### Exception Handling in Java

Java is the first OO language to force handling exceptions by way of checked exceptions. It is also possible to encounter unchecked or RuntimeExceptions in Java which do not explicitly outline the contract between a method and its callers. Both Checked and Unchecked exceptions extend from base Exception class in java.lang package. There is a big debate in Java community which one to use. Some developers prefer checked exceptions and others unchecked exceptions. In this document, I will outline some of the issues relating to exceptions handling and recommend some common guidelines for handling exceptions in your application. I will also briefly touch the new features of exception handling in Java 7.

**Checked vs. Unchecked Exceptions**

It is a good practice to use checked exceptions for all errors the application can recover from and unchecked exceptions for the errors the application cannot recover from. Below is a list of differences between checked and unchecked exceptions.

* Compiler will force you to properly handle the checked exceptions which make it harder to forget handling that exception. Unchecked exceptions make it easier to forget handling errors since the compiler does not force the developer to catch exceptions; therefore, most of the time unchecked exceptions are rarely caught by the caller.
* With checked exceptions, we have try-catch block to remind us to add a finally clause. With unchecked exceptions, we might not remember to add the try-finally block to perform resource cleanup.
* Javadoc automatically generates basic documentation for all exceptions in the throws clause. The documentation is generated with no extra effort on your part. But, this is not true with unchecked exceptions since they are not typically placed in the method's throws clause. If you decide to use unchecked exceptions, you need to document all the unchecked exceptions a method might throw.
* Unchecked runtime exceptions represent conditions that are occurred due to errors in your program's logic and cannot be reasonably recovered from run time. Checked exceptions represent invalid user input, database problems, network outages, files don’t exist in which the client application might recover.
* All exceptions are based on the Throwable class and all Throwables can have an underlying root cause by default. The root cause may be set in the Throwable's constructor or after construction by calling initCause. Having the root cause is very useful for troubleshooting. When a Throwable is passed over the network, it must first be serialized, and then deserialized. However, the root cause can cause a problem by forcing to throw NoClassDefFoundError if the root cause inside a Throwable object is not known to the receiver. This action will replace the original exception with something unrelated. One solution is to define a checked exception, which can hold only null as the root cause.
* Unchecked exceptions have the benefit of not forcing the client API to explicitly deal with them. They propagate to where you want to catch them, or they go all the way out and get reported. The Java API has many unchecked exceptions, such as NullPointerException and llegalArgumentException. You can use standard exceptions provided in Java rather than creating your own and this is considered as best practices. They can make the code easy to understand and avoid increasing the memory footprint of code.

**There are two basic issues surrounding exception handling:**

1. **When do you throw a RuntimeException Vs Checked Exception?**Checked Exceptions introduce tight coupling between a method and its direct and indirect callers. On the other hand, throwing runtime exceptions could result in unintended failures at the client level at runtime unless these exceptions are caught and handled by the caller code.
2. **How do we preserve Encapsulation?**When classes re-throw exceptions caught when calling methods on classes from other packages or components, the client code will have to deal with exceptions from packages that were not directly called resulting in incoherent code.

**Here are some known practices which should be considered while handling exceptions**

1. **Use what is appropriate**

Both checked and unchecked exceptions have advantages and disadvantages; therefore, use what is appropriate to your situation. Use Checked Exception for Recoverable error and Unchecked Exception for programming errors.

1. **Throw and catch exceptions appropriate to the abstraction.**Don’t propagate an exception thrown by a lower-level exception. For example, never let implementation specific checked exceptions escalate to the higher layers such as throwing a SQLException from data access code to the business objects layer: business objects layer do not need to know about SQLException.  
     
   Where possible, the best way to deal with lower-level exceptions is to avoid them by checking the validity of parameters before passing them on to lower layers. The next best approach is to have the higher layer silently work around these exceptions, insulating the caller of the higher-level method from lower-level problems.   
     
