JAVAWORLD

JAVA 101: LEARN JAVA
By Jeff Friesen, JavaWorld
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

Page 3 of 3

The `catch` block accesses the suppressed exception (thrown from `close()`) by invoking `Throwable`'s `Throwable[] getSuppressed()` method, which returns an array of suppressed exceptions. Only the first element is accessed because only one exception is suppressed.

Compile Listing 10 and run the application. You should observe the following output:

```
doWork() invoked
close() invoked
java.io.IOException: I/O error in work()
        at SupExDemo.doWork(SupExDemo.java:16)
        at SupExDemo.main(SupExDemo.java:23)
        Suppressed: java.io.IOException: I/O error in close()
                at SupExDemo.close(SupExDemo.java:10)
                at SupExDemo.main(SupExDemo.java:24)

java.io.IOException: I/O error in close()
```

Exception handling in JDK 7: Multi-catch

JDK 7 makes it possible to codify a single `catch` block that catches more than one type of exception. The purpose of this *multi-catch* feature is to reduce code duplication and reduce the temptation to catch overly broad exceptions (for instance, `catch (Exception e)`).

Suppose you've developed an application that gives you the flexibility to copy data to a database or file. Listing 11 presents a `CopyToDatabaseOrFile` class that simulates this situation, and demonstrates the problem with `catch` block code duplication.

Listing 11. `CopyToDatabaseOrFile.java`

```
import java.io.IOException;

import java.sql.SQLException;

public class CopyToDatabaseOrFile
{
   public static void main(String[] args)
   {
      try
      {
         copy();
      }
      catch (IOException ioe)
      {
         System.out.println(ioe.getMessage());
         // additional handler code
      }
      catch (SQLException sqle)
      {
         System.out.println(sqle.getMessage());
         // additional handler code that's identical to the previous handler's
         // code
      }
   }

   static void copy() throws IOException, SQLException
   {
      if (Math.random() < 0.5)
         throw new IOException("cannot copy to file");
      else
         throw new SQLException("cannot copy to database");
   }
}
```

JDK 7 overcomes this code duplication problem by letting you specify multiple exception types in a `catch` block where each successive type is separated from its predecessor by placing a vertical bar (|) between these types:

```
try
{
   copy();
}
catch (IOException | SQLException iosqle)
{
   System.out.println(iosqle.getMessage());
}
```

Now, when `copy()` throws either exception, the exception will be caught and handled by the `catch` block.

When multiple exception types are listed in a `catch` block's header, the parameter is implicitly regarded as `final`. As a result, you cannot change the parameter's value. For example, you cannot change the reference stored in the previous code fragment's `iosqle` parameter.

Shrinking bytecode

The bytecode resulting from compiling a `catch` block that handles multiple exception types will be smaller than compiling several `catch` blocks that each handle only one of the listed exception types. A `catch` block that handles multiple exception types contributes no duplicate bytecode during compilation. In other words, the bytecode doesn't contain replicated exception handlers.

Final re-throw

Starting with JDK 7, the Java compiler analyzes re-thrown exceptions more precisely than its predecessors, but only when no assignments are made to a re-thrown exception's `catch` block parameter, which is considered to be effectively `final`. When a preceding `try` block throws an exception that's a supertype/subtype of the parameter's type, the compiler throws the caught exception's actual type instead of throwing the parameter's type (as is done in previous Java versions).

The purpose of this *final re-throw* feature is to facilitate adding a `try-catch` statement around a block of code to intercept, process, and re-throw an exception without affecting the statically determined set of exceptions thrown from the code. Also, this feature lets you provide a common exception handler to partly handle the exception close to where it's thrown, and provide more precise handlers elsewhere that handle the re-thrown exception. Consider Listing 12.

