


Top 20 Java Exception Handling Best Practices

 howtodoinjava.com/best-practices/java-exception-handling-best-practices

This post is another addition in **best practices** series available in this blog. In this post, I am covering some well-known and some little known practices which you must consider while handling exceptions in your next java programming assignment. Follow this link to read more about **exception handling** in java.

Table of Contents

[Type of exceptions](#)

[User defined custom exceptions](#)

Best practices you must consider and follow

[Never swallow the exception in catch block](#)

[Declare the specific checked exceptions that your method can throw](#)

[Do not catch the Exception class rather catch specific sub classes](#)

[Never catch Throwable class](#)

[Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost](#)

[Either log the exception or throw it but never do the both](#)

[Never throw any exception from finally block](#)

[Always catch only those exceptions that you can actually handle](#)

[Don't use printStackTrace\(\) statement or similar methods](#)

[Use finally blocks instead of catch blocks if you are not going to handle exception](#)

[Remember "Throw early catch late" principle](#)

[Always clean up after handling the exception](#)

[Throw only relevant exception from a method](#)

[Never use exceptions for flow control in your program](#)

[Validate user input to catch adverse conditions very early in request processing](#)

[Always include all information about an exception in single log message](#)

[Pass all relevant information to exceptions to make them informative as much as possible](#)

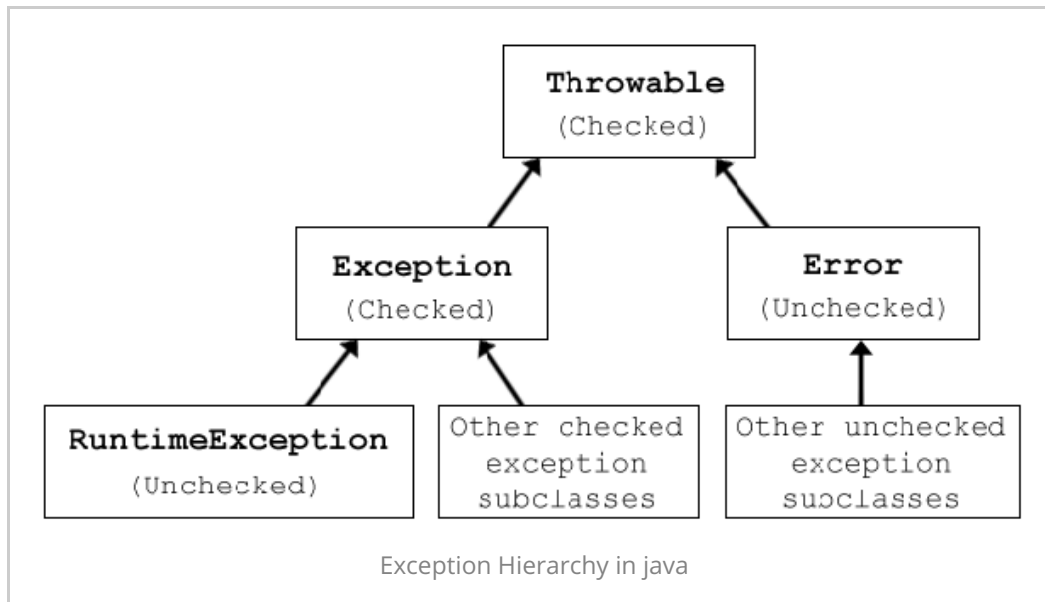
[Always terminate the thread which it is interrupted](#)

[Use template methods for repeated try-catch](#)

[Document all exceptions in your application in javadoc](#)

Before we dive into deep concepts of exception handling best practices, let's start with one of the most important concepts which is to understand that there are three general types of throwable classes in Java: checked exceptions, unchecked exceptions, and errors.

1. Type of exceptions in Java



Checked exceptions

These are exceptions that must be declared in the throws clause of a method. They extend **Exception** and are intended to be an “in your face” type of exceptions. Java wants you to handle them because they somehow are dependent on external factors outside your program. A checked exception indicates an expected problem that can occur during normal system operation. Mostly these exception happen when you try to use external systems over network or in file system. Mostly, the correct response to a checked exception should be to try again later, or to prompt the user to modify his input.

Unchecked exceptions

These are exceptions that do not need to be declared in a throws clause. JVM simply doesn't force you to handle them as they are mostly generated at runtime due to programmatic errors. They extend **RuntimeException**. The most common example is a **NullPointerException** [Quite scary.. Isn't it?]. An unchecked exception probably shouldn't be retried, and the correct action should be usually to do nothing, and let it come out of your method and through the execution stack. At a high level of execution, this type of exceptions should be logged.

