Theory: Exception handling

32 minutes

Verify to skip | Start practicing

As you already know, an exception interrupts the normal execution of a program. Normally this is not what we want to happen. Luckily, it is possible to write some code that will handle the exception without stopping the whole program. To do that, Java provides an **exception handling** mechanism that works with both **checked** and **unchecked** exceptions.

§1. How to handle an exception

After a line of code throws an exception, the Java runtime system attempts to find a suitable handler for it. Such a handler can be located in the same method where the exception occurred or in the calling method. As soon as a suitable handler is found and executed, the exception is considered as handled and the program runs normally.

Technically, an exception can be handled in the method where it occurs or in the calling method. The best approach to handle an exception is to do it in a method that has sufficient information to make the correct decision based on this exception.

Let's now learn three keywords for handling exceptions: `try`, `catch` and `finally`.

§2. The try-catch statement

Here is a simple `try-catch` template for handling exceptions:

```
try {
    // code that may throw an exception
} catch (Exception e) {
    // code for handling the exception
}
```

The `try` block is used to wrap the code that may throw an exception. This block can include all lines of code, including method calls.

The `catch` block is a handler for the specified type of exception and all of its subclasses. This block is executed when an exception of the corresponding type occurs in the `try` block.

> Note that the specified type in a `catch` block must extend the `Throwable` class.

In the presented template, the `catch` block can handle exceptions of the `Exception` class and all classes derived from it.

The following example demonstrates the execution flow with `try` and `catch`.

1 required topic

✕ Hierarchy of exceptions

10 dependent topics

Custom exceptions
Thread management
Reading files
Managing files
Writing files
Try with resources
Serialization basics
Java 11 HTTP client
Connecting to a database with JDBC
Exception handling

Table of contents:

↑ Exception handling
§1. How to handle an exception
§2. The try catch statement
§3. Getting info about an exception
§4. Catching multiple exceptions
§5. The finally block
§6. Conclusion
Discussion

```
    System.out.println("before the try-
catch block"); // it will be printed

    try {

     System.out.println("inside the try block before an exception"); // it wil

         System.out.println(2 / 0); // it throws ArithmeticException


     System.out.println("inside the try block after the exception"); // it won
    } catch (Exception e) {
         System.out.println("Division by zero!"); // it will be printed
    }

    System.out.println("after the try-
catch block"); // it will be printed
```

The output:

```
before the try-catch block
inside the try block before an exception
Division by zero!
after the try-catch block
```

The program does not print `"inside the try block after the exception"` since the `ArithmeticException` aborted the normal flow of the execution. Instead, it executes the print statement in the `catch` block. After completion of the `catch` block, the program executes the next statement (printing `"after the try-catch block"`) without returning to the `try` block again.

Replacing `Exception` with `ArithmeticException` or `RuntimeException` in the `catch` statement does not change the execution flow of the program. But replacing it with `NumberFormatException` will make the handler unsuitable for the exception and the program will fail.

> As we noted earlier, the `try-catch` statement can handle both checked and unchecked exceptions. But there is a difference: checked exceptions must be wrapped with a `try-catch` block or declared to be thrown in the method, while unchecked exceptions don't have to.

§3. Getting info about an exception

When an exception is caught by a `catch` block, it is possible to get some information on it:

```
try {
    double d = 2 / 0;
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

This code prints:

```
An exception occured: / by zero
```

§4. Catching multiple exceptions

It is always possible to use a single handler for all types of exceptions:

```
try {
    // code that may throw exceptions
} catch (Exception e) {
    System.out.println("Something goes wrong");
}
```

Obviously, this approach does not allow us to perform different actions depending on the type of exception that has occurred. Fortunately, Java supports the use of several handlers inside the same `try` block.

```
try {
    // code that throws exceptions
} catch (IOException e) {
    // handling the IOException and its subclasses
} catch (Exception e) {
    // handling the Exception and its subclasses
}
```

When an exception occurs in the `try` block, the runtime system determines the first suitable `catch` block according to the type of the exception. Matching goes from top to bottom.

> Important, the `catch` block with the base class has to be written below all blocks with subclasses. In other words, the more specialized handlers (like `IOException`) must be written before the more general ones (like `Exception`). Otherwise, the code won't compile.

Since Java 7, you can use a **multi-catch** syntax to have several exceptions handled in the same way:

```
try {
    // code that may throw exceptions
} catch (SQLException | IOException e) {
    // handling SQLException, IOException and their subclasses
    System.out.println(e.getMessage());
} catch (Exception e) {
    // handling any other exceptions
    System.out.println("Something goes wrong");
}
```

In the code above `SQLException` and `IOException` (alternatives) are separated by the `|` character. They will be handled in the same way.

> Note that alternatives in a multi-catch statement cannot be each other's subclasses.

§5. The finally block

There is another possible block called `finally`. All statements present in this block will always execute regardless of whether an exception occurs in the `try` block or not.

```
try {
    // code that may throw an exception
} catch (Exception e) {
    // exception handler
} finally {
    // code always be executed
}
```

The following example illustrates the order of execution of the `try-catch-finally` statement.

```
try {
    System.out.println("inside the try block");
    Integer.parseInt("101abc"); // throws NumberFormatException
} catch (Exception e) {
    System.out.println("inside the catch block");
} finally {
    System.out.println("inside the finally block");
}

System.out.println("after the try-catch-finally block");
```

The output:

```
inside the try block
inside the catch block
inside the finally block
after the try-catch-finally block
```

If we remove the line that throws `NumberFormatException`, the `finally` block is still executed after the `try` block.

```
inside the try block
inside the finally block
after the try-catch-finally block
```

Interesting: the `finally` block is executed even if an exception occurs in the `catch` block.

It is also possible to write `try` and `finally` without a `catch` block at all.

```
try {
    // code that may throw an exception
} finally {
    // code always be executed
}
```

In this template, the `finally` block is executed right after the `try` block.

§6. Conclusion

The `try-catch` statement allows us to handle both checked and unchecked exceptions .

The `try` block wraps the code that may throw an exception while the `catch` block handles this exception in case it occurs, also allowing us to get some information about it. It is possible to use several handlers to provide different scenarios for different types of exceptions.

Finally, there's an optional `finally` block that is always executed. Its main feature is that it executes even if we fail to handle an exception at all.