Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

Java provides three kinds of throwables: *checked exceptions*, *runtime exceptions*, and *errors*. There is some confusion among programmers as to when it is appropriate to use each kind of throwable. While the decision is not always clear-cut, there are some general rules that provide strong guidance.

The cardinal rule in deciding whether to use a checked or an unchecked exception is this: **use checked exceptions for conditions from which the caller can reasonably be expected to recover.** By throwing a checked exception, you force the caller to handle the exception in a `catch` clause or to propagate it outward. Each checked exception that a method is declared to throw is therefore a potent indication to the API user that the associated condition is a possible outcome of invoking the method.

By confronting the user with a checked exception, the API designer presents a mandate to recover from the condition. The user can disregard the mandate by catching the exception and ignoring it, but this is usually a bad idea (Item 77).

There are two kinds of unchecked throwables: runtime exceptions and errors. They are identical in their behavior: both are throwables that needn't, and generally shouldn't, be caught. If a program throws an unchecked exception or an error, it is generally the case that recovery is impossible and continued execution would do more harm than good. If a program does not catch such a throwable, it will cause the current thread to halt with an appropriate error message.

Use runtime exceptions to indicate programming errors. The great majority of runtime exceptions indicate *precondition violations*. A precondition violation is simply a failure by the client of an API to adhere to the contract established by the API specification. For example, the contract for array access specifies that the array index must be between zero and the array length minus one, inclusive. `ArrayIndexOutOfBoundsException` indicates that this precondition was violated.

One problem with this advice is that it is not always clear whether you're dealing with a recoverable conditions or a programming error. For example, consider the case of resource exhaustion, which can be caused by a programming error such as allocating an unreasonably large array, or by a genuine shortage of resources. If resource exhaustion is caused by a temporary shortage or by temporarily heightened demand, the condition may well be recoverable. It is a matter of judgment on the part of the API designer whether a given instance of resource exhaustion is likely to allow for recovery. If you believe a condition is likely to allow for recovery, use a checked exception; if not, use a runtime

exception. If it isn't clear whether recovery is possible, you're probably better off using an unchecked exception, for reasons discussed in Item 71.

While the Java Language Specification does not require it, there is a strong convention that *errors* are reserved for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions that make it impossible to continue execution. Given the almost universal acceptance of this convention, it's best not to implement any new `Error` subclasses. Therefore, **all of the unchecked throwables you implement should subclass `RuntimeException`** (directly or indirectly). Not only shouldn't you define `Error` subclasses, but with the exception of `AssertionError`, you shouldn't throw them either.

It is possible to define a throwable that is not a subclass of `Exception`, `RuntimeException`, or `Error`. The JLS doesn't address such throwables directly but specifies implicitly that they behave as ordinary checked exceptions (which are subclasses of `Exception` but not `RuntimeException`). So when should you use such a beast? In a word, never. They have no benefits over ordinary checked exceptions and would serve merely to confuse the user of your API.

API designers often forget that exceptions are full-fledged objects on which arbitrary methods can be defined. The primary use of such methods is to provide code that catches the exception with additional information concerning the condition that caused the exception to be thrown. In the absence of such methods, programmers have been known to parse the string representation of an exception to ferret out additional information. This is extremely bad practice (Item 12). Throwable classes seldom specify the details of their string representations, so string representations can differ from implementation to implementation and release to release. Therefore, code that parses the string representation of an exception is likely to be nonportable and fragile.

Because checked exceptions generally indicate recoverable conditions, it's especially important for them to provide methods that furnish information to help the caller recover from the exceptional condition. For example, suppose a checked exception is thrown when an attempt to make a purchase with a gift card fails due to insufficient funds. The exception should provide an accessor method to query the amount of the shortfall. This will enable the caller to relay the amount to the shopper. See Item 75 for more on this topic.

To summarize, throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions. Don't define any throwables that are neither checked exceptions nor runtime exceptions. Provide methods on your checked exceptions to aid in recovery.