TheServerSide.com

New Java 7 Features: How to Use a More Precise Rethrow in Exceptions from Project Coin

Aficionados of other languages such as Scala or Clojure are always slagging the fact that Java is too verbose and cumbersome. So, it's not too surprising to find out that Java 7 introduced a number of new facilities that can help to trim down the code an application developer is required to write.

Take a look at the following piece of code, focussing on the redundancy of the catch blocks while pontificating on how redundant they are.

```
class OpenException extends Exception {}
class CloseException extends Exception {}

public class PreciseRethrows {

        public static void main(String args[]) throws OpenException, CloseException {
                boolean flag = true;
                try {
                        if (flag){
                                throw new OpenException();
                        }
                        else {
                                throw new CloseException();
                        }
                } catch(OpenException oe) {
                        System.out.println(oe.getMessage());
                        throw oe;
                }
                catch (CloseException ce) {
                        System.out.println(ce.getMessage());
                        throw ce;
                }
        }
}
```

This is just an example where two exceptions are being handled, and for the most part, each exception is being handled the same way, with the message associated with the error being logged to the console, and the exception being subsequently re thrown. This code is annoying enough, with two catch blocks, but just imagine if there were five or ten? It's certainly understandable how developers from other languages can peek at Java's verbose exception handling and roll their eyes.

In versions prior to Java 7, one approach was to simply catch the generic Exception, as you can see in the following code snippet. This has the benefit of eliminating duplicate code, but the drawback is that the method itself no longer throws the specific OpenException or CloseException, but instead, the very generic Exception. The code is less verbose, but client applications no longer have the benefit of easily

dealing with the specific CloseException or OpenException when they are thrown.

```
public static void main(String args[]) throws Exception {
        boolean flag = true;
        try {
                if (flag){
                        throw new OpenException();
                }
                else {
                        throw new CloseException();
                }
        }
        catch (Exception e) {
                System.out.println(e.getMessage());
                throw e;
        }
}
```

Addressing the drawback of losing precision with regards to declaring the various Exceptions that a method might throw, Java 7 allows for a list of more specific or precise exceptions to be listed on the method signature.

For example, in Java 6, the following code snippet will generate the following compiler error: Unhandled exception type Exception

The Eclipse IDE then provides the following suggestion: "Add throws Declaration."

```
public static void main(String args[]) throws OpenException, CloseException {
        boolean flag = true;
        try {
                if (flag){
                        throw new OpenException();
                }
                else {
                        throw new CloseException();
                }
        }
        catch (Exception e) {
                System.out.println(e.getMessage());
                throw e;
        }
}
```

The problem here is that the method is catching the generic java.lang.Exception, but indicating that it only throws the very specific exceptions OpenException and CloseException. In Java 6, you would need to add java.lang.Exception to the method signature, but in Java 7, you do not.

Now one thing you should avoid is reassigning the exception in the catch block: if you do, the benefits of Java 7's precise rethrows feature goes out the window. So, if you were to do something like this:

```
        catch (Exception e) {
                System.out.println(e.getMessage());
                e = new OpenException();
                throw e;
```

```
        }
```

You would get an error that says something like this: Unhandled exception type Exception. And the only easy way to get rid of the exception is to additionally throw java.lang.Exception from the method, which is exactly what we were trying to avoid in the first place.

```
public static void main(String args[]) throws OpenException, CloseException, Exception
```

Basically, you can list specific exceptions in the throws clause of your method, even if they are not explicitly handled by a catch block if:

The try block actually throws that specific exception at some point in time.
The specific Exception isn't already handled at any point by a preceding catch block
The exception named in the throws clause of the method signature must be on the class hierarchy of at least one exception being handled and rethrown by a catch block (subtype or supertype)

In Java 7, the compiler will look at a method and figure out what the most specific exception is that might get thrown, as opposed to simply looking at the Exception which is the most general. The Java 7 compiler will then allow you to list the specialized exceptions that might get thrown from the method. In this case, our method that only ever throws the very generic java.lang.Exception from the catch block will be allowed to list the specific OpenException and CloseException in the method signature, because the JVM realizes that in fact, even though the exception being thrown from the catch block is caught in the very general Exception for, the Java 7 JVM also realizes that in reality, this exception has to be an instance of either an OpenException or CloseException.

Check out these other tutorials from TheServerSide's Sal Pece and Cameron McKenzie covering the new Java 7 features:

New Java 7 Features: Binary Notation and Literal Variable Initialization
New Java 7 Features: Numeric Underscores with Literals Tutorial
New Java 7 Features: Using String in the Switch Statement Tutorial
New Java 7 Features: The Try-with-resources Language Enhancement Tutorial
New Java 7 Features: Automatic Resource Management (ARM) and the AutoCloseable Interfact Tutorial
New Java 7 Features: Suppressed Exceptions and Try-with-resources Tutorial
Java 7 Mock Certification Exam: A Tricky OCPJP Question about ARM and Try-with-Resources
OCAJP Exam to Debuts in March 2010. OCPJP Released in June?
OCPJP & OCAJP Java 7 Exams: Oracle Drops the Training Requirement
OCAJP and OCPJP Changes for Java 7: New Objectives, a Format Change and a Price Hike

12 Jan 2012