Listing 12. `MonitorEngine.java`

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

class TemperatureException extends Exception
{
   TemperatureException(String msg)
   {
      super(msg);
   }
}

public class MonitorEngine
{
   public static void main(String[] args)
   {
      try
      {
         monitor();
      }
      catch (Exception e)
      {
         if (e instanceof PressureException)
            System.out.println("correcting pressure problem");
         else
            System.out.println("correcting temperature problem");
      }
   }

   static void monitor() throws Exception
   {
      try
      {
         if (Math.random() < 0.1)
            throw new PressureException("pressure too high");
         else
         if (Math.random() > 0.9)
            throw new TemperatureException("temperature too high");
         else
            System.out.println("all is well");
      }
      catch (Exception e)
      {
         System.out.println(e.getMessage());
         throw e;
      }
   }
}
```

Listing 12 simulates the testing of an experimental rocket engine to see if the engine's pressure or temperature exceeds a safety threshold. It performs this testing via the `monitor()` helper method.

`monitor()`'s `try` block throws `PressureException` when it detects a pressure extreme, and throws `TemperatureException` when it detects a temperature extreme. (Because this is only a simulation, random numbers are used -- the `java.lang.Math` class's `static double random()` method returns a random number between 0.0 and (almost) 1.0.) The `try` block is followed by a `catch` block designed to partly handle the exception by outputting a warning message. This exception is then re-thrown so that `monitor()`'s calling method can finish handling the exception.

Before JDK 7 you couldn't specify `PressureException` and `TemperatureException` in `monitor()`'s `throws` clause because the `catch` block's `e` parameter is of type `java.lang.Exception` and re-throwing an exception was treated as throwing the parameter's type. JDK 7 and successor JDKs have made it possible to specify these exception types in the `throws` clause because their compilers can determine that the exception thrown by `throw e` comes from the `try` block, and only `PressureException` and `TemperatureException` can be thrown from this block.

Because you can now specify `static void monitor() throws PressureException, TemperatureException`, you can provide more precise handlers where `monitor()` is called, as the following code fragment demonstrates:

```
try
{
   monitor();
}
catch (PressureException pe)
{
   System.out.println("correcting pressure problem");
}
catch (TemperatureException te)
{
   System.out.println("correcting temperature problem");
}
```

Because of the improved type checking offered by final re-throw, source code that compiled under previous versions of Java might fail to compile under JDK 7 and later JDKs. For example, consider Listing 13.

Listing 13. `BreakageDemo.java`

```
class SuperException extends Exception
{
}

class SubException1 extends SuperException
{
}

class SubException2 extends SuperException
{
}

public class BreakageDemo
{
   public static void main(String[] args) throws SuperException
   {
      try
      {
         throw new SubException1();
      }
      catch (SuperException se)
      {
         try
         {
            throw se;
         }
         catch (SubException2 se2)
         {
         }
      }
   }
}
```

Listing 13 compiles under JDK 6 and earlier. However, it fails to compile under JDK 7 and successor JDKs, whose compilers detect and report the fact that `SubException2` is never thrown in the body of the corresponding `try` statement. This is a small problem that you are unlikely to encounter in your programs, and a worthwhile trade-off for having the compiler detect a source of redundant code. Removing redundancies results in cleaner code and smaller classfiles.

Exception handling in JDK 9: Stack walking

Obtaining a stack trace via `Thread`'s or `Throwable`'s `getStackTrace()` method is costly and impacts performance. The JVM eagerly captures a snapshot of the entire stack (except for hidden stack frames), even when you only need the first few frames. Also, your code will probably have to process frames that are of no interest, which is also time-consuming. Finally, you cannot access the actual `java.lang.Class` instance of the class that declared the method represented by a stack frame. To access this `Class` object, you're forced to extend `java.lang.SecurityManager` to access the `protected` `getClassContext()` method, which returns the current execution stack as an array of `Class` objects.

`Thread::getStackTrace` might return a partial stack trace

According to JEP 259, a JVM implementation can omit some stack frames to improve performance. For example, `Thread::getStackTrace` might return a partial stack trace, which isn't useful when all stack frames are desired.

JDK 9 introduced the `java.lang.StackWalker` class (with its nested `Option` class and `StackFrame` interface) as a more performant and capable alternative to `StackTraceElement` (plus `SecurityManager`). To learn about `StackWalker` and its related types, see my **Java Q&A** blog post "Java 9's other new enhancements, Part 5: Stack-Walking API." This post includes mention of topics not yet covered in *Java 101* such as generic methods.

In conclusion

This article completes the **Java 101** two-part series on Java's exception framework. You might want to reinforce your understanding of various aspects of this framework by reviewing Oracle's Exceptions lesson in the Java Tutorials. This lesson includes a few questions and exercises that will help you go further in mastering Java exceptions.

Jeff Friesen teaches Java technology (including Android) to everyone.

Follow