CHAPTER 10

Exceptions

WHEN used to best advantage, exceptions can improve a program's readability, reliability, and maintainability. When used improperly, they can have the opposite effect. This chapter provides guidelines for using exceptions effectively.

Item 69: Use exceptions only for exceptional conditions

Someday, if you are unlucky, you may stumble across a piece of code that looks something like this:

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

What does this code do? It's not at all obvious from inspection, and that's reason enough not to use it (Item 67). It turns out to be a horribly ill-conceived idiom for looping through the elements of an array. The infinite loop terminates by throwing, catching, and ignoring an `ArrayIndexOutOfBoundsException` when it attempts to access the first array element outside the bounds of the array. It's supposed to be equivalent to the standard idiom for looping through an array, which is instantly recognizable to any Java programmer:

```
for (Mountain m : range)
    m.climb();
```

So why would anyone use the exception-based loop in preference to the tried and true? It's a misguided attempt to improve performance based on the faulty

reasoning that, since the VM checks the bounds of all array accesses, the normal loop termination test—hidden by the compiler but still present in the for-each loop—is redundant and should be avoided. There are three things wrong with this reasoning:

- Because exceptions are designed for exceptional circumstances, there is little incentive for JVM implementors to make them as fast as explicit tests.
- Placing code inside a `try-catch` block inhibits certain optimizations that JVM implementations might otherwise perform.
- The standard idiom for looping through an array doesn't necessarily result in redundant checks. Many JVM implementations optimize them away.

In fact, the exception-based idiom is far slower than the standard one. On my machine, the exception-based idiom is about twice as slow as the standard one for arrays of one hundred elements.

Not only does the exception-based loop obfuscate the purpose of the code and reduce its performance, but it's not guaranteed to work. If there is a bug in the loop, the use of exceptions for flow control can mask the bug, greatly complicating the debugging process. Suppose the computation in the body of the loop invokes a method that performs an out-of-bounds access to some unrelated array. If a reasonable loop idiom were used, the bug would generate an uncaught exception, resulting in immediate thread termination with a full stack trace. If the misguided exception-based loop were used, the bug-related exception would be caught and misinterpreted as a normal loop termination.

The moral of this story is simple: **Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow.** More generally, use standard, easily recognizable idioms in preference to overly clever techniques that purport to offer better performance. Even if the performance advantage is real, it may not remain in the face of steadily improving platform implementations. The subtle bugs and maintenance headaches that come from overly clever techniques, however, are sure to remain.

This principle also has implications for API design. **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. For example, the `Iterator` interface has the state-dependent method `next` and the

corresponding state-testing method `hasNext`. This enables the standard idiom for iterating over a collection with a traditional `for` loop (as well as the for-each loop, where the `hasNext` method is used internally):

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
    ...
}
```

If `Iterator` lacked the `hasNext` method, clients would be forced to do this instead:

```
// Do not use this hideous code for iteration over a collection!
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next();
        ...
    }
} catch (NoSuchElementException e) {
}
```

This should look very familiar after the array iteration example that began this item. In addition to being wordy and misleading, the exception-based loop is likely to perform poorly and can mask bugs in unrelated parts of the system.

An alternative to providing a separate state-testing method is to have the state-dependent method return an empty optional (Item 55) or a distinguished value such as `null` if it cannot perform the desired computation.

Here are some guidelines to help you choose between a state-testing method and an optional or distinguished return value. If an object is to be accessed concurrently without external synchronization or is subject to externally induced state transitions, you must use an optional or distinguished return value, as the object's state could change in the interval between the invocation of a state-testing method and its state-dependent method. Performance concerns may dictate that an optional or distinguished return value be used if a separate state-testing method would duplicate the work of the state-dependent method. All other things being equal, a state-testing method is mildly preferable to a distinguished return value. It offers slightly better readability, and incorrect use may be easier to detect: if you forget to call a state-testing method, the state-dependent method will throw an exception, making the bug obvious; if you forget to check for a distinguished return value, the bug may be subtle. This is not an issue for optional return values.

In summary, exceptions are designed for exceptional conditions. Don't use them for ordinary control flow, and don't write APIs that force others to do so.

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.

Item 71: Avoid unnecessary use of checked exceptions

Many Java programmers dislike checked exceptions, but used properly, they can improve APIs and programs. Unlike return codes and unchecked exceptions, they *force* programmers to deal with problems, enhancing reliability. That said, overuse of checked exceptions in APIs can make them far less pleasant to use. 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 outward. Either way, it places a burden on the user of the API. The burden increased in Java 8, as methods throwing checked exceptions can't be used directly in streams (Items 45–48).

This burden may be justified if the exceptional condition cannot be prevented by proper use of the API *and* the programmer using the API can take some useful action once confronted with the exception. Unless both of these conditions are met, an unchecked exception is appropriate. As a litmus test, ask yourself how the programmer will handle the exception. Is this the best that can be done?

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Can't happen!
}
```

