Exceptions-Part 2

Chapter 7, Core Java Volume I

Contents

Catch or Declare Rule

- *Declare or Catch Rule*: if any checked exception may occur, then the method catches it or declare it.

```
public void read(String filename) throws IOException
{
   InputStream in = new FileInputStream(filename);
   int b;
   while ((b = in.read()) != -1)
   {
         // process input
    }
} // propagate the exception to caller
```

```
//From Java.io package:
public FileInputStream (String name)
      throws FileNotFoundException {  … }
```

```
public void read(String filename) {
 try  {
     InputStream in =
          new FileInputStream(filename);
     int b;
     while ((b = in.read()) != -1)
     {
          // process input
     }
 }
 catch (IOException exception) {
     exception.printStackTrace();
 }  // handle the exception here
}
```

Catch or Declare Rule

- Exception: Sometimes you need to catch an exception when you override a method that is declared to throw no exceptions. Do not add more `throws` specifiers to a subclass method than those of the superclass method.

Throwing Exceptions

- You can throw an exception when something terrible happens in your code.
  - Example: you read a header that promised Content-length: 1024, but you got an end of file after 733 characters.
- Find an exception type to throw.
  - The Java library has an `EOFException` with description: “Signals that an EOF has been reached unexpectedly during input.”
- Construct an object and throw it:

```
throw new EOFException();
```

- Or better, provide a reason:

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

- If you throw a checked exception, you have to specify the exception in the method.

Throwing Exceptions

```
String readData(Scanner in) throws EOFException
{    …
     while (. . .)
     {
          if (!in.hasNext()) // EOF encountered
          {
               if (n < len)
               {
                    String gripe = "Content-length: " + len + ", Received: " + n; // to know the reason
                    throw new EOFException(gripe);  // EOFException is a checked exception
               }
          } //
          …
    }
   return s; // return only character value because it will not return an error code like C
}
```

Defining Exception Classes

- What if your situation isn't covered by an exception in the standard library?
- Create your own exception class.
- Derive it from Exception, RuntimeException, or preferably a more specific exception class:

```
class FileFormatException extends IOException
{
   public FileFormatException() {}
   public FileFormatException(String gripe)
   {
      super(gripe);
   }
}
```

- Then you can throw an object of your own exception type:

```
if (n < len) throw new FileFormatException();
```

Rethrowing and Chaining Exceptions

- Sometimes you want to catch an exception and rethrow it as a different type:

```
try
{
   access the database
}
catch (SQLException e)
{
   // do something (such as logging)
   throw new ServletException("database error: " + e.getMessage());
}
```

- Better: Set the original exception as the cause. ...

```
catch (SQLException e)
{ // do something
   var se = new ServletException("database error");
   se.initCause(e);
   throw se;
}
```

- The cause can later be retrieved with the *getCause* method.

```
Throwable original = se.getCause(); // Throwbale is superclass
```

The finally Clause

- Suppose your code accesses a resource that needs to be relinquished:

```
PrintWriter out = new PrintWriter(...);
. . . // an exception happens here
out.close();
```

- What if the . . . code throws an exception?
  - The out.close() statement is never executed!
- Remedy: Put it in a finally clause:

```
FileOutputStream out = new FileOutputStream(...);
try
{
   . . .
}
catch blocks here
finally
{
   out.close();
}
```

```
try
{
  FileOutputStream out = new FileOutputStream(...);
  try
  {
     . . .
  }
  finally
  {
    out.close();  // IOException might happen
  }
}
catch blocks here
```

Execution Scenarios

```
m() throws IOException
 {
  OutputStream out = new FileOutputStream(...);
  try
   {
     // 1
    code that might throw exceptions
     // 2
   }
   catch (EOFException e)
   {  // 3
      show error message
       // 4
   }
  finally
  {
        out.close(); // 5
  }
  // 6
} // End of method m()
```

Case 1: no exception thrown: 1, 2, 5, 6

Case 2: exception thrown and caught

i) no rethrow

Execution passes through: 1,3,4,5,6

ii) rethrow (after showing error message)

Execution passes through: 1,3,5

Case 3: exception thrown, but not caught

Execution passes through: 1, 5

The Try-with-resources Statement

- Useful shortcut:

```
try (Resource res = . . .)
{
   work with res
}
```

- The resource class must implement the AutoCloseable interface, which has a single method: void close() throws Exception
- When the try block exits, then res.close() is called automatically.
- Example

```
try (Scanner in = new Scanner(Paths.get("in.txt"), "UTF-8"))
{
   while (in.hasNext())
      out.println(in.next().toUpperCase());
}
```

The Try-with-resources Statement

- You can specify multiple resources:

```
try (Scanner in = new Scanner(Paths.get("in.txt"), "UTF-8");
        PrintWriter out = new PrintWriter("out.txt"))
{
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

Tips for Proper Use of Exceptions

- Exception handling is not supposed to replace a simple test.

```
try
{
   s.pop();
}
catch (EmptyStackException e)
{
}
```

⇨

```
if (!s.empty()) s.pop();
```

- Do not micromanage exceptions.

```
for (i = 0; i < 100; i++)
{
   try { n = s.pop(); }
   catch (EmptyStackException e) { . . . }
   try { out.writeInt(n); }
   catch (IOException e) { . . . }
}
```

⇨

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e) { . . . }
catch (EmptyStackException e) { . . . }
```

Tips for Proper Use of Exceptions

- Make good use of the exception hierarchy:
  - Don't just throw a RuntimeException. Don't just catch Throwable. Find an appropriate subclass or create your own.
  - Respect the difference between checked and unchecked exceptions.
  - Do not hesitate to turn an exception into another exception that is more appropriate.
- Do not ignore exceptions:

```
try
{
   code that threatens to throw checked exceptions
}
catch (Exception e)
{ }
```

- Propagating exceptions is not a sign of shame.