuffer(oSStream, 4096);
   BufferedOutputStream oDBuffer(oDStream, 4096);

   int c;

   while((c = oSBuffer.read()) != -1)  // could throw a IOException
   {
      oDBuffer.write(c);  // could throw a IOException
   }

   // I don't care about resources, as RAII handle them for me
}
```

I am missing something, or do I really have to produce ugly and bloated code in Java just to handle exceptions in the `close()` method of a Buffered Stream?

(Please, tell me I'm wrong somewhere...)

EDIT: Is it me, or when updating this page, I saw both the question and all the answers decreased by one point in a couple of minutes? Is someone enjoying himself too much while remaning anonymous?

EDIT 2: **McDowell** offered a very interesting link I felt I had to mention here: http://illegalargumentexception.blogspot.com/2008/10/java-how-not-to-make-mess-of-stream.html

EDIT 3: Following McDowell's link, I tumbled upon a proposal for Java 7 of a pattern similar to the C# using pattern: http://tech.puredanger.com/java7/#resourceblock . My problem is explicitly described. Apparently, even with the Java 7 `do`, the problems remain.

java design-patterns raii resource-management

Share

Improve this question

Follow

edited Jan 6, 2011 at 9:43

asked Oct 11, 2008 at 16:16

paercebal

83.2k ● 38 ● 134 ● 160

Apologies for my misguided answer. I wasn't sure whether you really were into figuring out a way of doing RAII in Java, or simply didn't know how one usually copies data from an input stream into an output stream in Java. – Alexander Oct 11, 2008 at 16:58

No problemo... The fact is that I did not know, too, how to make a clean and efficient copy... :-p ... – paercebal Oct 12, 2008 at 17:47

You can save 8 lines of code in first listing and 2 lines in second listing if you will use the java code convention. In that case this code will be not so ugly. – msangel Mar 4, 2013 at 2:56

@msangel : `You can save 8 lines of code in first listing and 2 lines in second listing if you will use the java code convention. In that case this code will be not so ugly.` : You ***SO*** missed the point of this post... :-D – paercebal Mar 4, 2013 at 13:49

1 Allocation of `oDBuffer` should really go inside the outer `try` statement. Which goes to show how inconvenient this way of doing things is. – Nicola Musatti May 30, 2013 at 11:29

5 Answers

Sorted by: Highest score (default)

19

The try/finally pattern is the correct way to handle streams in most cases for Java 6 and lower.

Some are advocating silently closing streams. Be careful doing this for these reasons: Java: how not to make a mess of stream handling

---

Java 7 introduces *try-with-resources*:

```
/** transcodes text file from one encoding to another */
public static void transcode(File source, Charset srcEncoding,
                             File target, Charset tgtEncoding)
                                                             throws IOException
{
    try (InputStream in = new FileInputStream(source);
         Reader reader = new InputStreamReader(in, srcEncoding);
         OutputStream out = new FileOutputStream(target);
         Writer writer = new OutputStreamWriter(out, tgtEncoding)) {
        char[] buffer = new char[1024];
        int r;
        while ((r = reader.read(buffer)) != -1) {
            writer.write(buffer, 0, r);
        }
    }
}
```

`AutoCloseable` types will be automatically closed:

```
public class Foo {
  public static void main(String[] args) {
    class CloseTest implements AutoCloseable {
      public void close() {
        System.out.println("Close");
      }
    }
    try (CloseTest closeable = new CloseTest()) {}
  }
}
```

Share

Improve this answer

Follow

edited Oct 1, 2011 at 11:48

answered Oct 11, 2008 at 16:33

McDowell

109k ● 31 ● 206 ● 270

In most case, but funnily enough not in this case. :) – Tom Hawtin - tackline Oct 11, 2008 at 16:45

@Tom - yes, this is not a good stream copy mechanism + I would go with yours. – McDowell Oct 11, 2008 at 16:51

My point was more about RAII than about the exact implementation of code using BufferOutputStream. Your link is the right answer to my RAII question. As an amusing side note, I had an opportunity to work with a new project on Java, but declined the invitation, and went for another on .NET instead (almost) only because of C#'s `using` and C++/CLI destructors and finalizers... – paercebal Jan 19, 2011 at 19:47

@McDowell : Thanks for the update on the try-with-resources code example. I would upvote your answer again, if I could do it more than once... :-) ... – paercebal Oct 28, 2011 at 17:14

4

There are issues, but the code you found lying about on the web is really poor.

Closing the buffer streams closes the stream underneath. You really don't want to do that. All you want to do is flush the output stream. Also there's no point in specifying the underlying streams are for files. Performance sucks because you are copying one byte at a time (actually if you use java.io use can use transferTo/transferFrom which is a bit faster still). While we are about it, the variable names suck to. So:

```
public static void copy(
    InputStream in, OutputStream out
) throw IOException {
    byte[] buff = new byte[8192];
    for (;;) {
        int len = in.read(buff);
        if (len == -1) {
            break;
        }
        out.write(buff, 0, len);
    }
}
```

If you find yourself using try-finally a lot, then you can factor it out with the "execute around" idiom.

In my opinion: Java should have someway of closing resources at end of scope. I suggest adding `private` as a unary postfix operator to close at the end of the enclosing block.

Share Improve this answer Follow

answered Oct 11, 2008 at 16:36

Tom Hawtin - tackline
147k ● 30 ● 221 ● 312

Thanks for the better code. For my current personnal project, it is not very important, but I copy/paste your code right now as a future replacement. +1. – paercebal Oct 11, 2008 at 16:53

If he's copying the file, then he probably *does* want to close the streams when he's done. The copy is complete, so there's no point in leaving the streams open. In this case, his nested try-finally blocks + close() calls are appropriate. – Derek Park Oct 11, 2008 at 17:54

Derek Park is right. While your code interested me, it still misses the point of the question, that is, resource disposal. Let's say I have a `copyFile(String in, String out)` method instanciating the FileOutputStream and FileInputStream, and calling this `copy(InputStream in, OutputStream out)` method, how `copyFile` ought to be written to handle the resource disposal correctly? – paercebal Jan 19, 2011 at 19:27

4

Unfortunately, this type of code tends to get a bit bloated in Java.

By the way, if one of the calls to oSBuffer.read or oDBuffer.write throws an exception, then you probably want to let that exception permeate up the call hierarchy.

Having an unguarded call to close() inside a finally-clause will cause the original exception to be replaced by one produced by the close()-call. In other words, a failing close()-method may hide the original exception produced by read() or write(). So, I think you want to ignore exceptions thrown by close() if *and only if* the other methods did not throw.

I usually solve this by including an explicit close-call, inside the inner try:

```
try {
  while (...) {
    read...
    write...
  }
  oSBuffer.close(); // exception NOT ignored here
  oDBuffer.close(); // exception NOT ignored here
} finally {
  silentClose(oSBuffer); // exception ignored here
  silentClose(oDBuffer); // exception ignored here
}
```

```
static void silentClose(Closeable c)  {
  try {
    c.close();
  } catch (IOException ie) {
    // Ignored; caller must have this intention
```

```
        }
    }
```

Finally, for performance, the code should probably work with buffers (multiple bytes per read/write). Can't back that by numbers, but fewer calls should be more efficient than adding buffered streams on top.

Share
Improve this answer
Follow

edited Oct 11, 2008 at 18:49

answered Oct 11, 2008 at 16:29
volley
6,701 ● 1 ● 29 ● 28

If you silently close this way, your code does no error handling if close throws an exception. Many streams (like the BufferedOutputStream) write data on close. – McDowell Oct 11, 2008 at 16:36

McDowell: Yes it should catch exceptions if close() throws an Exception. Note that the close()-calls are first made INSIDE the try-block! The finally-block is to ensure cleanup if any methods throw an exception. Right? (Note, I forgot the close of one of the buffers in the first post version.) – volley Oct 11, 2008 at 16:41

BufferedOutputStream is a bit of a bugger. I favour an explicit flush (in the non-exception case), but you have to remember it. IIRC, the close had broken exception handing before Java SE 1.6. – Tom Hawtin - tackline Oct 11, 2008 at 16:53

I know exceptions from close() are vital. But, if the close()-calls are made just after the final call to write(), then shouldn't that ensure proper tracking of exceptions? McDowell, please confirm whether there's a flaw here; I will then revoke the code myself, but I really want to know. :) – volley Oct 11, 2008 at 16:56

Some of the crypto/compression is a bit nasty. Not only do you have the underlying resource to deal with, but the implementation may also have some "C", non-Java memory resources. – Tom Hawtin - tackline Oct 11, 2008 at 17:37

3

Yes, that's how java works. There is control inversion - the user of the object has to know how to clean up the object instead of the object itself cleaning up after itself. This unfortunately leads to a lot of cleanup code scattered throughout your java code.

C# has the "using" keyword to automatically call Dispose when an object goes out of scope. Java has no such thing.

Share Improve this answer Follow

answered Oct 11, 2008 at 16:50
Dale

1 Whether there is a special syntax for it or not, the client code has to tell the resource when to clean up. There is no way the resource can tell. Of course you can abstract away the resource acquisition, set up and release, with the interesting code executed as a callback. – Tom Hawtin - tackline Oct 11, 2008 at 17:02

That's not true. C++ and C# objects do just fine disposing of themselves without any participation from the caller. Read up on these languages. – Eggs McLaren Oct 11, 2008 at 17:11

1 I do have some familiarity with those languages. In C# the client code calls the dispose method at the end of the using block. In C++ the client code calls the destructor at the end of scope. – Tom Hawtin - tackline Oct 11, 2008 at 17:22

Exactly. The C++ and C# caller doesn't have to participate in the destruction, nor does the caller have to know the inner workings of the object being destructed. Only Java has this control inversion where the caller must know not only how the object has to be destructed but its side effects. – Eggs McLaren Oct 11, 2008 at 17:27

2 In both C++ and C# the caller via appropriate syntax need only say "I want auto destruction" (C# via "using" and C++ via stack-based initialization) and the object itself takes care of the details. However in Java the caller has to perform the destruction themselves (calling close() or whatever). – Eggs McLaren Oct 11, 2008 at 17:48

2

For common IO tasks such as copying a file, code such as that shown above is reinventing the wheel. Unfortunately, the JDK doesn't provide any higher level utilities, but apache commons-io does.

For example, FileUtils contains various utility methods for working with files and directories (including copying). On the other hand, if you really need to use the IO support in the JDK, IOUtils contains a set of closeQuietly() methods that close Readers, Writers, Streams, etc. without throwing exceptions.

Share Improve this answer Follow

answered Oct 11, 2008 at 17:14

Dónal
187k ● 176 ● 581 ● 843