Or this?

```
} catch (TheCheckedException e) {
    e.printStackTrace();        // Oh well, we lose.
    System.exit(1);
}
```

If the programmer can do no better, an unchecked exception is called for.

The additional burden on the programmer caused by a checked exception is substantially higher if it is the *sole* checked exception thrown by a method. If there are others, the method must already appear in a `try` block, and this exception requires, at most, another `catch` block. If a method throws a single checked exception, this exception is the sole reason the method must appear in a `try` block and can't be used directly in streams. Under these circumstances, it pays to ask yourself if there is a way to avoid the checked exception.

The easiest way to eliminate a checked exception is to return an *optional* of the desired result type (Item 55). Instead of throwing a checked exception, the method simply returns an empty optional. The disadvantage of this technique is that the method can't return any additional information detailing its inability to perform the desired computation. Exceptions, by contrast, have descriptive types, and can export methods to provide additional information (Item 70).

You can also 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. This API refactoring transforms the calling sequence from this:

```
// Invocation with checked exception
try {
    obj.action(args);
} catch (TheCheckedException e) {
    ... // Handle exceptional condition
}
```

into this:

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    ... // Handle exceptional condition
}
```

This refactoring is not always appropriate, but where it is, it can make an API more pleasant to use. While the latter calling sequence is no prettier than the former, the refactored API is more flexible. If the programmer knows the call will succeed, or is content to let the thread terminate if it fails, the refactoring also allows this trivial calling sequence:

