Better Resource Management with Java SE 7: Beyond Syntactic Sugar

By Julien Ponge

Published May 2011

This article presents the Java 7 answer to the automatic resource management problem in the form of a new language construct, proposed as part of Project Coin, called the *try-with-resources* statement.

Downloads:

Java SE 7 Preview

Example Source Files (zip)

Introduction

The typical Java application manipulates several types of resources such as files, streams, sockets, and database connections. Such resources must be handled with great care, because they acquire system resources for their operations. Thus, you need to ensure that they get freed even in case of errors. Indeed, incorrect resource management is a common source of failures in production applications, with the usual pitfalls being database connections and file descriptors remaining opened after an exception has occurred somewhere else in the code. This leads to application servers being frequently restarted when resource exhaustion occurs, because operating systems and server applications generally have an upper-bound limit for resources.

Correct practices for the management of resources and exceptions in Java have been well documented. For any resource that was successfully initialized, a corresponding invocation to its `close()` method is required. This requires disciplined usage of `try/catch/finally` blocks to ensure that any execution path from a resource opening eventually reaches a call to a method that closes it. Static analysis tools, such as FindBugs, are of great help in identifying such type of errors. Yet often, both inexperienced and experienced developers get resource management code wrong, resulting at best in resource leaks.

However, it should be acknowledged that writing correct code for resources requires lots of boilerplate code in the form of nested `try/catch/finally` blocks, as we will see. Writing such code correctly quickly becomes a problem of its own. Meanwhile, other programming languages, such as Python and Ruby, have been offering language-level facilities known as *automatic resource management* to address this issue.

This article presents the Java Platform, Standard Edition (Java SE) 7 answer to the automatic resource management problem in the form of a new language construct proposed as part of Project Coin and called the *try-with-resources statement*. As we will see, it goes well beyond being just more syntactic sugar, like the enhanced for loops of Java SE 5. Indeed, exceptions can mask each other, making the identification of root problem causes sometimes hard to debug.

The article starts with an overview of resource and exception management before introducing the essentials of the `try-with-resources` statement from the Java developer point of view. It then shows how a class can be made ready for supporting such statements. Next, it discusses the issues of exception masking and how Java SE 7 evolved to fix them. Finally, it demystifies the syntactic sugar behind the language extension and provides a discussion and a conclusion.

Note: The source code for the examples described in this article can be downloaded here: sources.zip

Managing Resources and Exceptions

Let us get started with the following code excerpt:

```
private void incorrectWriting() throws IOException {
        DataOutputStream out = new DataOutputStream(new FileOutputStream("data")
        out.writeInt(666);
        out.writeUTF("Hello");
        out.close();
    }
```

Copy

At first sight, this method does not do much harm: It opens a file called `data` and then writes an integer and a string. The design of the stream classes in the `java.io` package makes it possible for them to be combined using the decorator design-pattern.

As an example, we could have added an output stream for compressing data between a `DataOutputStream` and a `FileOutputStream`. When a stream is closed, it also closes the stream that it is decorating. Going back again to the example, when `close()` is called on the instance of `DataOutputStream`, so is the `close()` method from `FileOutputStream`.

There is, however, a serious issue in this method regarding the call to the `close()` method. Suppose an exception is thrown while writing the integer or the string because the underlying file system is full. Then, the `close()` method has no chance of being called.

This is not so much of an issue regarding `DataOutputStream`, because it operates only on instances of `OutputStream` to encode and write primitive data types into arrays of bytes. The real problem is on `FileOutputStream`, because it internally holds an operating system resource on a file descriptor that is freed only when `close()` is called. Hence, this method leaks resources.

This issue is mostly harmless in the case of short-lived programs, but it could lead to an entire server having to be restarted in the case of long-running applications, as found on Java Platform, Enterprise Edition (Java EE) application servers, because the maximum number of open file descriptors permitted by the underlying operating system would be reached.

A correct way to rewrite the previous method would be as follows:

```
private void correctWriting() throws IOException {
        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new FileOutputStream("data"));
            out.writeInt(666);
            out.writeUTF("Hello");
        } finally {
            if (out != null) {
                out.close();
            }
        }
    }
```

Copy

In every case, a thrown exception is propagated to the invoker of the method, but the `finally` block after the `try` block ensures that the `close()` method of the data output stream is called. In turn, this ensures that the underlying file output stream's `close()` method is called too, resulting in the proper freeing of the operating system resources associated with a file.

try-with-resources Statement for the Impatient

There is, admittedly, a lot of boilerplate code in the previous example to ensure that resources are properly closed. With more streams, network sockets, or Java Database Connectivity (JDBC) connections, such boilerplate code makes it harder to read the business logic of a method. Even worse, it requires discipline from developers because it is easy to write the error handling and resource closing logic incorrectly.

