Theory: Try with resources

19 minutes

Verify to skip | Start practicing

We have mentioned that **input streams** should be closed after they were used. Let's discuss what happens when you're working with outside resources: how closing can be performed and why it is important.

§1. Why close?

When an input stream is created, the JVM notifies the OS about its intention to work with a file. If the JVM process has enough permissions and everything is fine, the OS returns a **file descriptor** — a special indicator used by a process to access the file. The problem is that the number of file descriptors is limited. This is a reason why it is important to notify the OS that the job is done and the file descriptor that is held can be released for further reusing. In previous examples, we invoked the method `close` for this purpose. Once it is called, the JVM releases all system resources associated with the stream.

§2. Pitfalls

Resource releasing works if the JVM calls the `close` method, but it is possible that the method will not be called at all.

Look at the example:

```
1  Reader reader = new FileReader("file.txt");
2  // code which may throw an exception
3  reader.close();
```

Suppose something goes wrong before the `close` invocation and an exception is thrown. It leads to a situation in which the method will never be called and system resources won't be released. It is possible to solve the problem by using the **try-catch-finally** construction:

```
1  Reader reader = null;
2
3  try {
4      reader = new FileReader("file.txt");
5      // code which may throw an exception
6  } finally {
7      reader.close();
8  }
```

In this and the following examples, we assume that `file.txt` exists and do not check the instance of `Reader` for `null` in the `finally` block. It is done to keep the code snippet as simple as possible, but it is not safe in the case of a real application.

Thrown exceptions cannot affect the invocation of the `close` method now.

Unfortunately, this solution still has some problems. That is, the `close` method can potentially raise exceptions itself. Suppose, now there are two exceptions: the first was raised inside the `try` section, and the second was thrown by the `finally` section. It leads to the loss of the first exception. Let's see why this happens:

2 required topics

- ✕ Exception handling
- ✕ Input streams

Table of contents:

```
void readFile() throws IOException {
    Reader reader = null;
    try {
        reader = new FileReader("file.txt");
        throw new RuntimeException("Exception1");
    } finally {

reader.close(); // throws new RuntimeException("Exception2")
    }
}
```

First, the `try` block throws an exception. As we know, the `finally` block is invoked anyway. In our example, now the `close` method throws an exception. When two exceptions occur, which one is thrown outside the method? It will be the last one: `Exception2` in our case. It means we will never know that the `try` block raised an exception at all.

Let's try to reason and fix this. Ok, we don't want to lose the first exception, so we upgrade the code a little bit and handle `Exception2` right after it was thrown:

```
void readFile() throws IOException {
    Reader reader = null;
    try {
        reader = new FileReader("file.txt");
        throw new RuntimeException("Exception1");
    } finally {
        try {

    reader.close(); // throws new RuntimeException("Exception2")
        } catch (Exception e) {
            // handle the Exception2
        }
    }
}
```

Now, the piece of code throws `Exception1` outside. It may be correct, but we still do not save information on both exceptions, and sometimes we don't want to lose it. So now, let's see how we can handle this situation nicely.

§3. Solution

A simple and reliable way called **try-with-resources** was introduced in Java 7.

```
try (Reader reader = new FileReader("file.txt")) {
    // some code
}
```

This construction has two parts enclosed by round and curly brackets. Round brackets contain statements that create an input stream instance. It is possible to create several objects as well. The code below is also fine:

```
try (Reader reader1 = new FileReader("file1.txt");
     Reader reader2 = new FileReader("file2.txt")) {
    // some code
}
```

The second part just contains some code for dealing with the object that was created in the first part.

As you see, there are no explicit calls of the `close` method at all. It is implicitly invoked for all objects declared in the first part. The construction guarantees closing all resources in a proper way.

Since Java 9, you may initialize an input stream outside the construction and then declare it in round brackets:

```
Reader reader = new FileReader("file.txt");
try (reader) {
    // some code
}
```

> Surely we do our best to write error-free programs. However, it is difficult to foresee all possible problems. The best practice is to wrap any code dealing with system resources by the try-with-resources construction

You may also use try-with-resources as a part of try-catch-finally like this:

```
try (Reader reader = new FileReader("file.txt")) {
    // some code
} catch(IOException e) {
    ...
} finally {
    ...
}
```

Now let's go back to our two-exceptions case. If both the `try` block and `close` method throw exceptions `Exception1` and `Exception2`:

```
void readFile() throws IOException {
    try (Reader reader = new FileReader("file.txt")) {
        throw new RuntimeException("Exception1");
    }
}
```

the method throws the resulting exception, which comprises information on both exceptions. It looks like this:

```
Exception in thread "main" java.lang.RuntimeException: Exception1
    at ...
    Suppressed: java.lang.RuntimeException: Exception2
        at ...
```

§4. Closeable resources

We have dealt with a file input stream to demonstrate how try-with-resources is used. However, not only resources based on files should be released. Closing is crucial for other outside sources like web or database connections. Classes that handle them have a `close` method and therefore can be wrapped by the try-with-resources statement.

For example, let's consider `java.util.Scanner`. Earlier we used `Scanner` for reading data from the standard input, but it can read data from a file as well. `Scanner` has a `close` method for releasing outside sources.

Let's consider an example of a program that reads two integers separated by a space from a file and prints them:

```
try (Scanner scanner = new Scanner(new File("file.txt"))) {
    int first = scanner.nextInt();
    int second = scanner.nextInt();
    System.out.println("arguments: " + first + " " + second);
}
```

Suppose something went wrong and the file content is `123 not_number`, where the second argument is a `String`. It leads to a `java.util.InputMismatchException` while parsing the second argument. Try-with-resources guarantees that file-related resources are released properly.

§5. Conclusion

Inappropriate resource handling may lead to serious problems. Resources associated with files, web, database, or other outside sources should be released after being used. Standard library classes dealing with outside sources have a `close` method for that purpose. Sometimes releasing resources in a proper way may get complicated. To simplify the process, Java 7 introduced the try-with-resources construction that does all the work for you. Do not forget to use it when you're dealing with system resources.

Report a typo

262 users liked this piece of theory. 9 didn't like it. **What about you?**

Start practicing

Verify to skip

Comments (9) Hints (0) Useful links (2) Show discussion