```
obj.action(args);
```

If you suspect that the trivial calling sequence will be the norm, then the API refactoring may be appropriate. The resulting API is essentially the state-testing method API in Item 69 and the same caveats apply: if an object is to be accessed concurrently without external synchronization or it is subject to externally induced state transitions, this refactoring is inappropriate because the object's state may change between the calls to `actionPermitted` and `action`. If a separate `actionPermitted` method would duplicate the work of the `action` method, the refactoring may be ruled out on performance grounds.

In summary, when used sparingly, checked exceptions can increase the reliability of programs; when overused, they make APIs painful to use. If callers won't be able to recover from failures, throw unchecked exceptions. If recovery may be possible and you want to *force* callers to handle exceptional conditions, first consider returning an optional. Only if this would provide insufficient information in the case of failure should you throw a checked exception.

Item 72: Favor the use of standard exceptions

An attribute that distinguishes expert programmers from less experienced ones is that experts strive for and usually achieve a high degree of code reuse. Exceptions are no exception to the rule that code reuse is a good thing. The Java libraries provide a set of exceptions that covers most of the exception-throwing needs of most APIs.

Reusing standard exceptions has several benefits. Chief among them is that it makes your API easier to learn and use because it matches the established conventions that programmers are already familiar with. A close second is that programs using your API are easier to read because they aren't cluttered with unfamiliar exceptions. Last (and least), fewer exception classes means a smaller memory footprint and less time spent loading classes.

The most commonly reused exception type is `IllegalArgumentException` (Item 49). This is generally the exception to throw when the caller passes in an argument whose value is inappropriate. For example, this would be the exception to throw if the caller passed a negative number in a parameter representing the number of times some action was to be repeated.

Another commonly reused exception is `IllegalStateException`. This is generally the exception to throw if the invocation is illegal because of the state of the receiving object. For example, this would be the exception to throw if the caller attempted to use some object before it had been properly initialized.

Arguably, every erroneous method invocation boils down to an illegal argument or state, but other exceptions are standardly used for certain kinds of illegal arguments and states. If a caller passes `null` in some parameter for which null values are prohibited, convention dictates that `NullPointerException` be thrown rather than `IllegalArgumentException`. Similarly, if a caller passes an out-of-range value in a parameter representing an index into a sequence, `IndexOutOfBoundsException` should be thrown rather than `IllegalArgumentException`.

Another reusable exception is `ConcurrentModificationException`. It should be thrown if an object that was designed for use by a single thread (or with external synchronization) detects that it is being modified concurrently. This exception is at best a hint because it is impossible to reliably detect concurrent modification.

A last standard exception of note is `UnsupportedOperationException`. This is the exception to throw if an object does not support an attempted operation. Its use is rare because most objects support all of their methods. This exception is used by classes that fail to implement one or more *optional operations* defined by an interface they implement. For example, an append-only `List` implementation would throw this exception if someone tried to delete an element from the list.

Do *not* reuse `Exception`, `RuntimeException`, `Throwable`, or `Error` directly. Treat these classes as if they were abstract. You can't reliably test for these exceptions because they are superclasses of other exceptions that a method may throw.

This table summarizes the 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

While these are by far the most commonly reused exceptions, others may be reused where circumstances warrant. For example, it would be appropriate to reuse `ArithmeticException` and `NumberFormatException` if you were implementing arithmetic objects such as complex numbers or rational numbers. If an exception fits your needs, go ahead and use it, but 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. Also, feel free to subclass a standard exception if you want to add more detail (Item 75), but remember that exceptions are serializable (Chapter 12). That alone is reason not to write your own exception class without good reason.

Choosing which exception to reuse can be tricky because the "occasions for use" in the table above do not appear to be mutually exclusive. Consider the case of an object representing a deck of cards, and suppose there were a method to deal a hand from the deck that took as an argument the size of the hand. If the caller passed a value larger than the number of cards remaining in the deck, it could be construed as an `IllegalArgumentException` (the `handSize` parameter value is too high) or an `IllegalStateException` (the deck contains too few cards). Under these circumstances, the rule is to **throw `IllegalStateException` if no argument values would have worked, otherwise throw `IllegalArgumentException`.**

Item 73: Throw exceptions appropriate to the abstraction

It is disconcerting when a method throws an exception that has no apparent connection to the task that it performs. This often happens when a method propagates an exception thrown by a lower-level abstraction. Not only is it disconcerting, but it pollutes the API of the higher layer with implementation details. If the implementation of the higher layer changes in a later release, the exceptions it throws will change too, potentially breaking existing client programs.

To avoid this problem, **higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction.** This idiom is known as *exception translation*:

```
// Exception Translation
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Here is an example of exception translation taken from the `AbstractSequentialList` class, which is a *skeletal implementation* (Item 20) of the `List` interface. In this example, exception translation is mandated by the specification of the `get` method in the `List<E>` interface:

```
/**
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()}).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

A special form of exception translation called *exception chaining* is called for in cases where the lower-level exception might be helpful to someone debugging the problem that caused the higher-level exception. The lower-level exception (the

cause) is passed to the higher-level exception, which provides an accessor method (`Throwable`'s `getCause` method) to retrieve the lower-level exception:

```
// Exception Chaining
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

The higher-level exception's constructor passes the cause to a *chaining-aware* superclass constructor, so it is ultimately passed to one of `Throwable`'s chaining-aware constructors, such as `Throwable(Throwable)`:

```
// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
    }
}
```

Most standard exceptions have chaining-aware constructors. For exceptions that don't, you can set the cause using `Throwable`'s `initCause` method. Not only does exception chaining let you access the cause programmatically (with `getCause`), but it integrates the cause's stack trace into that of the higher-level exception.

While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused. Where possible, the best way to deal with exceptions from lower layers is to avoid them, by ensuring that lower-level methods succeed. Sometimes you can do this by checking the validity of the higher-level method's parameters before passing them on to lower layers.

If it is impossible to prevent exceptions from lower layers, the next best thing is to have the higher layer silently work around these exceptions, insulating the caller of the higher-level method from lower-level problems. Under these circumstances, it may be appropriate to log the exception using some appropriate logging facility such as `java.util.logging`. This allows programmers to investigate the problem, while insulating client code and the users from it.