In the meantime, other programming languages have introduced constructs for simplifying the handling of such cases. As an example, the previous method would be written as follows in Ruby:

```
def writing_in_ruby
       File.open('rdata', 'w') do |f|
           f.write(666)
           f.write("Hello")
       end
   end
```

Copy

And it would be written like this in Python:

```
def writing_in_python():
       with open("pdata", "w") as f:
           f.write(str(666))
           f.write("Hello")
```

Copy

In Ruby, the `File.open` method takes a block of code to be executed, and ensures that the opened file is closed even if the block's execution emits an exception.

The Python example is similar in that the special `with` statement takes an object that has a `close` method and a code block. Again, it ensures proper resource closing no matter if an exception is thrown or not.

Java SE 7 introduced a similar language construct as part of Project Coin. The example can be rewritten as follows:

```
private void writingWithARM() throws IOException {
       try (DataOutputStream out
               = new DataOutputStream(new FileOutputStream("data"))) {
           out.writeInt(666);
           out.writeUTF("Hello");
       }
   }
```

Copy

The new construct extends `try` blocks to declare resources much like is the case with `for` loops. Any resource declared within a `try` block opening will be closed. Hence, the new construct shields you from having to pair `try` blocks with corresponding `finally` blocks that are dedicated to proper resource management. A semicolon separates each resource, for example:

```
try (
       FileOutputStream out = new FileOutputStream("output");
       FileInputStream  in1 = new FileInputStream("input1");
       FileInputStream  in2 = new FileInputStream("input2")
   ) {
       // Do something useful with those 3 streams!
   }   // out, in1 and in2 will be closed in any case
```

Copy

Finally, such a `try-with-resources` statement may be followed by `catch` and `finally` blocks, just like regular try statements prior to Java SE 7.

Making an Auto-Closeable Class

As you probably guessed already, a `try-with-resources` statement cannot manage every class. A new interface called `java.lang.AutoCloseable` was introduced in Java SE 7. All it does is provide a void method named `close()` that may throw a checked exception (`java.lang.Exception`). Any class willing to participate in `try-with-resources` statements should implement this interface. It is strongly recommended that implementing classes and sub-interfaces declare a more precise exception type than `java.lang.Exception`, or, even better, declare no exception type at all if invoking `close()` should not fail.

Such `close()` methods have been retro-fitted into many classes of the standard Java SE run-time environment , including the
`java.io, java.nio, javax.crypto, java.security, java.util.zip, java.util.jar, javax.net`
, and `java.sql packages`. The major advantage of this approach is that existing code continues working just as before, while new code can easily take advantage of the `try-with-resources` statement.

Let us consider the following example:

```
public class AutoClose implements AutoCloseable {

       @Override
       public void close() {
           System.out.println(">>> close()");
           throw new RuntimeException("Exception in close()");
       }

       public void work() throws MyException {
           System.out.println(">>> work()");
           throw new MyException("Exception in work()");
       }

       public static void main(String[] args) {
           try (AutoClose autoClose = new AutoClose()) {
               autoClose.work();
           } catch (MyException e) {
               e.printStackTrace();
           }
       }
   }
class MyException extends Exception {

       public MyException() {
           super();
       }

       public MyException(String message) {
           super(message);
       }
   }
```

Copy

The `AutoClose` class implements `AutoCloseable` and can thus be used as part of a try-with-resources statement, as illustrated in the `main()` method. Intentionally, we added some console output, and we throw exceptions both in the `work()` and `close()` methods of the class. Running the program yields the following output:

```
>>> work()
   >>> close()
   MyException: Exception in work()
          at AutoClose.work(AutoClose.java:11)
          at AutoClose.main(AutoClose.java:16)
          Suppressed: java.lang.RuntimeException: Exception in close()
                 at AutoClose.close(AutoClose.java:6)
                 at AutoClose.main(AutoClose.java:17)
```

The output clearly proves that `close()` was indeed called before entering the `catch` block that should handle the exception. Yet, the Java developer discovering Java SE 7 might be surprised to see the exception stack trace line prefixed by “`Suppressed: (…)`”. It matches the exception thrown by the `close()` method, but you could never encounter such a form of stack trace prior to Java SE 7. What is going on here?

