Top 20 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

are serious runtime environment problems that are almost certainly not recoverable. Some examples are OutOfMemoryError, LinkageError, and StackOverflowError. They generally crash you 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 { //Incorrect
}
```

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, Spe
}
```

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 informatio
}
```

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 informatio
}
```

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 {
```

Before we dive into deep concepts of exception handling best practices, lets 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

Exception Hierarchy 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

```
  someMethod();  //Throws exceptionOne
} finally {
  cleanUp();    //If finally also threw any exce
}
```

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; //Avoid this as it doesn't help anyt
}
```

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();  //Method 2
} finally {
  cleanUp();    //do cleanup here
```

```
}
```

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
```

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) {} //Don't do
  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(Connectio
        try{
            conn.close();
        } catch(Exception ex){
            //Log Exception - Cannot close conne
        }
    }
}
```

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 !!

Was this post helpful?

YES NO

About Lokesh Gupta

A family guy with fun loving nature. Love computers, programming and solving everyday problems. Find me on Facebook and Twitter.

Feedback, Discussion and Comments

Erika Smith

October 16, 2019

Hello,

This blog is java exception handing best practices assignment help and benefits information. thank you for share this blog.

Phuong Nguyen Ho

July 24, 2019

Thank you! A very nice article for newbie. (Like me! :D)

CesarPonce

December 19, 2018

Nice article.

Question you missed;

Use “try-catch” block surrounding only the line that throws the Exception, when?

Use “try-catch” block surrounding more than only the line that throws the Exception, when?

Thanks.

Azhagusurya

August 10, 2017

Most valuable post ever .you help me a lot through this post really … thanks a lot ….

Tushar

May 25, 2017

Thanks a lot. Very informative article.

Lucas

January 2, 2017

Hi,

My app code is analysed by Sonar and this one is showing Major issues when a public methods throw several checked exceptions.

The message shown is: “Refactor this method to throw at most one checked exception instead of: MyException1, MyException2, etc…”

And sonar’s explanation is: “Using checked exceptions forces method callers to deal with errors, either by propagating them or by handling them.

This makes those exceptions fully part of the API of the method.

To keep the complexity for callers reasonable, methods should not throw more than one kind of checked exception.”

This sonar tip conflicts with your point no 2 in this article.

So it is a bit confusing. What is the best way of doing so?

Should we throw only one kinf of exception embedding the real exception in his cause?

Best regards,

Lucas

> Lokesh Gupta
>
> January 2, 2017
>
> Point 2 ONLY talk about being specific (rather than being generic) which is preferred way.
>
> Sonar follows fixed set of static rules – and does not understand the context. You as developer is the most suitable person to decide the behavior.

Thalisson

August 13, 2019

Also, you can apply the point 5 for this kind of rules, which still make point 2 as valid.

Himesh

August 7, 2016

Very informative article. Keep Posting..)

Ankur

June 14, 2016

Hi Lokesh,

Thanks for the blog it's all gold dust.

I have query when you say "8) Always catch only those exceptions that you can actually handle" what if I want log some info that will help or add customize the message to add some more details.

Would not be a good idea to catch customize the message an and re throw it?

> Lokesh Gupta
>
> June 15, 2016
>
> Yes, you can do it this as well. A little more context information is always helpful.

Ankur

June 15, 2016

Thanks Lokesh.

Rajeev

April 6, 2016

First of all I would like to thank you for such a nice article. I have a small doubt i.e in the 7th point you written that never throw an exception from finally block but in the 19th point you used a try-catch template in finally block which is throwing an exception which means that finally block is throwing the exception. Please make me understand in this regard.

> Lokesh Gupta
>
> April 6, 2016
>
> Updated point – 19. It's was wrong example. If there is an error while closing the exception, then you cannot anything about it. So log it for reference.

Rajeev

April 6, 2016

okay. Can you please make similar post about threads(like multiple threads of same object on methods, multiple threads of same object on static methods, multiple threads of different objects on methods , multiple threads of different objects on static methods)

> Lokesh Gupta
>
> April 6, 2016
>
> I will find time.

Kushal

April 15, 2015

I was asked in an interview say we have a class a where v have try catch block with exception handling..

We call this method from other class ..how to show the exception and message in current class.

```
Class A{
void somemethod(){
try{
// some code
}catch(Exception e){
String s= "Error is "+e;
}

