Java Part 8: Exceptions and Working with Files

Real programs meet bad input, missing files and unexpected values. An exception is how Java reports such a problem while the program runs, and handling it well is the difference between a crash and a useful message. Four steps cover this: catch, classify, clean up and read the file. In this part, Dina, a revenue analyst at a hotel group, uses them to read a bookings file that contains a few bad rows, and each step answers one practical question.

1. Catch

When something goes wrong, Java creates an exception object and throws it, and the normal flow stops. Java then looks up through the chain of calls for a matching catch block, and if it finds none, the program ends and prints a stack trace. We guard risky code with try, and we place the recovery in catch, naming the type of exception we can really handle. Several catch blocks can follow one try, with the most specific type first. An empty catch block that swallows the problem silently is the worst habit, because the failure then hides: we at least log the message or let the exception travel on. The question for this step is: what can go wrong here, and what is the sensible reaction?

Example: A web shop asks the customer to type a quantity and converts the text into a number. One customer types letters instead, so the conversion throws an exception. The shop guards it with try, and the catch block shows a friendly message and asks again. Because the catch handles only that exception, the shop keeps running.

At work: Dina's program reads each line of the bookings file and converts the price field into a BigDecimal. On the four hundredth line the price is the word pending, and the conversion throws a NumberFormatException. She puts the conversion inside try and catches exactly that type. The catch block records the line number and the message, and the program moves on to the next line.

2. Classify

Exceptions come in two families. Checked exceptions, such as IOException, are problems outside our control, like a missing file, and the compiler forces us either to catch them or to declare them with the word throws in the method signature. Unchecked exceptions, which extend RuntimeException, such as IllegalArgumentException and NumberFormatException, usually point to a mistake in the program or in the data, and the compiler does not force us to handle them. Errors, such as running out of memory, are not something a normal program tries to recover from. We can raise our own exception with the word throw and a clear message, for example when a method receives a negative number of nights. A method that cannot do its job should throw an exception rather than return a silent wrong value. The question for this step is: should this method handle the problem, or pass it to a caller who knows more?

Example: A payroll method receives a negative number of hours. It cannot calculate a fair salary from that value, so it throws an IllegalArgumentException with the message hours must not be negative. This unchecked exception points to bad data, so the compiler does not force anyone to handle it. The caller sees the problem immediately instead of printing a wrong salary.

At work: Dina's method that opens the bookings file declares throws IOException, because a missing file is not something it can fix. The main method catches the exception and prints a clear message with the file name. Her method that creates a booking from a line throws an IllegalArgumentException when the number of nights is below one. Now each layer of her program deals with the problems it understands.

This document is a property of Asaseeyat Academy - © Copy Right Protected.

3. Clean Up

Some resources must be closed after use, such as files, network connections and database sessions. A finally block runs whether or not an exception occurred, which makes it a place for cleanup. The modern and safer way is try with resources: we declare the resource in parentheses after try, and Java closes it automatically, even when an exception is thrown. Resources that can be used this way implement the AutoCloseable interface, the contract idea from the previous part. When several resources are opened together, they are closed in the reverse order. Forgetting to close a file may keep it locked for other programs. The question for this step is: what must be released when we are done, even if something fails?

Example: A reporting tool opens a connection to a database to read the sales. It declares the connection in parentheses after try, and the connection implements AutoCloseable. At the end of the block, Java closes it automatically. Even if the query fails halfway, the connection is closed too, so no connection is left open.

At work: Dina's first version opens a reader on the bookings file and forgets to close it when a line fails. A colleague then finds the file locked and cannot write over it. She rewrites the reading code in the try with resources style. The file is now closed in every case, whether the report succeeds or fails.

4. Read the File

The class Files in the java.nio.file package reads and writes whole files with one call. Path.of builds a path from a file name. The method readAllLines returns a list with one string per line, and it throws IOException, so it needs handling. For very large files we read line by line instead, so the whole file never sits in memory. The method split on a comma separates the fields of a simple CSV file, but fields that contain commas inside quotation marks need a proper CSV library. To write results, the methods write and writeString save text to a file with one call. The question for this step is: what shape does each line have, and what do we do with a line that does not match?

Example: A logistics analyst receives a list of shipments exported as a text file. One call to readAllLines returns every line, and each line is split on the comma into the fields of a shipment object. A line with a missing field does not match the expected shape, so it goes to a separate problems list with its number.

At work: Dina reads the file with a single call that returns every line: `List<String> lines = Files.readAllLines(Path.of("bookings.csv"));`. She splits each line on the comma, builds a booking from the fields and catches the exceptions of the earlier steps. Good bookings go to the report, and bad lines go to a problem list with their line numbers. The monthly report is finished, and a short file lists the three rows that need correction.

Conclusion

To recap, an exception is Java's way to report a problem while the program runs. We catch only what we can handle, classify problems as checked or unchecked, and clean up with try with resources. Files reads and writes whole files, and a bad line should be recorded instead of stopping everything. Dina's monthly report now survives a bad row and tells her exactly which rows need correction. Next time, we will put everything together in a small business application. A good habit is to write error messages that name the file, the line and the value.

Sources

- Oracle: The Java Tutorials, Exceptions – https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html
- Oracle: The Java Tutorials, The try-with-resources Statement – https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
- Oracle: The Java Tutorials, Reading, Writing, and Creating Files – https://docs.oracle.com/javase/tutorial/essential/io/file.html
- Oracle: The Java Tutorials, Catch or Specify Requirement – https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html
- dev.java: Learn Java – https://dev.java/learn/

- Java is a trademark of Oracle and/or its affiliates. This video is independent training material and is not affiliated with Oracle.

This document is a property of Asaseeyat Academy - © Copy Right Protected.