Exception Masking

To understand what happened in the previous example, let us get rid of the `try-with-resources` statement for a moment, and manually rewrite a correct resource management code. First, let us extract the following static method to be invoked by the `main` method:

```
public static void runWithMasking() throws MyException {
       AutoClose autoClose = new AutoClose();
       try {
           autoClose.work();
       } finally {
           autoClose.close();
       }
   }
```

Copy

Then, let us transform the `main`

```
public static void main(String[] args) {
       try {
           runWithMasking();
       } catch (Throwable t) {
           t.printStackTrace();
       }
   }
```

Copy

Now, running the program gives the following output:

```
>>> work()
   >>> close()
   java.lang.RuntimeException: Exception in close()
          at AutoClose.close(AutoClose.java:6)
          at AutoClose.runWithMasking(AutoClose.java:19)
          at AutoClose.main(AutoClose.java:52)
```

This code, which is idiomatic for proper resource management prior to Java SE 7, shows the problem of one exception being masked by another exception. Indeed, the client code to the `runWithMasking()` method is notified of an exception being thrown in the `close()` method, while in reality, a first exception had been thrown in the `work()`

However, only one exception can be thrown at a time, meaning that even correct code misses information while handling exceptions. Developers lose significant time debugging when a main exception is masked by a further exception being thrown while closing a resource. The astute reader could question such claims, because exceptions can be nested, after all. However, nested exceptions should be used for causality between one exception and another, typically to wrap a low-level exception within one aimed at higher layers of an application architecture. A good example is a JDBC driver wrapping a socket exception into a JDBC connection. Here, there are really two exceptions: one in `work()` and one in `close()`, and there is absolutely no causality relationship between them.

Supporting "Suppressed" Exceptions

Because exception masking is such an important problem in practice, Java SE 7 extends exceptions so that “suppressed” exceptions can be attached to primary exceptions. What we called a “masked” exception previously is actually an exception to be suppressed and attached to a primary exception.

The extensions to `java.lang.Throwable` are as follows:

- `public final void addSuppressed(Throwable exception)` appends a suppressed exception to another one, so as to avoid exception masking.
- `public final Throwable[] getSuppressed()` gets the suppressed exceptions that were added to an exception.

These extensions were introduced especially for supporting the `try-with-resources`

Going back to the previous `runWithMasking()` method, let us rewrite it with support for suppressed exceptions in mind:

```
public static void runWithoutMasking() throws MyException {
        AutoClose autoClose = new AutoClose();
        MyException myException = null;
        try {
            autoClose.work();
        } catch (MyException e) {
            myException = e;
            throw e;
        } finally {
            if (myException != null) {
                try {
                    autoClose.close();
                } catch (Throwable t) {
                    myException.addSuppressed(t);
                }
            } else {
                autoClose.close();
            }
        }
    }
```

Copy

Clearly, this represents a fair amount of code just for properly handling two exception-throwing methods of a single auto-closeable class! A local variable is used to capture the primary exception, that is, the one that the `work()` method may throw. If such an exception is thrown, it is captured and then thrown again immediately, so as to delegate the remaining work to the `finally`

Entering the `finally` block, the reference to the primary exception is checked. If an exception was thrown, the exception that the `close()` method may throw would be attached to it as a suppressed exception. Otherwise, the `close()`

Let us run the modified program with this new method:

```
>>> work()
   >>> close()
   MyException: Exception in work()
          at AutoClose.work(AutoClose.java:11)
          at AutoClose.runWithoutMasking(AutoClose.java:27)
          at AutoClose.main(AutoClose.java:58)
          Suppressed: java.lang.RuntimeException: Exception in close()
                 at AutoClose.close(AutoClose.java:6)
                 at AutoClose.runWithoutMasking(AutoClose.java:34)
                 ... 1 more
```

As you can see, we manually reproduced the behavior of the `try-with-resources` statement earlier.

