TheServerSide.com

New Java 7 Features: Automatic Resource Management (ARM) and the AutoCloseable Interface Tutorial

The whole idea of the new Java 7 *try-with-resources* syntax is to improve the track record of Java professionals with regards to properly cleaning up application resources when they are no longer required in the code. Since the majority of these 'clean-up' problems occur when the call to a close method is skipped due to a complication that gets introduced during the exception handling phase, it's important to take a good look at how try-with-resource blocks work when exceptions are thrown into the mix.

Just a few exceptions...

I'm going to add the following three exceptions to the try-with-resources example we started with in the previous tutorial (we reproduce the code from the previous tutorial below):

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

For the sake of simplicity, this tutorial will focus on a single component, the OpenDoor class.

The OpenDoor class

In this iteration of the OpenDoor class, each method will have the capacity to throw an exception. The OpenDoor constructor might throw the OpenException, the swing() method might throw the SwingException, and the close() method might throw the CloseException. I say 'might', because to begin with, each throw call is commented out, as you can see in the following code:

```
class OpenDoor implements AutoCloseable {

  public OpenDoor() throws Exception {
    System.out.println("The door is open.");
    //throw new OpenException()
  };

  public void swing() throws Exception {
    System.out.println("The door is becoming unhinged.");
    //throw new SwingException();
  }

  public void close() throws Exception {

    System.out.println("The door is closed.");
    // throw new CloseException();
  }
```

```
}
```

The TryWithResources runnable class

And here's the runnable TryWithResources class. References to the OpenWindow class from the previous tutorial have been removed. Notice the additional call to *e.getClass()* in the catch block.

```
public class TryWithResources {

        public static void main(String[] args) throws Exception {

                try ( OpenDoor door = new OpenDoor() ) {
                        door.swing();
                }
                catch (Exception e) {
                        System.out.println("Is there a draft? " + e.getClass());
                }
                finally {
                        System.out.println("I'm putting a sweater on, regardless. ");
                }
        }
}
```

When the unadultered code runs, that is, when no exceptions are thrown, the output is as follows:

```
The door is open.
The door is becoming unhinged.
The door is closed.
I'm putting a sweater on, regardless.
```

Basically, the door is opened as the constructor is called, someone swings the door, the door gets automatically closed by the magical try-with-resources code block, and the finally clause runs indicating that someone is putting on a sweater, because finally clauses *always* run, and besides, it's never a bad idea to put on a sweater when it's drafty.

Of course, we have three exceptions that are currently commented out in the code. Let's throw them one at a time.

Exceptions in a constructor in a try-with-resources initialization block

What happens when the constructor of the OpenDoor class throws an exception? Will the close() method still get automatically invoked by the try-with-resources block? Let's find out by removing the comments from the throw clause in the constructor:

```
public OpenDoor() throws Exception {
        System.out.println("The door is open.");
        throw new OpenException();
}
```

When we run the code, we get the following output:

```
The door is open.
Is there a draft? class OpenException
I'm putting a sweater on, regardless.
```

As you can see, the body of the try-with-resources block is not invoked on an object that throws an Exception in the resource declaration phase. When an exception is thrown in the resource declaration phase, the object that throws the exception is not considered to have been initialized properly, and as a result, the close method is not invoked on it. Note however, if any objects *were* properly intialized before the Constructor in question threw an Exception, their close methods would indeed be invoked in an order reverse to the order in which they were intialized.

Exceptions in the body of a try-with-resources block

Instead of throwing an Exception in the OpenDoor constructor, what would happen if we threw one only in the swing method, which would cause an exception to happen in the body of the try, as opposed in the initialization block:

```
public void swing() throws Exception {
        System.out.println("The door is becoming unhinged.");
        throw new SwingException();
}
```

In this case, we see the following output:

```
The door is open.
The door is becoming unhinged.
The door is closed.
Is there a draft? class SwingException
I'm putting a sweater on, regardless.
```

Looking at the code, we see that:
1. The OpenDoor constructor is invoked, triggering "The door is open." to output to the console.
2. The door is swung, causing "The door is becoming unhinged" to be written to the console.
3. A SwingException is thrown
4. The close() method is invoked, causing "The door is closed." to be output to the console.
5. The exception is handled, generating the following console output: Is there a draft? class SwingException
6. The finally block is executed, outputting: "I'm putting a sweater on, regardless."

This is the standard, expected behavior of a try-with-resources code block, although it is easy to get confused as to when the close() method will be invoked. The rule is that any AutoCloseable objects will have their close() methods invoked after the try method has executed, but before any exception handling surrounding the try block occurs.

Exceptions in a close() method of an AutoCloseable resource

Now, what happens when an exception occurs in the close() method of an AutoCloseable resource? Well, by commenting out all of the other pieces of code that throw an Exception, and uncomment the throwing of the CloseException in the close() method, as so:

```
public void close() throws Exception {
   System.out.println("The door is closed.");
   throw new CloseException();
}
```

we get the following output:

```
The door is open.
The door is becoming unhinged.
The door is closed.
Is there a draft? class CloseException
I'm putting a sweater on, regardless.
```

Exceptions with multiple resources in try-with-resources blocks

So, we've seen how the try-with-resources block works when there's only one resource that's involved, and we've seen the way the flow of control moves through a try-with-resource block when a resource throws an exception, either in the body of the block, the resource declaration, or the close() method. But what happens when there are two resources involved?

Going back to basics, we know that we can initialize multiple resources in the resource declaration block. And we know that for each AutoCloseable resource that's properly initialized in the resource declaration block, there will be a corresponding call to that object's close() method when the try block is completed.

But what happens if there are two objects that are initialized in the resource declaration, and one of them throws an exception *during* creation? The rule is that if any resources are initialized in the resource declaration block before an exception is thrown, the close method will be invoked on those objects, but not on the object that threw the Exception during initialization.

So, imagine we added in a simple OpenWindow class:

```
class OpenWindow implements AutoCloseable {
        public OpenWindow() {
                System.out.println("The window is open.");
        }

        public void close() throws Exception {
                System.out.println("The window is closed.");
        }
}
```

And then edited the main method to initialize an OpenWindow, which will initialize properly, before initializing the OpenDoor, which throws an OpenException. In this case, the close will be invoked on the properly initialized OpenWindow instance, but

not on the OpenDoor instance, giving the following output:

```
The window is open.
The door is open.
The window is closed.
Is there a draft? class OpenException
I'm putting a sweater on, regardless.
```

Notice how the Window gets closed, but the door does not. As you can see, any objects that are successfully initialized in the resource declaration get closed, while instances that throw an exception during resource initialization and trigger the exception handling process do not have their close method invoked.

When the program actually runs, we see the following output:

The window is open.
The door is open.
The window is closed.
Is there a draft? class OpenException
I'm putting a sweater on, regardless.

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
OCAJP7 Exam to Debuts in March 2012. OCPJP7 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