Theory: Try with resources

14 minutes 0 / 3 problems solved

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

§1. Why close?

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

§2. Pitfalls

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

Look at the example:

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

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

```
Reader reader = null;

try {
    reader = new FileReader("file.txt");
    // code which may throw an exception
} finally {
    reader.close();
}
```

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

Thrown exceptions can not affect the invocation of `close` method now.

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

```
void readFile() throws IOException {
    Reader reader = null;
    try {
        reader = new FileReader("file.txt");
        throw new RuntimeException("Exception1");
    } finally {
        reader.close(); // throws new RuntimeException("Exception2")
    }
}
```

Current topic:

Try with resources

Topic depends on:

- ✕ Exception handling (Stage 7)
- ✕ Input streams

Table of contents:

First, *try* block throws an exception. As we know *finally* block is invoked anyway. In our example, now `close` method throws an exception. When two exceptions occurred, which one is thrown outside the method? It will be the last one: `Exception2` in our case. It means we will never know that *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 of creating 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 `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 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 *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 outer sources such as web or database connections. Classes that handle them have `close` method and therefore can be wrapped by 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 `close` method for releasing outer sources.

Let's consider an example of the program that reads two integers separated by 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 outer sources should be released after being used. Standard library classes dealing with outer sources have `close` method for that purpose. Sometimes releasing resources in a proper way may get complicated. To simplify the process Java 7 introduced try-with-resources construction that does all the work for you. Do not forget to use it when you're dealing with system resources.

Comments (3) Hints (0) Useful links (1) Show discussion