Syntantic Sugar Demystified

The `runWithoutMasking()` method that we implemented reproduces the behavior of the `try-with-resources` statement by properly closing resources and preventing exception masking. In reality, the Java compiler expands to code that matches the code of `runWithoutMasking()` for the following method, which uses a `try-with-resources` statement:

```
public static void runInARM() throws MyException {
       try (AutoClose autoClose = new AutoClose()) {
           autoClose.work();
       }
   }
```

Copy

This can be checked through decompilation. While we could compare the byte code using `javap`, which is part of the Java Development Kit (JDK) binary tools, let us use a byte code-to-Java source code decompiler instead. The code of `runInARM()` is extracted as follows by the `JD-GUI` tool (after reformatting):

```
public static void runInARM() throws MyException {
       AutoClose localAutoClose = new AutoClose();
       Object localObject1 = null;
       try {
           localAutoClose.work();
       } catch (Throwable localThrowable2) {
           localObject1 = localThrowable2;
```

```
                throw localThrowable2;
            } finally {
                if (localAutoClose != null) {
                    if (localObject1 != null) {
                        try {
                            localAutoClose.close();
                        } catch (Throwable localThrowable3) {
                            localObject1.addSuppressed(localThrowable3);
                        }
                    } else {
                        localAutoClose.close();
                    }
                }
            }
        }
```

Copy

As we can see, the code that we manually wrote shares the same resource management canvas as the one inferred by the compiler on a `try-with-resources` statement. It should also be noted that the compiler handles the case of possibly `null` resource references to avoid null pointer exceptions when invoking `close()` on a `null` reference by adding extra `if` statements in `finally` blocks to check whether a given resource is `null` or not. We did not do that in our manual implementation, because there is no chance that the resource is `null`. The compiler systematically generates such code, however.

Let us now consider another example, this time involving three resources:

```
private static void compress(String input, String output) throws IOException {
        try(
            FileInputStream fin = new FileInputStream(input);
            FileOutputStream fout = new FileOutputStream(output);
            GZIPOutputStream out = new GZIPOutputStream(fout)
        ) {
            byte[] buffer = new byte[4096];
            int nread = 0;
            while ((nread = fin.read(buffer)) != -1) {
                out.write(buffer, 0, nread);
            }
        }
    }
```

Copy

This method manipulates three resources to compress a file: one stream for reading, one stream for compressing, and one stream to an output file. This code is correct from a resource management perspective. Prior to Java SE 7, you would have had to write code similar to the code we obtained by decompiling the class containing this method, again with `JD-GUI`:

```
private static void compress(String paramString1, String paramString2)
            throws IOException {
        FileInputStream localFileInputStream = new FileInputStream(paramString1)
```

```
Object localObject1 = null;
   try {
       FileOutputStream localFileOutputStream = new FileOutputStream(paramS
    Object localObject2 = null;
       try {
           GZIPOutputStream localGZIPOutputStream = new GZIPOutputStream(lo
        Object localObject3 = null;
           try {
               byte[] arrayOfByte = new byte[4096];
               int i = 0;
               while ((i = localFileInputStream.read(arrayOfByte)) != -1) {
                   localGZIPOutputStream.write(arrayOfByte, 0, i);
               }
           } catch (Throwable localThrowable6) {
               localObject3 = localThrowable6;
               throw localThrowable6;
           } finally {
               if (localGZIPOutputStream != null) {
                   if (localObject3 != null) {
                       try {
                           localGZIPOutputStream.close();
                       } catch (Throwable localThrowable7) {
                           localObject3.addSuppressed(localThrowable7);
                       }
                   } else {
                       localGZIPOutputStream.close();
                   }
               }
           }
       } catch (Throwable localThrowable4) {
           localObject2 = localThrowable4;
           throw localThrowable4;
       } finally {
           if (localFileOutputStream != null) {
               if (localObject2 != null) {
                   try {
                       localFileOutputStream.close();
                   } catch (Throwable localThrowable8) {
                       localObject2.addSuppressed(localThrowable8);
                   }
               } else {
                   localFileOutputStream.close();
               }
           }
       }
   } catch (Throwable localThrowable2) {
       localObject1 = localThrowable2;
       throw localThrowable2;
   } finally {
       if (localFileInputStream != null) {
           if (localObject1 != null) {
               try {
                   localFileInputStream.close();
               } catch (Throwable localThrowable9) {
                   localObject1.addSuppressed(localThrowable9);
               }
           } else {
               localFileInputStream.close();
           }
       }
```

```
        }
    }
```

Copy

The benefits of the `try-with-resources` statement in Java SE 7 are self-explanatory for such an example: You have less code to write, the code is easier to read, and, last but not least, the code does not leak resources!

Discussion

The definition of the `close()` method in the `java.lang.AutoCloseable` interface mentions that a `java.lang.Exception` may be thrown. However, the previous `AutoClose` example declared this method without mentioning any checked exception, which we did on purpose, partly to illustrate exceptions masking.

