JAVAWORLD JAVA

SEP 5, 2019 10:39 AM PDT

**About**
A beginner's library for learning about essential Java programming concepts, syntax, APIs, and packages.

ADVANCED JAVA LANGUAGE FEATURES

# Exceptions in Java, Part 2: Advanced features and types

**Java exception handling with stack traces, exception chaining, try-with-resources, final re-throw, and more**

**Listing 6. `StackTraceElementDemo.java` (version 2)**

```
public class StackTraceElementDemo
{
   public static void main(String[] args) throws NoStackTraceException
   {
      try
      {
         a();
      }
      catch (NoStackTraceException nste)
      {
         nste.printStackTrace();
      }
   }

   static void a() throws NoStackTraceException
   {
      b();
   }

   static void b() throws NoStackTraceException
   {
      throw new NoStackTraceException();
   }
}

class NoStackTraceException extends Exception
{
   @Override
   public synchronized Throwable fillInStackTrace()
   {
      setStackTrace(new StackTraceElement[]
                    {
                       new StackTraceElement("*StackTraceElementDemo*",
                                             "b()",
                                             "StackTraceElementDemo.java",
                                             22),
                       new StackTraceElement("*StackTraceElementDemo*",
                                             "a()",
                                             "StackTraceElementDemo.java",
                                             17),
                       new StackTraceElement("*StackTraceElementDemo*",
                                             "main()",
                                             "StackTraceElementDemo.java",
                                             7)
                    });
      return this;
```

```
    }
}
```

I've surrounded the `StackTraceElementDemo` class name with asterisks to prove that this stack trace is the one being output. Run the application and you'll observe the following stack trace:

```
NoStackTraceException
        at *StackTraceElementDemo*.b()(StackTraceElementDemo.java:22)
        at *StackTraceElementDemo*.a()(StackTraceElementDemo.java:17)
        at *StackTraceElementDemo*.main()(StackTraceElementDemo.java:7)
```

**Learning more about `StackTraceElement`**

To learn more about `StackTraceElement` and related methods, see Dustin Marx's blog post, "The Surprisingly Simple StackTraceElement."

## Causes and exception chaining

A `catch` block often responds to an exception by throwing another exception. The first exception is said to *cause* the second exception to occur. Furthermore, the exceptions are said to be implicitly *chained* together.

For example, a library method's `catch` block might be invoked with an *internal exception* that shouldn't be visible to the library method's caller. Because the caller needs to be notified that something went wrong, the `catch` block creates an *external exception* that conforms to the library method's contract interface, and which the caller can handle.

Because the internal exception might be helpful in diagnosing the problem, the `catch` block should be able to wrap the internal exception in the external exception. The wrapped exception is known as a *cause* because its existence causes the external exception to be thrown. Also, the wrapped internal exception (cause) is explicitly *chained* to the external exception.

JDK 1.4 introduced support for causes and exception chaining via two `Throwable` constructors (along with `Exception`, `RuntimeException`, and `Error` counterparts) and a pair of methods:

- `Throwable(String message, Throwable cause)`
- `Throwable(Throwable cause)`
- `Throwable getCause()`
- `Throwable initCause(Throwable cause)`

The `Throwable` constructors (and their subclass counterparts) let you wrap the cause while constructing a throwable. If you're dealing with legacy code whose custom exception classes don't support either constructor, you can wrap the cause by invoking `initCause()` on the throwable. Note that `initCause()` can be called just one time. Either way, you can return the cause by invoking `getCause()`. This method returns `null` when there is no cause.

Listing 7 presents a `CauseDemo` application that demonstrates causes (and exception chaining).

**Listing 7. `CauseDemo.java`**

```
public class CauseDemo
{
   public static void main(String[] args)
   {
      try
      {
         Library.externalOp();
      }
      catch (ExternalException ee)
      {
         ee.printStackTrace();
      }
   }
}

class Library
{
   static void externalOp() throws ExternalException
   {
      try
      {
         throw new InternalException();
      }
      catch (InternalException ie)
      {
         throw (ExternalException) new ExternalException().initCause(ie);
      }
   }

   private static class InternalException extends Exception
   {
   }
}

class ExternalException extends Exception
{
}
```

Listing 7 reveals `CauseDemo`, `Library`, and `ExternalException` classes. `CauseDemo`'s `main()` method invokes `Library`'s `externalOp()` method and catches its thrown `ExternalException` object. The `catch` block invokes `printStackTrace()` to output the external exception and its cause.

