Releasing Resources and Catching Exceptions

Releasing System Resources

Many of the resources that are used in this API, such as streams or channels, implement or extend the `java.io.Closeable` interface. A requirement of a `Closeable` resource is that the `close()` method must be invoked to release the resource when no longer required. Neglecting to close a resource can have a negative implication on an application's performance. The *try-with-resources* statement, described in the next section, handles this step for you.

Closing a Resource

For the sake of simplicity, the previous examples omits two things: the handling of the exceptions and the closing of your reader.

All the I/O operations throw the same, default exception in the Java I/O API: the `IOException`. Depending on the type of resource you are accessing, some more exceptions can be thrown. For instance, if your `reader` reads characters from a file, you may have to handle the `FileNotFoundException`.

Closing an I/O resource is a must in your application. Leaving resources unclose will cause your application to crash in the long run.

Starting with Java SE 7, the closing of I/O resources can be done using the *try-with-resources* statement. Let us rewrite the previous code using this pattern.

```java
Path path = Paths.get("file.txt");
try (BufferedReader reader = Files.newBufferedReader(path)) {

    // do something with the reader

} catch (IOException e) {
    // do something with the exception
}
```

In this example, the `reader` object can be used in the *try* block. When the program leaves this block, whether it is normally or exceptionally, the `close()` method of the `reader` object will be called for you.

Closing Several Resources

You may see file readers and buffered readers created using their constructors. These were the patterns used before the introduction of the `Files` factory class in Java SE 7. In this case, you will see the creation of several intermediate I/O resources, that must be closed in the right order.

In the case of a buffered reader created using a file reader, the correct pattern is the following.

```java
File file = new File("file.txt");

try (FileReader fileReader = new FileReader(file);
     BufferedReader bufferedReader = new BufferedReader(fileReader);) {

    // do something with the bufferedReader or the fileReader

} catch (IOException e) {
    // do something with the exception
}
```

Catching Exceptions

With file I/O, unexpected conditions are a fact of life: a file exists (or does not exist) when expected, the program does not have access to the file system, the default file system implementation does not support a particular function, and so on. Numerous errors can be encountered.

All methods that access the file system can throw an `IOException`. It is best practice to catch these exceptions by embedding these methods into a *try-with-resources statement*, introduced in the Java SE 7 release. The *try-with-resources* statement has the advantage that the compiler automatically generates the code to close the resource(s) when no longer required. The following code shows how this might look:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

For more information, see the section The try-with-resources Statement.

Alternatively, you can embed the file I/O methods in a try block and then catch any exceptions in a `catch` block. If your code has opened any streams or channels, you should close them in a `finally` block. The previous example would look something like the following using the *try-catch-finally* approach:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
    try{
        if (writer != null)
            writer.close();
    } catch (IOException x) {
        System.err.format("IOException: %s%n", x);
    }
```

```
}
```

For more information, see the section Catching and Handling Exceptions.

In addition to `IOException`, many specific exceptions extend `FileSystemException`. This class has some useful methods that return the file involved (`getFile()`), the detailed message string (`getMessage()`), the reason why the file system operation failed (`getReason()`), and the "other" file involved, if any (`getOtherFile()`).

The following code snippet shows how the `getFile()` method might be used:

```
try (...) {
    ...
} catch (NoSuchFileException x) {
    System.err.format("%s does not exist\n", x.getFile());
}
```

For purposes of clarity, the file I/O examples in this section may not show exception handling, but your code should always include it.

Using Varargs

Several Files methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the `CopyOption` argument indicates that the method accepts a variable number of arguments, or *varargs*, as they are typically called:

```
Path Files.move(Path, Path, CopyOption...)
```

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (`CopyOption[]`) of values.

In the following example, the method can be invoked as follows:

```
Path source = ...;
Path target = ...;
Files.move(source,
```

```
            target,
            REPLACE_EXISTING,
            ATOMIC_MOVE);
```

For more information about varargs syntax, see the section Arbitrary Number of Arguments.

Method Chaining

Many of the file I/O methods support the concept of method chaining.

You first invoke a method that returns an object. You then immediately invoke a method on that object, which returns yet another object, and so on. Many of the I/O examples use the following technique:

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem()
        .getUserPrincipalLookupService()
        .lookupPrincipalByName("me");
```

This technique produces compact code and enables you to avoid declaring temporary variables that you do not need.

Last update: *January 25, 2023*

Releasing Resources and Catching Exceptions ➜ Reading and Writing Small Files

Home > Tutorials > The Java I/O API > File Operations Basics > Releasing Resources and Catching Exceptions