EXCEPTION HANDLING

BEST PRACTICES

LEMİ ORHAN ERGİN

@lemiorhan lemiorhanergin.com @lemiorhan

1 USE CHECKED EXCEPTION FOR RECOVERABLE ERROR & UNCHECKED EXCEPTION FOR PROGRAMMING ERROR

Checked exceptions ensures that you provide exception handling code for error conditions, which is a way from language to enforcing you for writing robust code, but same time it also add lots of clutter into code and makes it unreadable. Also, it seems reasonable to catch exception and do something if you have alternatives or recovery strategies.

2

AVOID OVERUSING CHECKED EXCEPTION

CATCH BLOCK WITH MULTIPLE EXCEPTIONS

JAVA7

```
catch (IOException|SQLException ex) {
    logger.log(ex);
}
```

AUTOMATIC RESOURCE MANAGEMENT WITH TRY-WITH-RESOURCES

JAVA7

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Checked Exception has there advantage in terms of enforcement, but at same time it also litters the code and makes it unreadable by obscuring business logic. You can minimize this by not overusing checked Exception which result in much cleaner code. You can also use newer Java 7 features like “one catch block for multiple exceptions” and “automatic resource management”, to remove some duplication.

3

CONVERTING CHECKED EXCEPTION INTO RUNTIMEEXCEPTION

```
try {
    riskyOperation();
} catch (IOException ioe) {
    throw new CustomRuntimeException(ioe);
}
```

This is one of the technique used to limit use of checked Exception in many of frameworks like Spring ,where most of checked Exception, which stem from JDBC is wrapped into DataAccessException, an unchecked Exception. This Java best practice provides benefits, in terms of restricting specific exception into specific modules, like SQLException into DAO layer and throwing meaningful RuntimeException to client layer.

4

REMEMBER EXCEPTIONS ARE COSTLY IN TERMS OF PERFORMANCE

One thing which is worth remembering is that Exceptions are costly, and can slow down your code. Suppose you have method which is reading from ResultSet and often throws SQLException than move to next element, will be much slower than normal code which doesn't throw that Exception. So minimizing catching unnecessary Exception and moving on, without fixing there root cause. Don’t just throw and catch exceptions, if you can use boolean variable to indicate result of operation, which may result in cleaner and performance solution. Avoid unnecessary Exception handling by fixing root cause.

5 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 !!

6 DECLARE THE SPECIFIC CHECKED EXCEPTIONS THAT YOUR METHOD CAN THROW

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

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

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.

7

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.

8

NEVER CATCH THROWABLE CLASS

```
try {
    someMethod();
} catch (Throwable t) {
    // handle throwable
}
```

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

9 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());
}
```

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

Incorrect way of wrapping exceptions destroys the stack trace of the original exception, and is always wrong.

10

EITHER LOG THE EXCEPTION OR THROW IT BUT NEVER DO THE BOTH

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

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.

11 NEVER THROW ANY EXCEPTION FROM FINALLY BLOCK

```
try {
  // Throws exceptionOne
  someMethod();
} finally {
  // If finally also threw any exception,
  // the exceptionOne will be lost forever
  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.

12

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.

13 DON'T USE PRINTSTACKTRACE() STATEMENT OR SIMILAR METHODS

```
catch (NoSuchMethodException e) {
  System.out.println(e.getStackTrace());
}
```

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.

14 USE FINALLY BLOCKS INSTEAD OF CATCH BLOCKS IF YOU ARE NOT GOING TO HANDLE EXCEPTION

```
try {
   someMethod();
} finally {
   cleanUp(); //do cleanup here
}
```

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

15 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.

16 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.

You can use new features Java7 to run auto-cleanup via try-with-resources statement.

17

EXCEPTION NAMES MUST BE CLEAR AND MEANINGFUL

Name your checked exceptions stating the cause of the exception. You can have your own exception hierarchy by extending current Exception class. But for specific errors, throw an exception like “AccountLockedException” instead of “AccountException” to be more specific.