Library's externalOp() method deliberately throws an InternalException object, which its catch block maps to an ExternalException object. Because ExternalException doesn't support a constructor that can take a cause argument, initCause() is used to wrap the InternalException object.

Run this application and you'll observe the following stack traces:

```
ExternalException
        at Library.externalOp(CauseDemo.java:26)
        at CauseDemo.main(CauseDemo.java:7)
Caused by: Library$InternalException
        at Library.externalOp(CauseDemo.java:22)
        ... 1 more
```

The first stack trace shows that the external exception originated in Library's externalOp() method (line 26 in CauseDemo.java) and was thrown out of the call to this method on line 7. The second stack trace shows that the internal exception cause originated in Library's externalOp() method (line 22 in CauseDemo.java). The `... 1 more` line refers to the first stack trace's final line. If you could remove this line, you'd observe the following output:

```
ExternalException
        at Library.externalOp(CauseDemo.java:26)
        at CauseDemo.main(CauseDemo.java:7)
Caused by: Library$InternalException
        at Library.externalOp(CauseDemo.java:22)
        at CauseDemo.main(CauseDemo.java:7)
```

You can prove that the second trace's final line is a duplicate of the first stack trace's final line by changing ee.printStackTrace(); to ee.getCause().printStackTrace();.

## More about ...more and interrogating a chain of causes

In general, a `... n more` line indicates that the last *n* lines of the cause's stack trace are duplicates of the last *n* lines of the previous stack trace.

My example revealed only one cause. Exceptions thrown from non-trivial real-world applications may contain extensive chains of many causes. You can access these causes by employing a loop such as the following:

```
catch (Exception exc)
{
   Throwable t = exc.getCause();
   while (t != null)
   {
      System.out.println(t);
      t = t.getCause();
   }
}
```

## Managing resources with try-with-resources

Java applications often access files, database connections, sockets, and other resources that depend on related system resources (e.g., file handles). The scarcity of system resources implies that they must eventually be released, even when an exception occurs. When system resources aren't released, the application eventually fails when attempting to acquire other resources, because no more related system resources are available.

In Part 1, I mentioned that resources (actually, the system resources on which they depend) are released in a `finally` block. This can lead to tedious boilerplate code, such as the file-closing code that appears below:

```
finally
{
   if (fis != null)
      try
      {
         fis.close();
      }
      catch (IOException ioe)
      {
         // ignore exception
      }
   if (fos != null)
      try
      {
         fos.close();
      }
      catch (IOException ioe)
      {
         // ignore exception
      }
}
```

Not only does this boilerplate code add bulk to a classfile, the tedium in writing it might lead to a bug, perhaps even failing to close a file. JDK 7 introduced `try`-with-resources to overcome this problem.

## The basics of try-with-resources

The `try`-with-resources construct automatically closes open resources when execution leaves the scope in which they were opened and used, whether or not an exception is thrown from that scope. This construct has the following syntax:

```
try (resource acquisitions)
{
   // resource usage
}
```

The `try` keyword is parameterized by a semicolon-separated list of resource-acquisition statements, where each statement acquires a resource. Each acquired resource is available to the body of the `try` block, and is automatically closed when execution leaves this body. Unlike a regular `try` statement, `try`-with-resources doesn't require `catch` blocks and/or a `finally` block to follow `try()`, although they can be specified.

Consider the following file-oriented example:

```
try (FileInputStream fis = new FileInputStream("abc.txt"))
{
   // Do something with fis and the underlying file resource.
}
```

In this example, an input stream to an underlying file resource (`abc.txt`) is acquired. The `try` block does something with this resource, and the stream (and file) is closed upon exit from the `try` block.

### Using var with `try`-with-resources

JDK 10 introduced support for `var`, an identifier with special meaning (i.e., not a keyword). You can use `var` with `try`-with-resources to reduce boilerplate. For example, you could simplify the previous example to the following:

```
try (var fis = new FileInputStream("abc.txt"))
{
   // Do something with fis and the underlying file resource.
}
```

## Copying a file in a try-with-resources context

In Part 1, I excerpted the `copy()` method from a file-copy application. This method's `finally` block contains the file-closing boilerplate presented earlier. Listing 8 improves this method by using `try`-with-resources to handle the cleanup.