In summary, if it isn't feasible to prevent or to handle exceptions from lower layers, use exception translation, unless the lower-level method happens to guarantee that all of its exceptions are appropriate to the higher level. Chaining provides the best of both worlds: it allows you to throw an appropriate higher-level exception, while capturing the underlying cause for failure analysis (Item 75).

Item 74: Document all exceptions thrown by each method

A description of the exceptions thrown by a method is an important part of the documentation required to use the method properly. Therefore, it is critically important that you take the time to carefully document all of the exceptions thrown by each method (Item 56).

Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown using the Javadoc `@throws` tag. Don't take the shortcut of declaring that a method throws some superclass of multiple exception classes that it can throw. As an extreme example, don't declare that a public method `throws Exception` or, worse, `throws Throwable`. In addition to denying any guidance to the method's user concerning the exceptions it is capable of throwing, such a declaration greatly hinders the use of the method because it effectively obscures any other exception that may be thrown in the same context. One exception to this advice is the `main` method, which can safely be declared to throw `Exception` because it is called only by VM.

While the language does not require programmers to declare the unchecked exceptions that a method is capable of throwing, it is wise to document them as carefully as the checked exceptions. Unchecked exceptions generally represent programming errors (Item 70), and familiarizing programmers with all of the errors they can make helps them avoid making these errors. A well-documented list of the unchecked exceptions that a method can throw effectively describes the *preconditions* for its successful execution. It is essential that every public method's documentation describe its preconditions (Item 56), and documenting its unchecked exceptions is the best way to satisfy this requirement.

It is particularly important that methods in interfaces document the unchecked exceptions they may throw. This documentation forms a part of the interface's *general contract* and enables common behavior among multiple implementations of the interface.

Use the Javadoc `@throws` tag to document each exception that a method can throw, but do *not* use the `throws` keyword on unchecked exceptions. It is important that programmers using your API are aware of which exceptions are checked and which are unchecked because the programmers' responsibilities differ in these two cases. The documentation generated by the Javadoc `@throws` tag without a corresponding throws clause in the method declaration provides a strong visual cue to the programmer that an exception is unchecked.

It should be noted that documenting all of the unchecked exceptions that each method can throw is an ideal, not always achievable in the real world. When a class undergoes revision, it is not a violation of source or binary compatibility if an exported method is modified to throw additional unchecked exceptions. Suppose a class invokes a method from another, independently written class. The authors of the former class may carefully document all of the unchecked exceptions that each method throws, but if the latter class is revised to throw additional unchecked exceptions, it is quite likely that the former class (which has not undergone revision) will propagate the new unchecked exceptions even though it does not document them.

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. A common example is `NullPointerException`. It is fine for a class's documentation comment to say, "All methods in this class throw a `NullPointerException` if a null object reference is passed in any parameter," or words to that effect.

In summary, document every exception that can be thrown by each method that you write. This is true for unchecked as well as checked exceptions, and for abstract as well as concrete methods. This documentation should take the form of `@throws` tags in doc comments. Declare each checked exception individually in a method's `throws` clause, but do not declare unchecked exceptions. If you fail to document the exceptions that your methods can throw, it will be difficult or impossible for others to make effective use of your classes and interfaces.

Item 75: Include failure-capture information in detail messages

When a program fails due to an uncaught exception, the system automatically prints out the exception's stack trace. The stack trace contains the exception's *string representation*, the result of invoking its `toString` method. This typically consists of the exception's class name followed by its *detail message*. Frequently this is the only information that programmers or site reliability engineers will have when investigating a software failure. If the failure is not easily reproducible, it may be difficult or impossible to get any more information. Therefore, it is critically important that the exception's `toString` method return as much information as possible concerning the cause of the failure. In other words, the detail message of an exception should *capture the failure* for subsequent analysis.

To capture a failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception. For example, the detail message of an `IndexOutOfBoundsException` should contain the lower bound, the upper bound, and the index value that failed to lie between the bounds. This information tells a lot about the failure. Any or all of the three values could be wrong. The index could be one less than the lower bound or equal to the upper bound (a "fencepost error"), or it could be a wild value, far too low or high. The lower bound could be greater than the upper bound (a serious internal invariant failure). Each of these situations points to a different problem, and it greatly aids in the diagnosis if you know what sort of error you're looking for.