Class B{
A a = new A();
a.some method();
//how to display the string error here
}
```

Lokesh Gupta

April 15, 2015

1) Set the error message in e in class A.

2) Wrap a.someMethod() in try catch block in class B. and in catch block, print the exception in logger/console.

adarshs241@gmail.com

March 10, 2015

Hi ,

Can you please explain below code,is that right ?

protected String someMethod(String className)

{

String packageName = “”;

try

{

//Some Code

}

catch (Throwable t)

{

packageName = “”;

}

return packageName;

}

Lokesh Gupta

March 15, 2015

Syntactically code is correct. No compilation issues.

Other than above two debatable issues : Method parameter “className” is not used. And catch block used for controlling application logic. Both are debatable.

Srinu

November 10, 2014

Nice article Sir!!

Keith

October 4, 2014

You just broke your own rule (7) Never throw any exception from finally block) only a few lines later – you said not to throw exceptions from the finally clause. Then you did it with this:

finally{

DBUtil.closeConnection(conn);

// which equals this:

try{

conn.close();

} catch(SQLException ex){

throw new RuntimeException("Cannot close connection", ex);

// and voila – you are throwing an exception from a finally clause – lol.

}

}

Lokesh Gupta

October 5, 2014

Good catch. Though the example I taken is broken, idea still stand relevant. Thanks for pointing out.

Tom

October 5, 2014

Most correctly DBUtil.closeConnection() should *log and absorb* exceptions. There is no recovery possible for failure on close, but neither does it have any impact on subsequent higher-level/ business logic.

This is an illustration that exception-handling should depend on A) whether it affects the client code, B) whether there is any contingency the client can specifically recover from, or C) whether it's just an outright failure (not recoverable in any smart way).

Since connections being closed are not going to be used any further anyway, the answer to A) is "no" — there is no need to throw further, and exceptions can be logged & swallowed.

In this situation, such practice is entirely correct — and gives better reliability. "Hard failures" which wouldn't actually affect the business level are mitigated to "soft failures", if the connections are repeatedly breaking eventually service/ connection pool may degrade but this softens & delays failure.

Apache commons-io provides a good example of this.

IOUtils.closeQuietly() closes files/streams etc quietly and, since exceptions on closure are no longer relevant to client functionality, it handles any IOException thrown by “absorbing”.

I prefer to log such exceptions & absorb them, personally, but “close” is one of the few correct situations where exceptions can be sent to a sink.

Bhaskar Verma

August 21, 2014

Hi Sir,Nice Post.Sir please tell me,how to handle exception in web-application of struts-spring-Hibernate. eg-Suppose if exception has come in DAO layer of application,how to handle this exception. Please give with proper example.

Lokesh Gupta

August 21, 2014

There is no hard and fast rule for handling exceptions in any specific way. In struts-spring-hibernate project or any other project, you need to identify that

1) Which exceptions you want to handle and which one you want to pass on to framework?

2) How the application must behave in event of any certain exception?

Above two questions will guide you everytime you have any confusion.

Bhaskar Verma

August 21, 2014

Sir,Good suggestion.

Can you please provide some sample code for this.Suppose in DAO layer the data is not available,then where i have to handle the exception in DAO layer or pass on to Service Layer for handling.

Lokesh Gupta

August 21, 2014

Unfortunately that also depends on scenario you are facing. E.g. In a search page, if the record does not exist then you want to let ue know that record does not exist. If he typed something wrong which resulted in exception then also you want him to notify.

But if there is exception during query execution e.g. timeout; then you would not like to show it to user, rather you will re-try it immediately or try any other data source possibly.

Again above arguments are completely fiction based 🙂 and they change as per need of the time.

Koushik Paul

July 8, 2014

Finally a good post, 99% of what u get from google is very basic information which only helps u to do some certification, Rather this one is more in depth which might help u in day to day use . Please let me know if you have something for collection too.

Tom @ Literate Java

May 31, 2014

Good article. Runtime versus “checked exceptions” is also an important question, for developers.

Runtime exceptions are widely preferred by Spring, Hibernate and modern frameworks. Java 8's lambdas also favour them.

Checked exceptions have always been a controversial "feature" of Java. They were originally intended to highlight contingencies, but have always been incompatible with best-practice "throw early, catch late" exception-handling/ and FP functional programming constructs.

http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/

Maddy

March 4, 2014

Thanks Lokesh for such nice text…

Here is one more blog targeting "Exception Handling in Threading Environment", if someone interested can read…

Exceptions handling of single-threaded and multi-threaded programs are different from each other. When single-threaded program terminates due to uncaught exception, it stop running and print exception stack trace on console. In other hand, when multi-threaded program encounter with uncaught exception, the stack trace may be printed, but no one may be watch that console. Because when that thread fail, application may appear continuously working. Therefore, there might be change that its failure may go unnoticed.

Read more from here

satish

February 12, 2014

Thanks a lot

Bala

November 22, 2013

Thank you. Much more useful

khoaphamdl

November 14, 2013

It’s really useful. Thanks

phenix

September 28, 2013

The example in 19 is bad chosen, as it conflicts with 7.

> Lokesh Gupta
>
> September 28, 2013
>
> You are right. I agree. But sometimes it happens when you try to make any point you actually mess with another. Anyways, logically both are correct if read independently.

JavaIsKing

August 31, 2013

Good list to memorize for all devs, even i, as an experienced dev, learnt a new practice.

mm

July 10, 2013

At point 19 you recommend using template methods. But your example is missing a real template method. Template method means in pseudo code:

public abstract class SuperClass {

public void importantMethod() {

// do something before

this.theTemplateMethod();

// do something after

}

public abstract void theTemplateMethod();

}

public class SubClass extends SuperClass {

private Knowledge knowledge;

public void theTemplateMethod() {

// special things we can only do in subClass,

// because only the subClass has some special knowledge.

}

}

Instead your code shows delegation of closing a connection to a class named DBUtil, which better should be called ConnectionService or ConnectionCloseService. So in turn you really describe a “delegation of closing a connection to a service to hide exception handling”.

Vijayakumar Ramdoss

June 6, 2013

good work

Adrià Figuera Puig (@adriafiguera)

April 12, 2013

Thank you! great post!!!

Comments are closed on this article!

Meta Links

About Me
Contact Us
Privacy policy
Advertise
Guest and Sponsored Posts

Recommended Reading

10 Life Lessons
Secure Hash Algorithms
How Web Servers work?
How Java I/O Works Internally?
Best Way to Learn Java
Java Best Practices Guide
Microservices Tutorial
REST API Tutorial
How to Start New Blog