### Listing 8. `Copy.java`

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java Copy srcfile dstfile");
         return;
      }

      try
      {
         copy(args[0], args[1]);
      }
      catch (IOException ioe)
      {
         System.err.println("I/O error: " + ioe.getMessage());
      }
   }

   static void copy(String srcFile, String dstFile) throws IOException
   {
      try (FileInputStream fis = new FileInputStream(srcFile);
           FileOutputStream fos = new FileOutputStream(dstFile))
      {
         int c;
         while ((c = fis.read()) != -1)
            fos.write(c);
      }
   }
}
```

`copy()` uses `try`-with-resources to manage source and destination file resources. The round bracketed-code following `try` attempts to create file input and output streams to these files. Assuming success, its body executes, copying the source file to the destination file.

Whether an exception is thrown or not, `try`-with-resources ensures that both files are closed when execution leaves the `try` block. Because the boilerplate file-closing code that was shown earlier isn't needed, Listing 8's `copy()` method is much simpler and easier to read.

## Designing resource classes to support try-with-resources

The `try`-with-resources construct requires that a resource class implement the `java.lang.Closeable` interface or the JDK 7-introduced `java.lang.AutoCloseable` superinterface. Pre-Java 7 classes like `java.io.FileInputStream` implemented `Closeable`, which offers a `void close()` method that throws `IOException` or a subclass.

Starting with Java 7, classes can implement `AutoCloseable`, whose single `void close()` method can throw `java.lang.Exception` or a subclass. The `throws` clause has been expanded to accommodate situations where you might need to add `close()` methods that can throw exceptions outside of the `IOException` hierarchy; for example, `java.sql.SQLException`.

Listing 9 presents a `CustomARM` application that shows you how to configure a custom resource class so that you can use it in a `try`-with-resources context.

**Listing 9. `CustomARM.java`**

```java
public class CustomARM
{
   public static void main(String[] args)
   {
      try (USBPort usbp = new USBPort())
      {
         System.out.println(usbp.getID());
      }
      catch (USBException usbe)
      {
         System.err.println(usbe.getMessage());
      }
   }
}

class USBPort implements AutoCloseable
{
   USBPort() throws USBException
   {
      if (Math.random() < 0.5)
         throw new USBException("unable to open port");
      System.out.println("port open");
   }

   @Override
   public void close()
   {
      System.out.println("port close");
   }

   String getID()
   {
      return "some ID";
   }
}

class USBException extends Exception
{
   USBException(String msg)
   {
      super(msg);
   }
}
```

Listing 9 simulates a USB port in which you can open and close the port and return the port's ID. I've employed `Math.random()` in the constructor so that you can observe `try`-with-resources when an exception is thrown or not thrown.

Compile this listing and run the application. If the port is open, you'll see the following output:

```
port open
some ID
port close
```

If the port is closed, you might see this:

```
unable to open port
```

## Suppressing exceptions in try-with-resources

If you've had some programming experience, you might have noticed a potential problem with `try`-with-resources: Suppose the `try` block throws an exception. This construct responds by invoking a resource object's `close()` method to close the resource. However, the `close()` method might also throw an exception.

When `close()` throws an exception (e.g., `FileInputStream`'s `void close()` method can throw `IOException`), this exception masks or hides the original exception. It seems that the original exception is lost.

In fact, this isn't the case: `try`-with-resources suppresses `close()`'s exception. It also adds the exception to the original exception's array of suppressed exceptions by invoking `java.lang.Throwable`'s `void addSuppressed(Throwable exception)` method.

Listing 10 presents a SupExDemo application that demonstrates how to repress an exception in a `try`-with-resources context.

**Listing 10. `SupExDemo.java`**

```
import java.io.Closeable;
import java.io.IOException;

public class SupExDemo implements Closeable
{
   @Override
   public void close() throws IOException
   {
      System.out.println("close() invoked");
      throw new IOException("I/O error in close()");
   }

   public void doWork() throws IOException
   {
      System.out.println("doWork() invoked");
      throw new IOException("I/O error in work()");
   }

   public static void main(String[] args) throws IOException
   {
      try (SupExDemo supexDemo = new SupExDemo())
      {
         supexDemo.doWork();
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
         System.out.println();
         System.out.println(ioe.getSuppressed()[0]);
      }
   }
}
```

Listing 10's `doWork()` method throws an `IOException` to simulate some kind of I/O error. The `close()` method also throws the `IOException`, which is suppressed so that it doesn't mask `doWork()`'s exception.