## 10 Exception Handling Java Best Practices

Here is my collection of 10 Java best practices to write Exception handling code in Java. There have been both applause and criticism of checked Exception in Java, which is a language feature to force dealing with Exceptions. In this article, we will look to minimize use of checked Exception and learn when to use checked vs unchecked exceptions in Java as well.

**1) Use Checked Exception for Recoverable error and Unchecked Exception for programming error.**

Choosing between checked and unchecked exception is always been confusing for Java programmers. 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.  See checked vs unchecked exceptions for more information on choosing between checked and Runtime Exception in Java.

**2) Close or release resource in finally block**

This is a well-known best practice in Java and quite a standard, while dealing with networking and IO classes. Closing resources in finally block guarantees that precious and scarce resource released properly in case of normal and aborted execution, guaranteed by finally block. From Java 7, language has a more interesting automatic resource management or ARM blocks, which can do this for you. Nevertheless, always remember to close resources in finally block, which is important to release limited resources like FileDescriptors, used in case of both socket and files.

**3) Including cause of Exception in stack-trace**

Many times Java library and open source code wraps one Exception into another, when one exception is thrown due to result of another exception. Its become extremely important to log or print cause of root exception. Java Exception class provides getCause() method to retrieve cause which can be used to provide more information about root cause of Exception. This Java best practice helps a lot while debugging or troubleshooting an issue. Always remember to pass original Exception, into constructor of new Exception, if you are wrapping one exception into another.

**4) Always provide meaning full message on Exception**

message of Exception is the most important place, where you can point out cause of problem because this is the first place every programmer looks upon. Always try to provide precise and factual information here. For example, compare these two Exception messages for IllegalArgumentException :

message 1: "Incorrect argument for method"

message 2: "Illegal value for ${argument}: ${value}

first one just says that argument is illegal or incorrect, but second one include both name of argument and its illegal value which is important to point out cause of error. Always follow this *Java best practice*, when writing code for handling exceptions and errors in Java.

**5) Avoid overusing Checked Exception**

Checked Exception has their 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.

**6) Converting Checked Exception into RuntimeException**

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.

**7) 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.

**8) Avoid empty catch blocks**

Nothing is more worse than empty catch block, because it not just hides the Errors and Exception, but also may leave your object in unusable or corrupt state. Empty catch block only make sense, if you absolutely sure that Exception is not going to affect object state on any ways, but still its better to log any error comes during program execution. This is not a Java best practice, but a most common practice, while writing Exception handling code in Java.

**9) Use Standard Exceptions**

Our ninth Java best practice advise on using standard and inbuilt Java Exceptions. Using standard Exception instead of creating own Exception every now and then is much better in terms of maintenance and consistency. Reusing standard exception makes code more readable, because most of  Java developers are familiar with standard RuntimeException from JDK like, IllegalStateException,IllegalArgumentException or NullPointerException, and they will immediately be able to know purpose of Exception, instead of looking out another place on code or docs to find out purpose of user defined Exceptions.

**10) Document Exception thrown by any method**

Java provides throw and throws keyword to throw exception and in javadoc you have @throw to document possible Exception thrown by any method. This becomes increasingly important if you are writing API or public interface. With proper documentation of Exception thrown by any method you can potentially alert anyone who is using it.

That's all on Java best practices to follow while handing Exceptions in Java. Do let us know what practices you follow while writing Exception handling code in Java.

Nine out of ten are items in "Effective Java, Second Edition." If that's not just coincidence, if you were inspired by that book, it seems fair to give the authors credit.

@Anonymous, If that's the case, then it just a coincidence mate. Though I did read the book, and I am big fan of him, this post was mainly a collection of best practices, what I follow. To be fair, every one with few years of experience in Java knows about it, it's more Java programmer with couple of years experience. Having said, that I highly recommend Effective Java to any Java developer, you can also read it about in my post of 5 good books for Java Programmer

Hi Javin, Can you please share tips to write robust code, I mean code which handles exceptional scenario gracefully e.g. reading a malformed message if you are expecting a fixed length binary message, or an XML which doesn't has required XPATH . Most importantly how to find those exceptional scenario and how to decide action needs to be taken on that time?

Here are my 2 Cents on Exception best practices :  
  
1) Consider Making Business Exception class as Immutable, to avoid any accidental change on Error. Make them final class, with all private final fields, initialized by constructor arguments.  
  
2) Never Ever show Exception stacktrace in web pages, this is particularly important for Java developer, who writes JSP pages. Instead of showing Exception stacktrace to client, Use Global exception handler to show 404 page.  
  
3) While writing your own Exception classes, both checked and unchecked, make sure you have provided methods to get exception, message, error code, which is required by clients for effective exception handling. Failing to do so may result in Client parsing error message and extracting relevant information, which results in ugly, brittle code.  
  
Anyway Good list, Best practices are worth reading. Cheers

Please give some example for better understanding

What about the avoiding "try/catch(Exception)" blocks?  
AKA pokemon exception handling. you gotta catch 'em all.

I want to add some more information to the point 7.  
Exceptions in java are slow because of the time taken to fill the stack trace to the exception object. Time taken for Creating an exception object , Throwing an object and Catching an object is comparatively less compared to filling the stack trace to the exception object. We can avoid filling stack trace by creating an custom exception which overrides fillInStackTrace method of the Throwable class . These are Cheap Exceptions and can be used for flow control(although not advised) in some particular situations.

Hello manju, I agree exceptions are not cheap and should not be used for flow control, but your tips sounds interesting :)