18

THROW EXCEPTIONS FOR ERROR CONDITIONS WHILE IMPLEMENTING A METHOD

```
public void someMethod() {
  // on error 1
  return -1;
  // on error 2
  return -2;
}
```

If you return -1, -2, -3 etc. values instead of FileNotFoundException, that method can not be understand. Use exceptions on errors.

19 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.

20

NEVER USE EXCEPTIONS FOR FLOW CONTROL

Never do that. It makes code hard to read, hard to understand and makes it ugly.

21

NEVER USE EXCEPTIONS FOR FLOW CONTROL IN YOUR PROGRAM

```
try {
  // do some logic
  throw new OperationException();
} catch (OperationException e) {
  // log the exception message
  // or throw a new exception
}
```

It is useless to catch the exception you throw in the try block. Do not manage business logic with exceptions. Use conditional statements instead.

22

ONE TRY BLOCK MUST EXIST FOR ONE BASIC OPERATION

Granularity is very important. One try block must exist for one basic operation. So don't put hundreds of lines in a try-catch statement.

23

DO NOT HANDLE EXCEPTIONS INSIDE LOOPS

```
for (Message message:messageList) {
  try {
    // do something that can throw ex.
  } catch (SomeException e) {
    // handle exception
  }
}
```

```
try {
  for (Message message:messageList) {
    // do something that can throw ex.
  }
} catch (SomeException e) {
  // handle exception
}
```

Exception handling inside a loop is not recommended for most cases. Surround the loop with exception block instead.

24

ALWAYS INCLUDE ALL INFORMATION ABOUT AN EXCEPTION IN SINGLE LOG MESSAGE

```
try {
 someMethod();
} catch (OperationException e) {
  LOGGER.debug(“some message”);
  // handle exception
  LOGGER.debug(“some another message”);
}
```

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.

25 PASS ALL RELEVANT INFORMATION TO EXCEPTIONS TO MAKE THEM INFORMATIVE AS MUCH AS POSSIBLE

```
catch (SomeException e) {
    logger.log(“error occurred”, e);
}
```

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.

26

ALWAYS TERMINATE THE THREAD WHICH IT IS INTERRUPTED

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

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

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.

27 USE TEMPLATE METHODS FOR REPEATED TRY-CATCH

```
class DBUtil{
    public static void closeConnection(Connection conn){
        try{
            conn.close();
        } catch(SQLException ex){
            throw new RuntimeException("Cannot close connection", ex);
        }
    }
}
```

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

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.

28

DOCUMENT ALL EXCEPTIONS IN YOUR APPLICATION IN 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.

29

CATCH ALL EXCEPTIONS BEFORE THEY REACH UP TO THE UI

You have to catch all exceptions before they reach up to the UI and make your user sad. This means on the "highest level" you want to catch anything that happened further down. Then you can let the user know there was a problem and at the same time take measures to inform the developers, like sending out alarm mails or whatever

REFERENCES

JAVA EXCEPTION HANDLING BEST PRACTICES BY LOKESH GUPTA

http://howtodoinjava.com/2013/04/04/java-exception-handling-best-practices/

15 BEST PRACTICES FOR EXCEPTION HANDLING BY CAGDAS BASARANER

http://codebuild.blogspot.co.uk/2012/01/15-best-practices-about-exception.html

10 EXCEPTION HANDLING BEST PRACTICES IN JAVA PROGRAMMING BY JAVIN PAUL

http://javarevisited.blogspot.co.uk/2013/03/0-exception-handling-best-practices-in-Java-Programming.html

@lemiorhan

@lemiorhan

@lemiorhan

agilistanbul.com

lemiorhanergin.com

LEMİ ORHAN ERGİN

lemiorhan@agilistanbul.com

Founder & Author @ agilistanbul.com