Chapter 10. Exceptions

ITEM 69: USE EXCEPTIONS ONLY FOR EXCEPTIONAL CONDITIONS

- **Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow**
- **A well-designed API must not force its clients to use exceptions for ordinary control flow**
- A class with a "state-dependent" method that can be invoked only under certain unpredictable conditions should generally have a separate
    - "state-testing" method indicating whether it is appropriate to invoke the state-dependent method (or)
    - to have the state-dependent method return an empty optional or a distinguished value such as `null` if it cannot perform the desired computation

ITEM 70: USE CHECKED EXCEPTIONS FOR RECOVERABLE CONDITIONS AND RUNTIME EXCEPTIONS FOR PROGRAMMING ERRORS

- **use checked exceptions for conditions from which the caller can reasonably be expected to recover**
- **Use runtime exceptions to indicate programming errors** - *precondition violations*
- **all of the unchecked throwables you implement should subclass** `RuntimeException`
- extremely bad practice - parse the string representation of an exception to ferret out additional information
- checked exceptions generally indicate recoverable conditions - important to provide methods that furnish information to help the caller recover from the exceptional condition

ITEM 71: AVOID UNNECESSARY USE OF CHECKED EXCEPTIONS

- burden on the user of the API - if a method throws checked exceptions, the code that invokes it must handle them in one or more `catch` blocks, or declare that it throws them and let them propagate
- methods throwing checked exceptions **can't be used directly in streams**
- additional burden - if it is the *sole* checked exception thrown by a method, it pays to ask yourself if there is a way to avoid the checked exception
- ways to eliminate a checked exception:
    - return an *optional* of the desired result type - disadvantage of this technique is that the method can't return any additional information detailing its inability to perform the desired computation
    - **turn a checked exception into an unchecked exception** by breaking the method that throws the exception into two methods, the first of which returns a `boolean` indicating whether the exception would be thrown
- If callers won't be able to recover from failures, throw unchecked exceptions

ITEM 72: FAVOR THE USE OF STANDARD EXCEPTIONS

Use only if the conditions under which you would throw it are consistent with the exception's documentation: **reuse must be based on documented semantics, not just on name**

Most commonly reused exceptions:

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocation
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been detected where it is prohibited
`UnsupportedOperationException`	Object does not support method

- **Do not Exception RuntimeException, Throwable, or Error directly**
- `ArithmeticException` and `NumberFormatException` if you were implementing arithmetic objects such as complex numbers or rational numbers
- feel free to subclass a standard exception if you want to add more detail, but remember that exceptions are serializable

ITEM 73: THROW EXCEPTIONS APPROPRIATE TO THE ABSTRACTION

- ***exception translation*** - higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction
- ***exception chaining*** - used when the lower-level exception might be helpful to someone debugging the problem that caused the higher-level exception
- **While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused**
- ways to deal with exceptions from lower layers:
    - avoid them, by ensuring that lower-level methods succeed (by checking the validity of the higher-level method's parameters before passing them on to lower layers)
    - have the higher layer silently work around these exceptions, insulating the caller of the higher-level method from lower-level problems (log the exception using some appropriate logging facility such as `java.util.logging`)

ITEM 74: DOCUMENT ALL EXCEPTIONS THROWN BY EACH METHOD

- **Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown** using the Javadoc `@throws` tag
- don't declare that a public method `throws Exception / Throwable` (except main method)

- A well-documented list of the unchecked exceptions that a method can throw effectively describes the *preconditions* for its successful execution (**especially for the methods in interfaces**)
- **Use the Javadoc `@throws` tag to document each exception that a method can throw, but do *not* use the `throws` keyword on unchecked exceptions**
- **If an exception is thrown by many methods in a class for the same reason, you can document the exception in the class's documentation comment** rather than documenting it individually for each method

ITEM 75: INCLUDE FAILURE-CAPTURE INFORMATION IN DETAIL MESSAGES

- critically important that the exception's `toString` method return as much information as possible concerning the cause of the failure
- **To capture a failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception**
- **do not include passwords, encryption keys, and the like in detail messages**
- Unlike a user-level error message, the detail message is primarily for the benefit of programmers or site reliability engineers, when analyzing a failure (information content is far more important than readability)
- One way to ensure that exceptions contain adequate failure-capture information in their detail messages is to require this information in their constructors instead of a string detail message

ITEM 76: STRIVE FOR FAILURE ATOMICITY

- *failure-atomic* method - **a failed method invocation should leave the object in the state that it was in prior to the invocation**
- Ways to achieve failure atomicity:
  - design immutable objects (failure atomicity is free)
  - mutable objects - check parameters for validity before performing the operation
  - order the computation so that any part that may fail takes place before any part that modifies the object
  - perform the operation on a ***temporary copy of the object*** and to replace the contents of the object with the temporary copy once the operation is complete
  - write *recovery code* - intercepts a failure that occurs in the midst of an operation, and causes the object to roll back its state to the point before the operation began
- in case of violation - API documentation should clearly indicate what state the object will be left in

ITEM 77: DON'T IGNORE EXCEPTIONS

- **An empty `catch` block defeats the purpose of exceptions**
- **If you choose to ignore an exception, the `catch` block should contain a comment explaining why it is appropriate to do so, and the variable should be named `ignored`**