Errors

are serious runtime environment problems that are almost certainly not recoverable. Some examples are **OutOfMemoryError**, **LinkageError**, and **StackOverflowError**. They generally crash your program or part of program. Only a good logging practice will help you in determining the exact causes of errors.

2. User defined custom exceptions

Anytime when user feels that he wants to use its own application specific exception for

some reasons, he can create a new class extending appropriate super class (mostly its `Exception`) and start using it in appropriate places. These user defined exceptions can be used in two ways:

1. Either directly throw the custom exception when something goes wrong in application

```
throw new DaoObjectNotFoundException("Couldn't find dao with id " + id);
```

2. Or wrap the original exception inside custom exception and throw it

```
catch (NoSuchMethodException e) {  
    throw new DaoObjectNotFoundException("Couldn't find dao with id " + id, e);  
}
```

Wrapping an exception can provide extra information to the user by adding your own message/ context information, while still preserving the stack trace and message of the original exception. It also allows you to hide the implementation details of your code, which is the most important reason to wrap exceptions.

Now lets start exploring the best practices followed for exception handling industry wise.

3. Java exception handling best practices you must consider and follow

3.1. Never swallow the exception in catch block

```
catch (NoSuchMethodException e) {  
    return null ;  
}
```

Doing this not only return “null” instead of handling or re-throwing the exception, it totally swallows the exception, losing the cause of error forever. And when you don't know the reason of failure, how you would prevent it in future? Never do this !!

3.2. Declare the specific checked exceptions that your method can throw

```
public void foo() throws Exception {  
}
```

Always avoid doing this as in above code sample. It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

```
public void foo() throws SpecificException1, SpecificException2 {  
}
```

3.3. Do not catch the Exception class rather catch specific sub classes

```
try {  
    someMethod();  
} catch (Exception e) {  
    LOGGER.error( "method has failed" , e);  
}
```

The problem with catching Exception is that if the method you are calling later adds a new checked exception to its method signature, the developer's intent is that you should handle the specific new exception. If your code just catches Exception (or Throwable), you'll never know about the change and the fact that your code is now wrong and might break at any point of time in runtime.

3.4. Never catch Throwable class

Well, its one step more serious trouble. Because java errors are also subclasses of the Throwable. Errors are irreversible conditions that can not be handled by JVM itself. And for some JVM implementations, JVM might not actually even invoke your catch clause on an Error.

3.5. Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost

```
catch (NoSuchMethodException e) {  
    throw new MyServiceException( "Some information: " + e.getMessage());  
}
```

This destroys the stack trace of the original exception, and is always wrong. The correct way of doing this is:

```
catch (NoSuchMethodException e) {  
    throw new MyServiceException( "Some information: " , e);  
}
```

3.6. Either log the exception or throw it but never do the both

```
catch (NoSuchMethodException e) {  
    LOGGER.error( "Some information" , e);  
    throw e;  
}
```

As in above example code, logging and throwing will result in multiple log messages in log files, for a single problem in the code, and makes life hell for the engineer who is trying to dig through the logs.

3.7. Never throw any exception from finally block

```
try {  
    someMethod();  
} finally {  
    cleanUp();  
}
```

This is fine, as long as `cleanUp()` can never throw any exception. In the above example, if `someMethod()` throws an exception, and in the finally block also, `cleanUp()` throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever. If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

3.8. Always catch only those exceptions that you can actually handle

```
catch (NoSuchMethodException e) {  
    throw e;  
}
```

Well this is most important concept. Don't catch any exception just for the sake of catching it. Catch any exception only if you want to handle it or, you want to provide additional contextual information in that exception. If you can't handle it in catch block, then best advice is just don't catch it only to re-throw it.

3.9. Don't use `printStackTrace()` statement or similar methods

Never leave `printStackTrace()` after finishing your code. Chances are one of your fellow colleague will get one of those stack traces eventually, and have exactly zero knowledge as to what to do with it because it will not have any contextual information appended to it.

3.10. Use finally blocks instead of catch blocks if you are not going to handle exception

```
try {  
    someMethod();  
} finally {  
    cleanUp();  
}
```

This is also a good practice. If inside your method you are accessing some method 2, and method 2 throw some exception which you do not want to handle in method 1, but still want some cleanup in case exception occur, then do this cleanup in finally block. Do not use catch block.

