Day 17: Exception handling

1 Exception handling

If you try to parse (e.g. using `Integer.parseInteger()`) something that is not an number, you get a `NumberFormatException`. If you try to call a method on an object, but the pointer to the object is `null`, you get a `NullPointerException`. But what are exceptions? And is there a way to handle them so that they do not always crash a program?

Exceptions represent exceptional circumstances that should never occur in the normal (some would say *ideal*) execution of a program. When one of these exceptional circumstances —like trying to follow a `null` pointer— happens, the Java Virtual Machine throws an exception to indicate it.

An exception interrupts the normal execution of the method where it happened. It *breaks* the method where it was happening and moves up to the calling method, where it may be caught. If it is not caught in the calling method, it will go up again, and again, until it is caught. If it is not caught, not even in the original `main` method, it will be caught by the Java Virtual Machine and it will be printed on the standard output as you have seen several times already.

```
Exception in thread "main" java.lang.NullPointerException
  at LinkedListNode.addNode(LinkedListNode.java:49)
  at LinkedList.addNode(LinkedList.java:30)
  at HospitalManager.launch(HospitalManager.java:71)
  at HospitalManager.main(HospitalManager.java:13)
```

Reading a stack trace

An exception contains information about the state of the Java Virtual Machine when it was thrown. In particular, it contains the whole stack of method calls at the time. This is very useful because it helps programmers to find out what went wrong (and maybe why).

The most useful line of the stack trace tends to be the first one, because it says exactly where the exceptional situation happened. For example, in the stack trace above we can observe that a `NullPointerException` was thrown at line 49 of the code of class `LinkedListNode`. With this information a programmer can look at that specific line and try to understand why a null pointer was accessed when this should not happen.

If we want more information we just need to follow the stack trace line by line. We can see that the program started at the main method of class `HospitalManager`, that the `launch()` method was called on like 13, and that this method called the method `addNode()` in class `LinkedList` (on line 71), which in turn called the method `addNode()` of class `LinkedListNode` (on line 30), where the exception was thrown.

There is no limit to the length of a stack trace. It can have as many steps as the size of your stack, which is several thousands of calls deep (depending on the number of local variables per method call).

Sometimes the stack trace goes into code that has been written by a third party, as in the following example (comments ommitted for brevity):

```
01    public class ExceptionThrower {
02       public static void main(String[] args) {
03          ExceptionThrower et = new ExceptionThrower();
04          et.launch();
05       }
06       private void launch() {
07          System.out.print("Write a number: ");
08          int n = getNumber();
09          String isOdd = (n % 2 == 0) ? "odd" : "even";
10          System.out.println("You entered " + n + ", it is " + isOdd + ".");
11       }
12       private int getNumber() {
13          String str = System.console().readLine();
14          int result = Integer.parseInt(str);
15          return result;
16       }
17    }
```

If you enter numbers like 2 or 19, all will work well. However, if you enter something like “two” you will get a detailed stack trace to blame you for your incompetence typing integer numbers:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:449)
  at java.lang.Integer.parseInt(Integer.java:499)
  at ExceptionThrower.getNumber(ExceptionThrower.java:14)
  at ExceptionThrower.launch(ExceptionThrower.java:8)
  at ExceptionThrower.main(ExceptionThrower.java:4)
```

We can see that the program started in the `main` method, then called the method `launch()`, that this method called the method `getNumber()`, and then this method called `Integer.parseInt()`. From that point on, the stack trace shows method calls happening inside classes of the basic Java Library, like `Integer` and `NumberFormatException`, but you can see that the information is exactly the same: method calls and line numbers. If you have access to the source code of those external classes, you will see the line numbers and will be able to trace the stack of method calls in detail. If you work with external libraries from which you do not have the source code, you will only see the names of the methods but not the line numbers.

Exceptions interrupt the normal flow

The example allows us to observe the “interrupting” nature of exceptions. We can see that `NumberFormatException` interrupted the execution of method `getNumber()` (line 15 was never executed), then interrupted the caller method `launch()` (lines 09–10 were never executed), and then interrupted the `main` method. Since the exception was never *caught*, it went all the way up to the `main` method and then was shown by the Java Virtual Machine.

1.1 Catching exceptions

Exceptions are handled by using try/catch constructs: you "try" some code where exceptions may be thrown; if exceptions are thrown, you "catch" them and do something about them. See the following example:

```
12    private int getNumber() {
13       int result = 0; // default
14       try {
15          String str = System.console().readLine();
16          result = Integer.parseInt(str);
17          System.out.println("You entered " + result + ".");
18       } catch (NumberFormatException ex) {
19          System.out.println("What you entered is not an integer number!");
20       }
21       return result;
22    }
```

If a `NumberFormatException` is thrown at line 16, the normal execution of the method is interrupted. Program execution jumps out of the current `try` clause (i.e. until the next closing curly bracket). It is caught there —there is a `catch` statement for that type of exception— and the code inside the `catch` clause is executed. Then the execution proceeds normally to line 21, where `result` is returned.

If an exception jumps up to the end of the scope and is not caught, it will jump up to the end of the next scope. If it is still not caught, it will jump up to the end of the next scope and so on. This usually means that the exception is moving up the method stack from calling method to calling method until it is caught.

If it is never caught, it will crash your program and appear on screen. In a way, you can think of this as a huge `catch` statement just out of your main method that only prints the stack trace.

```
// WARNING: this is only a metaphore, not real code
try {
    mainMethod();
} catch (AnyException ex) {
    ex.printStackTrace().
    System.exit(0); // Stop the Java Virtual Machine
}
```

The method `printStackTrace()` is very useful, and is almost always used in `catch` statements because it really helps in debugging your program when a exception is thrown on an unexpected part of your code. The method prints the name of every single method that had been called when the exception was thrown, including the line that was being executed (if the source code for the class is known).

A `try` clause can have many `catch` clauses associated to it, as in the following example:

```
12    private int getNumber() {
13       int result = 0; // default
14       try {
15           System.out.print("Enter a number with at least 3 digits: ");
16           String str = System.console().readLine();
17           result = Integer.parseInt(str);
18           System.out.println("You entered " + result + ".");
19           char thirdDigit = str.charAt(2);
20           System.out.println("The third digit is " + thirdDigit + ".");
21       } catch (NumberFormatException ex) {
22           System.out.println("ERROR 1: You did not enter an integer.");
23       } catch (IndexOutOfBoundsException ex) {
24           System.out.println("ERROR 2: Your integer did not have three digits.");
25       }
26       return result;
27     }
```

If a `NumberFormatException` is thrown on line 17, the execution of the program will jump to line 21, where it will be caught. After that `catch` clause is executed, the program will continue at the end of the try/catch construct at line 26. If a `IndexOutOfBoundsException` is thrown at line 19, the execution of the program will jump to line 21, and then to line 23, where it will be caught. After that `catch` clause is executed, the program will continue at the end of the try/catch construct at line 26. Once an exception is caught, it does not go "up" anymore. It disappears.

A `try` clause can also have a `finally` clause, which can used for cleaning up resources. We will see more of this when we learn about input/output.

1.2 Which methods throw exceptions?

The exceptions thrown by a method should be specified in the documentation, i.e. in its JavaDoc comments. This is done by means of `@throws`. This is as important as describing the arguments for the method or the return type. (Check the methods in the core Java Library: they all document the exceptions they can throw). The JavaDoc for the aforemention method `getNumber()` looks like this:

```
/**
 * Gets a number as introduced by the user on the console, which
 * must be at least three digits long. Otherwise, an
 * IndexOutOfBounds will be thrown.
 *
 * @return the number introduced by the user
 *
 * @throws NumberFormatException if the user types a non-number
 * @throws IndexOutOfBoundsException if the user types less than three digits
 */
```

Additionally, many exceptions must be documented in the code too. These are called *checked exceptions.*

1.3 Checked and runtime exceptions

All exceptions in Java extend class `Exception`, in package `java.lang`. Many exceptions, including all you have met so far, are in the same package. Classes from this package are automatically imported, so you do not need to do import them explicitly.

There is a subclass of `Exception` called `RuntimeException`. Descendants of the latter are called (unsurprisingly) *runtime exceptions*, while all other descendants of `Exception` are called *checked exceptions*. The difference between these two types is that checked exceptions *must* be either explicitly declared or explicitly caught in the code.

If a checked exception is thrown in some line of a method it must be caught (by using a try/catch construct). If it is not caught, it must be declared as possibly thrown, as in the following example:

```
/**
 * ...
 * @throws SomeCheckedException when that exceptional situation happens
 */
public void doggyMethod() throws SomeCheckedException {
    // do things here that throw some checked exception
}
```

By declaring that the method throws `SomeCheckedException`, the programmer is telling the compiler to check that any method calling `doggyMethod` is catching the exception. . . or throwing it up once again. If a checked exception can be thrown in a line, the line must be inside a `try` block or the exception must be declared as thrown. Otherwise, the compiler will complain.

This does not happen with `RuntimeException` and its decendants (including those that you already know like `NullPointerException` and `NumberFormatException`). The reason is that runtime exceptions can be thrown by so many commonly used methods that it would be a lot of work to always declare them or catch them.

1.4 Mishandling exceptions

There are two common mistakes when handling exceptions, and both of them are related to laziness. As laziness is an essential part of being human (or at least very common), it is important to be warned to avoid the same mistakes that thousands of programmers have done in the past (sometimes with terrible consequences).

The first common mistake is **not being specific** enough when catching exceptions. When your code throws three or four different exceptions, it is quite tempting to use a quick `catch(Exception)` that will work for all of them. This is usually a bad idea in the long run because different exceptions happen in different circumstances and should be dealt with in different ways. If you use the same code for all of them, this may be confusing further down the line. As a rule of thumb, it is better to catch different exceptions separately.

The second —and more dangerous— mistake is **not doing anything** when catching an exception. This is really bad. Once an exception is caught, it disappears, it does not move up anymore. Unless you do something about it there and then, nobody will ever know that an exceptional situation happened. For all your users know, your program may be deleting money from their bank accounts without a hint that something is wrong until it is too late. This is so important that I will write it in big letters so that you do not ever forget. Repeat with me:

Do not write empty `catch` blocks. Ever.

Sometimes it is boring to write `catch` blocks because you are interested in the main algorithm and not the exceptional situations along the borders of your problem. If you follow the Test Driven Methodology we have seen in past weeks this will happen less often, but it will still happen, and you are going to feel a strong temptation to just write an empty `catch` block and "come later to write it properly".

Believe me when I tell you that "later" never comes. If you cannot think of anything reasonable for your `catch` block —or do not want to—, copy and paste either of the following two lines (or both) in there and "come later to write it properly":

```
ex.printStackTrace();
throw new RuntimeException(ex); // See more about this in next section
```

If you leave empty `catch` blocks in your code you will face many sleepless nights hunting missing exceptions and ghost bugs. You have been warned.

1.5 Throwing your own exceptions

Creating your own exceptions is easy. Just create a class that extends either `Exception` (for checked exceptions) or `RuntimeException` (for runtime exceptions). This is rare, but it can be useful in some projects where you want to be more precise in your exception handling beyond the exceptions offered by the Java library.

Exceptions usually have three constructors: one without parameters, one that takes a String (the message of the exception), and one that takes another exception (so the new exception acts as a wrapper of the old exception).

Throwing an exception is very easy. It is done by means of the keyword `throw`. (See the example below.) Any exception can be thrown[1].

```
//...
throw new SomeException("Something exceptional happened!");
// ...
```

This line creates a new exception (an object of class `SomeException`) and throws it. The exception can be created in a different line, of course, and assigned to a variable (which is then thrown).

Re-throwing an exception When an exception is caught, it disappears and does not go further up in the method call stack... unless that is exactly what we want, in which case we can just either wrap it inside another exception and throw the latter:

```
//...
} catch(SomeException e) {
    throw new RuntimeException(e);
}
// ...
```

[1]Note that "throw" and "throws" are different. The former explicitly throws an exception while the latter declares than a method can throw an exception to the calling party.

or —simpler but not recommended— re-throw it directly (it is still an exception, after all, so it can be thrown):

```
//...
} catch(SomeException e) {
    throw e;
}
// ...
```

The first option is quite better because it provides additional information. I would *not* recommend to use the latter *in any case*. It is only included here to illustrate the relationship between exceptions and the `throw` keyword.

1.6 Afterthoughts on exception handling

1.6.1 Should I create checked or runtime exceptions?

This is a difficult question, and there is not a clear answer for it. Personally, I would recommend to make all new exceptions `RuntimeException`.

Runtime exceptions will make your code easier to write and not more difficult to read. They will go up your stack when thrown and crash your program —which provides a good incentive to fix the bug (probably by catching those exceptions at the right place).

Eliminating the distinction between checked and runtime exceptions (having only the latter) seems to be the route taken by more modern languages, including some very popular ones like Python.

1.6.2 Throw throws throwables

The keyword `throw` can be used to throw other classes, not just exceptions. In particular, any descendant of class `Throwable` can be thrown (and caught).

There are two types of throwables in Java: `Exception` and `Error`. We already know exceptions. Errors are used to indicate serious problems, like the `StackOverflowError` that is thrown when the program runs out of stack space (e.g. due to a buggy infinite recursive call). It is not expected from normal applications to catch errors, so you should not worry about. On the other hand, you should know that it is possible.