   If it isn’t feasible to prevent or handle exceptions from lower layers, then higher layers should catch lower-level exceptions, and in their place, throw exceptions that can be explained in terms of higher-level abstraction.

Exception chaining is appropriate in cases where lower-level exception might be helpful to someone debugging the problem that caused the higher level exception. This is done by passing the lower-level exception (“the cause”) to the higher level exception that provides an accessor method (Throwable.getCause) to retrieve the lower-level exception.

*Refine Exceptions* in order to derive more meaningful exceptions inside the package. This can be accomplished by sub-classing the base package exceptions to more meaningful exceptions. The exception class name should describe the exception it throws, not the thrower.

1. Always clean-up resources in finally block  
   If you are using resources like files, database connections or [network connections](http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html?page=2##), make sure you clean them up before allowing control to be passed to a different part of the system. If the API you are invoking uses only unchecked exceptions, you should still clean up resources after use, with try-finally blocks. Clean-up code should occur in the finally block because the system will always execute the code within a finally block regardless of what happens in the try block. Be sure that the code within the finally block doesn’t also throw an exception that might terminate the cleanup; another try-finally block may be needed.
2. **Do not ignore Exceptions by having empty catch blocks**When a method from an API throws a checked exception, it is trying to tell you that you should take some counter action. If the checked exception does not make sense to you, convert it into an unchecked exception and throw it again, but do not ignore it by catching it with empty catch block and let the application continue like the exception never occurred. If you must put an empty catch block, put a comment that makes the reasoning is behind for ignoring the exception.
3. **Either log the exception or throw it but never do the both**Logging and throwing will result in multiple log messages in log files. Several logs for same exception makes the logs hard to read and cause confusion.
4. **Do not use Exceptions for flow control**Use exception handling only in exceptional situations. Do not throw an exception to indicate a change. This makes the code difficult to read and also makes the code slower.
5. **Do not display stack trace**  
   It is usually considered bad practice to display a stack trace to end user. Exposing a stack trace may be a security risk.When one exception is thrown due to result of another exception, it is 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.
6. **Be as specific as possible**

Do not group together related exceptions in a generic exception class because you can lose important information. Always provide meaning full message on Exception provide precise and factual information.

1. **Avoid overusing Checked Exception**

Remember Exceptions are costly in terms of performance. Don’t just throw and catch exceptions since Exceptions are costly.

1. **Convert Checked Exception into RuntimeException**

You would not want to throw SQLException to the client from DAO layer. Instead you would catch the SQLException and throw more meaningful RuntimeException to client layer.

1. **Use Standard Exceptions**

Using standard Exception instead of creating your own Exception is much better in terms of maintenance and consistency. Also, using standard Exception makes code more readable since we all Java developers are familiar with standard exceptions.

1. **Document Exception thrown by any method**

With proper documentation of Exception thrown by any method you can alert anyone who is using it.

1. **Do not catch Throwable class**

Java errors are also subclasses of the Throwable. Errors are considered not recoverable conditions and it cannot even be handled by JVM itself. Always catch only those exceptions that you can actually handle.

1. **Do not throw exceptions from finally block**

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. If you do, you can lose the original exception from the try block.

1. **Practice Throw early catch late principle**

Throw an exception as soon as you can, and catch it late as much as possible. Validate user inputs in early stage to minimize the exception handling.

**If you are using Java 7, there is some good news. Java 7 provides better exception handling.**

There are two improvements to exception handling.

**1) Multi-catch:** You can catch multi exceptions type in one catch block.

**2) Final Rethrow:** Allows you to catch an exception type and its subtype and rethrow it without having to add a throws clause to the method signature.

try {

//some code here

} catch (final Throwable ex) {

//some more code here

throw ex;

}

In the above example using the **final** keyword allows you to throw an exception of the exact dynamic type that will be throwed. In the above code if an IllegalArgumentException occurs, an IllegalArgumentException will be thrown from the catch block.