3.11. Remember “Throw early catch late” principle

This is probably the most famous principle about Exception handling. It basically says that you should throw an exception as soon as you can, and catch it late as much as possible. You should wait until you have all the information to handle it properly.

This principle implicitly says that you will be more likely to throw it in the low-level methods, where you will be checking if single values are null or not appropriate. And you will be making the exception climb the stack trace for quite several levels until you reach a sufficient level of abstraction to be able to handle the problem.

3.12. Always clean up after handling the exception

If you are using resources like database connections or network connections, make sure you clean them up. If the API you are invoking uses only unchecked exceptions, you should still clean up resources after use, with try – finally blocks. Inside try block access the resource and inside finally close the resource. Even if any exception occur in accessing the resource, then also resource will be closed gracefully.

3.13. Throw only relevant exception from a method

Relevancy is important to keep application clean. A method which tries to read a file; if throws `NullPointerException` then it will not give any relevant information to user. Instead it will be better if such exception is wrapped inside custom exception e.g. `NoSuchFileFoundException` then it will be more useful for users of that method.

3.14. Never use exceptions for flow control in your program

We have read it many times but sometimes we keep seeing code in our project where developer tries to use exceptions for application logic. Never do that. It makes code hard to read, understand and ugly.

3.15. Validate user input to catch adverse conditions very early in request processing

Always validate user input in very early stage, even before it reached to actual controller. It will help you to minimize the exception handling code in your core application logic. It also helps you in making application consistent if there is some error in user input.

For example: If in user registration application, you are following below logic:

- 1) Validate User
- 2) Insert User
- 3) Validate address

- 4) Insert address
- 5) If problem the Rollback everything

This is very incorrect approach. It can leave you database in inconsistent state in various scenarios. Rather validate everything in first place and then take the user data in dao layer and make DB updates. Correct approach is:

- 1) Validate User
- 2) Validate address
- 3) Insert User
- 4) Insert address
- 5) If problem the Rollback everything

3.16. Always include all information about an exception in single log message

```
LOGGER.debug( "Using cache sector A" );  
LOGGER.debug( "Using retry sector B" );
```

Don't do this.

Using a multi-line log message with multiple calls to `LOGGER.debug()` may look fine in your test case, but when it shows up in the log file of an app server with 400 threads running in parallel, all dumping information to the same log file, your two log messages may end up spaced out 1000 lines apart in the log file, even though they occur on subsequent lines in your code.

Do it like this:

```
LOGGER.debug( "Using cache sector A, using retry sector B" );
```

3.17. Pass all relevant information to exceptions to make them informative as much as possible

This is also very important to make exception messages and stack traces useful and informative. What is the use of a log, if you are not able to determine anything out of it. These type of logs just exist in your code for decoration purpose.

3.18. Always terminate the thread which it is interrupted

```
while ( true ) {  
    try {  
        Thread.sleep( 100000 );  
    } catch (InterruptedException e) {}  
    doSomethingCool();  
}
```

InterruptedException is a clue to your code that it should stop whatever it's doing. Some common use cases for a thread getting interrupted are the active transaction timing out, or a thread pool getting shut down. Instead of ignoring the InterruptedException, your code should do its best to finish up what it's doing, and finish the current thread of execution. So to correct the example above:

```
while ( true ) {  
    try {  
        Thread.sleep( 100000 );  
    } catch (InterruptedException e) {  
        break ;  
    }  
    doSomethingCool();  
}
```

3.19. Use template methods for repeated try-catch

There is no use of having a similar catch block in 100 places in your code. It increases code duplicity which does not help anything. Use template methods for such cases.

For example below code tries to close a database connection.

```
class DBUtil{  
    public static void closeConnection(Connection conn){  
        try {  
            conn.close();  
        } catch (Exception ex){  
            }  
    }  
}
```

This type of method will be used in thousands of places in your application. Don't put whole code in every place rather define above method and use it everywhere like below:

```
public void dataAccessCode() {  
    Connection conn = null ;  
    try {  
        conn = getConnection();  
        ....  
    } finally {  
        DBUtil.closeConnection(conn);  
    }  
}
```

3.20. Document all exceptions in the application with javadoc

Make it a practice to javadoc all exceptions which a piece of code may throw at runtime. Also try to include possible course of action, user should follow in case these exception occur.

That's all i have in my mind for now related to Java exception handling best practices. If you found anything missing or you does not relate to my view on any point, drop me a

comment. I will be happy to discuss.

Happy Learning !!