BEST PRACTICES FOR USING AND WRITING EXCEPTIONS IN JAVA

This article lays down some of the best practices which you can use during your design or code reviews, and assumes that you are familiar with the basics of exceptions in Java.I have learned many of these best practices from Effective Java by Joshua Bloch, which tells how to use java to best effect

Best Practices for Using and Writing Exceptions in Java

1. **Exceptions should be only used for exceptional conditions**. It should not be used for control flow using techniques that might look very clever, like catching an exception and ending a loop.
2. To not force the programmers to use exceptions for control flow, **APIs must provide alternatives to exceptions to control the flow**. For example, if Iterator class in collections does not have a hasNext() method, we would have to use a while(true) loop and then catch NoSuchElementException to exit the loop.
3. **As alternatives** to exceptions for flow control, **we can have either state testing method** (like hasNext()) and then call the dependent method (like nextItem()) **or return a distinguished value** (such as null) when the object is in an inappropriate state.
4. Though **a state testing method is the preferred one**, if there is a chance for the state to change between a state testing method call (like hasNext()) and its dependent method (like nextItem()), or if a separate state testing method could duplicate the work of the state dependent method, we can use distinguished return type method.
5. **Catching a checked exception and ignoring it is** usually **a bad idea.** It is better to declare it using throws so that the caller of the method can handle it or the program will fail fast, than fail silently. Or at the very least, the catch block should have a comment explaining why it is appropriate to ignore.
6. **When using a distinguished return value, we need to make sure that the distinguished value is not a possible correct value** that can be returned.
7. While writing an exception, **if there is a chance of recovery, then use checked exception, else use unchecked exception**; and if in doubt, it would be better you use unchecked exception. **Overuse of checked exception can make an API far less user friendly** as they have to handle it all **and it burdens more if this is the only exception thrown** as we need to have a try-catch or throws just for this.
8. **One approach for converting a checked exception into unchecked exception is to split the method int a state testing method** (like hasNext()) to see if an exception might be thrown **and a state dependent method** (like nextItem()). However this will not work, if there is a chance for the state to

change between a state testing method call and its dependent method, or if a separate state testing method could duplicate the work of the state dependent method

9. Even though java spec does not require, there is a strong convention that **errors are reserved for use by the JVM to indicate failures that make it impossible to continue**.

10. Even though a Throwable is identical to ordinary checked exceptions, **we should not use Throwable instead of checked exceptions** as they will simply confuse users.

11. **You should provide methods in your exception classes to get the details of the failure programmatically** and should not leave them with no option than to parse the string representation of the exception.

12. **Whenever possible use standard exceptions provided by Java or subclass them** and add more functionality. This will make the code easier to learn and use. Commonly reused exceptions are IllegalArgumentException, NullPointerException, IllegalStateException, IndexOutOfBoundsException, ConcurrentModificationException and UnsupportedOperationException.

13. **Higher layers should catch lower layer exceptions and in their place, throw exceptions that can be explained in terms of the higher level abstraction**. This is called exception translation. **For better debugging we can do exception chaining** by passing the lower level exception (the cause) to higher level exception (using a suitable constructor or Throwable.initCause) and then access it later using an accessor (using Throwable.getCause).

14. **Exception translation should not be overused**; whenever possible we should try to handle exception in lower layer and try to make the lower layer methods succeed.

15. **Always declare checked exceptions individually** and not a common parent class for all the exceptions;**don't ever declare 'throws Exception'.** Do not declare unchecked exceptions using throws clause.

16. **Document all checked and unchecked exceptions using the javadoc @throws tag** and the conditions under which it is thrown. Even though documenting all of the unchecked exceptions is ideal, it is not always achievable in real world.

17. **Detail message of an exception should capture as much information as possible**, which will help in analysis, especially for non-reproducible failure. The detail message should also contain the values of all parameters and fields that contributed to the exception.

18. **Whenever an exception is thrown from a method, we need to make sure that the current object is in the state prior to execution of that method or at least we should document what state the object will be after that exception is thrown.** There are various ways to implement this failure atomicity like making the class immutable, checking parameters for validity and throwing exception before performing the operation, write recovery code to roll back or performing the operation on a temporary copy of the object. **If failure atomicity is not practical due to cost and complexity, don't do it, but document it.** Errors are unrecoverable and hence we should not even attempt for failure atomicity for them.

Source : http://javajee.com/best-practices-for-using-and-writing-exceptions-in-java