Exception handling in Java isn’t an easy topic. Beginners find it hard to understand and even experienced developers can spend hours discussing how and which exceptions should be thrown or handled.

That’s why most development teams have their own set of rules on how to use them. And if you’re new to a team, you might be surprised how different these rules can be to the ones you’ve used before.

Nevertheless, there are several best practices that are used by most teams. Here are the 9 most important ones that help you get started or improve your exception handling.

**1. Clean Up Resources in a Finally Block or Use a Try-With-Resource Statement**

It happens quite often that you use a resource in your try block, like an *[InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "_blank)*, which you need to close afterward. A common mistake in these situations is to close the resource at the end of the try block.

public void doNotCloseResourceInTry() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

// do NOT do this

inputStream.close();

} catch (FileNotFoundException e) {

log.error(e);

} catch (IOException e) {

log.error(e);

}

}

The problem is that this approach seems to work perfectly fine as long as no exception gets thrown. All statements within the try block will get executed, and the resource gets closed.

But you added the try block for a reason. You call one or more methods which might throw an exception, or maybe you throw the exception yourself. That means you might not reach the end of the try block. And as a result, you will not close the resources.

You should, therefore, put all your clean up code into the finally block or use a try-with-resource statement.

**Use a Finally Block**

In contrast to the last few lines of your try block, the finally block gets always executed. That happens either after the successful execution of the try block or after you handled an exception in a catch block. Due to this, you can be sure that you clean up all the opened resources.

public void closeResourceInFinally() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

log.error(e);

}

}

}

}

**Java 7’s Try-With-Resource Statement**

Another option is the try-with-resource statement which I explained in more detail in my [introduction to Java exception handling](https://stackify.com/specify-handle-exceptions-java/#tryWithResource).

You can use it if your resource implements the *[AutoCloseable](https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html" \t "_blank)* interface. That’s what most Java standard resources do. When you open the resource in the *try* clause, it will get automatically closed after the *try* block got executed, or an exception handled.

public void automaticallyCloseResource() {

File file = new File("./tmp.txt");

try (FileInputStream inputStream = new FileInputStream(file);) {

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} catch (IOException e) {

log.error(e);

}

}

**2. Prefer Specific Exceptions**

The more specific the exception is that you throw, the better. Always keep in mind that a co-worker who doesn’t know your code, or maybe you in a few months, need to call your method and handle the exception.

Therefore make sure to provide them as many information as possible. That makes your API easier to understand. And as a result, the caller of your method will be able to handle the exception better or [avoid it with an additional check](https://stackify.com/top-java-software-errors/).

So, always try to find the class that fits best to your exceptional event, e.g. throw a *[NumberFormatException](https://docs.oracle.com/javase/8/docs/api/java/lang/NumberFormatException.html" \t "_blank)*instead of an *[IllegalArgumentException](https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html" \t "_blank)*. And avoid throwing an unspecific *Exception*.

public void doNotDoThis() throws Exception {

...

}

public void doThis() throws NumberFormatException {

...

}

**3. Document the Exceptions You Specify**

Whenever you [specify an exception](https://stackify.com/specify-handle-exceptions-java/#specify) in your method signature, you should also [document it in your Javadoc](http://blog.joda.org/2012/11/javadoc-coding-standards.html). That has the same goal as the previous best practice: Provide the caller as many information as possible so that he can avoid or handle the exception.

So, make sure to add a *@throws* declaration to your Javadoc and to describe the situations that can cause the exception.

/\*\*

\* This method does something extremely useful ...

\*

\* @param input

\* @throws MyBusinessException if ... happens

\*/

public void doSomething(String input) throws MyBusinessException {

...

}

**4. Throw Exceptions With Descriptive Messages**

The idea behind this best practice is similar to the two previous ones. But this time, you don’t provide the information to the caller of your method. The exception’s message gets read by everyone who has to understand what had happened when the exception was reported in the log file or your monitoring tool.

It should, therefore, describe the problem as precisely as possible and provide the most relevant information to understand the exceptional event.

Don’t get me wrong; you shouldn’t write a paragraph of text. But you should explain the reason for the exception in 1-2 short sentences. That helps your operations team to understand the severity of the problem, and it also makes it easier for you to analyze any service incidents.

If you throw a specific exception, its class name will most likely already describe the kind of error. So, you don’t need to provide a lot of additional information. A good example for that is the *NumberFormatException*. It gets thrown by the constructor of the class *java.lang.Long* when you provide a *String* in a wrong format.

try {

new Long("xyz");

} catch (NumberFormatException e) {

log.error(e);

}

The name of the *NumberFormatException* class already tells you the kind of problem. Its message only needs to provide the input string that caused the problem. If the name of the exception class isn’t that expressive, you need to provide the required information in the message.

17:17:26,386 ERROR TestExceptionHandling:52 - java.lang.NumberFormatException: For input string: "xyz"

**5. Catch the Most Specific Exception First**

Most IDEs help you with this best practice. They report an unreachable code block when you try to catch the less specific exception first.