The specification for auto-closeable classes suggests that throwing `java.lang.Exception` be avoided in favor of specific checked exceptions, and that no checked exception be mentioned if the `close()` method is not expected to fail. It also advises not to declare any exception that should not be suppressed, with `java.lang.InterruptedException` being the best example. Indeed, suppressing it and attaching it to another exception is likely to cause the thread interruption event to be ignored and put an application in an inconsistent state.

A legitimate question regarding the use of the `try-with-resources` statement is that of performance impact compared to manually crafting proper resource management code. There is actually no performance impact, because the compiler infers the minimal correct code for properly handling all exceptions, as we illustrated through decompilation in the previous examples.

At the end of the day, `try-with-resources` statements are syntactic sugar just like the enhanced `for` loops introduced in Java SE 5 for expanding loops over iterators.

That being said, we can limit the complexity of a `try-with-resources` statement expansion. Generally, the more a `try` block declares resources, the more complex the generated code will be. The previous `compress()` method could be rewritten with just two resources instead of three, generating less exception-handling blocks:

```
private static void compress(String input, String output) throws IOException {
        try(
            FileInputStream fin = new FileInputStream(input);
            GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(out
        ) {
            byte[] buffer = new byte[4096];
            int nread = 0;
            while ((nread = fin.read(buffer)) != -1) {
                out.write(buffer, 0, nread);
            }
        }
    }
```

Copy

As was the case before the advent of the `try-with-resources` statement in Java, a general rule of thumb is that developers should always understand the trade-offs when chaining resources instantiations. To do that, the best approach is to read the specifications of each resource's `close()` method to understand the semantics and implications.

Going back to the writingWithARM() example at the beginning of the article, chaining is safe because `DataOutputStream` is unlikely to throw an exception on `close()`. However, this is not the case with the last example, because `GZIPOutputStream` attempts to write the remaining compressed data as part of the `close()` method. If an exception was thrown earlier while writing the compressed file, the `close()` method in `GZIPOutputStream` is more than likely to throw a further exception as well, resulting in the `close()` method in `FileOutputStream` not being called and leaking a file descriptor resource.

A good practice is to have a separate declaration in a `try-with-resources` statement for each resource that holds critical system resources, such as a file descriptor, a socket, or a JDBC connection where you must make sure that a `close()` method is eventually called. Otherwise, provided that the related resources APIs permit this, chaining allocations is not just a convenience: It also yields more compact code while preventing resource leaks.

Conclusion

This article introduced a new language construct in Java SE 7 for the safe management of resources. This extension has more impact than being just yet more syntactic sugar. Indeed, it generates correct code on behalf of the developer, eliminating the need to write boilerplate code that is easy to get wrong. More importantly, this change has been accompanied with evolutions to attach one exception to another, thus providing an elegant solution to the well-known problem of exceptions masking each other.

See Also

Here are some additional resources:

- Java SE 7 Preview: http://jdk7.java.net/preview/
- Original proposal for Automatic Resource Management by Joshua Bloch, February 27, 2009: http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000011.html and //docs.google.com/View?id=ddv8ts74_3fs7483dp
- Project Coin: http://openjdk.java.net/projects/coin/
- JSR334 early draft preview: http://jcp.org/aboutJava/communityprocess/edr/jsr334/index.html
- JSR334 public review: http://jcp.org/aboutJava/communityprocess/pr/jsr334/index.html
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases* by Joshua Bloch and Neal Gafter (Addison-Wesley Professional, 2005)
- *Effective Java Programming Language Guide* by Joshua Bloch (Addison-Wesley Professional, 2001)
- The decorator design pattern: http://en.wikipedia.org/wiki/Decorator_pattern
- FindBugs, a static code analysis tool: http://findbugs.sourceforge.net/
- JD-GUI, a Java byte-code decompiler: http://java.decompiler.free.fr/?q=jdgui
- Python: http://www.python.org/
- Ruby: http://www.ruby-lang.org/

About the Author

Julien Ponge is a long-time open source craftsman. He created the IzPack installer framework and has participated in several other projects, including the GlassFish application server in cooperation with Sun

Microsystems. Holding a Ph.D. in computer science from UNSW Sydney and UBP Clermont-Ferrand, he is currently an associate professor in Computer Science and Engineering at INSA de Lyon and a researcher as part of the INRIA Amazones team. Speaking both industrial and academic languages, he is highly motivated in further developing synergies between those worlds.