The problem is that only the first catch block that matches the exception gets executed. So, if you catch an *IllegalArgumentException* first, you will never reach the catch block that should handle the more specific *NumberFormatException* because it’s a subclass of the *IllegalArgumentException*.

Always catch the most specific exception class first and add the less specific catch blocks to the end of your list.

You can see an example of such a try-catch statement in the following code snippet. The first catch block handles all *NumberFormatException*s and the second one all *IllegalArgumentException*s which are not a *NumberFormatException*.

public void catchMostSpecificExceptionFirst() {

try {

doSomething("A message");

} catch (NumberFormatException e) {

log.error(e);

} catch (IllegalArgumentException e) {

log.error(e)

}

}

**6. Don’t Catch Throwable**

[*Throwable*](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html) is the superclass of all exceptions and errors. You can use it in a catch clause, but you should never do it!

If you use *Throwable* in a catch clause, it will not only catch all exceptions; it will also catch all errors. Errors are thrown by the JVM to indicate serious problems that are not intended to be handled by an application. Typical examples for that are the *[OutOfMemoryError](https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html)* or the *[StackOverflowError](https://docs.oracle.com/javase/8/docs/api/java/lang/StackOverflowError.html)*. Both are caused by situations that are outside of the control of the application and can’t be handled.

So, better don’t catch a *Throwable* unless you’re absolutely sure that you’re in an exceptional situation in which you’re able or required to handle an error.

public void doNotCatchThrowable() {

try {

// do something

} catch (Throwable t) {

// don't do this!

}

}

**7. Don’t Ignore Exceptions**

Have you ever analyzed a bug report where only the first part of your use case got executed?

That’s often caused by an ignored exception. The developer was probably pretty sure that it would never be thrown and added a catch block that doesn’t handle or logs it. And when you find this block, you most likely even find one of the famous “This will never happen” comments.

public void doNotIgnoreExceptions() {

try {

// do something

} catch (NumberFormatException e) {

// this will never happen

}

}

Well, you might be analyzing a problem in which the impossible happened.

So, please, never ignore an exception. You don’t know how the code will change in the future. Someone might remove the validation that prevented the exceptional event without recognizing that this creates a problem. Or the code that throws the exception gets changed and now throws multiple exceptions of the same class, and the calling code doesn’t prevent all of them.

You should at least write a log message telling everyone that the unthinkable just had happened and that someone needs to check it.

public void logAnException() {

try {

// do something

} catch (NumberFormatException e) {

log.error("This should never happen: " + e);

}

}

**8. Don’t Log and Throw**

That is probably the most often ignored best practice in this list. You can find lots of code snippets and even libraries in which an exception gets caught, logged and rethrown.

try {

new Long("xyz");

} catch (NumberFormatException e) {

log.error(e);

throw e;

}

It might feel intuitive to log an exception when it occurred and then rethrow it so that the caller can handle it appropriately. But it will write multiple error messages for the same exception.

17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatException: For input string: "xyz"

Exception in thread "main" java.lang.NumberFormatException: For input string: "xyz"

at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

at java.lang.Long.parseLong(Long.java:589)

at java.lang.Long.(Long.java:965)

at com.stackify.example.TestExceptionHandling.logAndThrowException(TestExceptionHandling.java:63)

at com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:58)

The additional messages also don’t add any information. As explained in best practice #4, the exception message should describe the exceptional event. And the stack trace tells you in which class, method, and line the exception was thrown.

If you need to add additional information, you should catch the exception and wrap it in a custom one. But make sure to follow best practice number 9.

public void wrapException(String input) throws MyBusinessException {

try {

// do something

} catch (NumberFormatException e) {

throw new MyBusinessException("A message that describes the error.", e);

}

}

So, only catch an exception if you want to handle it. Otherwise, specify it in the method signature and let the caller take care of it.

**9. Wrap the Exception Without Consuming it**

It’s sometimes better to catch a standard exception and to wrap it into a custom one. A typical example for such an exception is an application or framework specific business exception. That allows you to add additional information and you can also implement a special handling for your exception class.

When you do that, make sure to set the original exception as the cause. The *Exception* class provides specific constructor methods that accept a *Throwable* as a parameter. Otherwise, you lose the stack trace and message of the original exception which will make it difficult to analyze the exceptional event that caused your exception.

public void wrapException(String input) throws MyBusinessException {

try {

// do something

} catch (NumberFormatException e) {

throw new MyBusinessException("A message that describes the error.", e);

}

}

**Summary**

As you’ve seen, there are lots of different things you should consider when you throw or catch an exception. Most of them have the goal to improve the readability of your code or the usability of your API.

Exceptions are most often an error handling mechanism and a communication medium at the same time. You should, therefore, make sure to discuss the best practices and rules you want to apply with your coworkers so that everyone understands the general concepts and uses them in the same way.