One caveat concerns security-sensitive information. Because stack traces may be seen by many people in the process of diagnosing and fixing software issues, **do not include passwords, encryption keys, and the like in detail messages.**

While it is critical to include all of the pertinent data in the detail message of an exception, it is generally unimportant to include a lot of prose. The stack trace is intended to be analyzed in conjunction with the documentation and, if necessary, source code. It generally contains the exact file and line number from which the exception was thrown, as well as the files and line numbers of all other method invocations on the stack. Lengthy prose descriptions of the failure are superfluous; the information can be gleaned by reading the documentation and source code.

The detail message of an exception should not be confused with a user-level error message, which must be intelligible to end users. Unlike a user-level error message, the detail message is primarily for the benefit of programmers or site reliability engineers, when analyzing a failure. Therefore, information content is far more important than readability. User-level error messages are often *localized*, whereas exception detail messages rarely are.

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. The detail message can then be generated automatically to include the information. For example, instead of a `String` constructor, `IndexOutOfBoundsException` could have had a constructor that looks like this:

```
/**
 * Constructs an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value
 * @param upperBound the highest legal index value plus one
 * @param index      the actual index value
 */
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                 int index) {
    // Generate a detail message that captures the failure
    super(String.format(
            "Lower bound: %d, Upper bound: %d, Index: %d",
            lowerBound, upperBound, index));

    // Save failure information for programmatic access
    this.lowerBound = lowerBound;
    this.upperBound = upperBound;
    this.index = index;
}
```

As of Java 9, `IndexOutOfBoundsException` finally acquired a constructor that takes an `int` valued `index` parameter, but sadly it omits the `lowerBound` and `upperBound` parameters. More generally, the Java libraries don't make heavy use of this idiom, but it is highly recommended. It makes it easy for the programmer throwing an exception to capture the failure. In fact, it makes it hard for the programmer not to capture the failure! In effect, the idiom centralizes the code to generate a high-quality detail message in the exception class, rather than requiring each user of the class to generate the detail message redundantly.

As suggested in Item 70, it may be appropriate for an exception to provide accessor methods for its failure-capture information (`lowerBound`, `upperBound`, and `index` in the above example). It is more important to provide such accessor methods on checked exceptions than unchecked, because the failure-capture information could be useful in recovering from the failure. It is rare (although not inconceivable) that a programmer might want programmatic access to the details of an unchecked exception. Even for unchecked exceptions, however, it seems advisable to provide these accessors on general principle (Item 12, page 57).

Item 76: Strive for failure atomicity

After an object throws an exception, it is generally desirable that the object still be in a well-defined, usable state, even if the failure occurred in the midst of performing an operation. This is especially true for checked exceptions, from which the caller is expected to recover. **Generally speaking, a failed method invocation should leave the object in the state that it was in prior to the invocation**. A method with this property is said to be *failure-atomic.*

There are several ways to achieve this effect. The simplest is to design immutable objects (Item 17). If an object is immutable, failure atomicity is free. If an operation fails, it may prevent a new object from getting created, but it will never leave an existing object in an inconsistent state, because the state of each object is consistent when it is created and can't be modified thereafter.

For methods that operate on mutable objects, the most common way to achieve failure atomicity is to check parameters for validity before performing the operation (Item 49). This causes most exceptions to get thrown before object modification commences. For example, consider the `Stack.pop` method in Item 7:

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

If the initial size check were eliminated, the method would still throw an exception when it attempted to pop an element from an empty stack. It would, however, leave the size field in an inconsistent (negative) state, causing any future method invocations on the object to fail. Additionally, the `ArrayIndexOutOfBoundsException` thrown by the `pop` method would be inappropriate to the abstraction (Item 73).

A closely related approach to achieving failure atomicity is to order the computation so that any part that may fail takes place before any part that modifies the object. This approach is a natural extension of the previous one when arguments cannot be checked without performing a part of the computation. For example, consider the case of `TreeMap`, whose elements are sorted according to some ordering. In order to add an element to a `TreeMap`, the element must be of a type that can be compared using the `TreeMap`'s ordering. Attempting to add an incorrectly

typed element will naturally fail with a `ClassCastException` as a result of searching for the element in the tree, before the tree has been modified in any way.

A third approach to achieving failure atomicity is to 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. This approach occurs naturally when the computation can be performed more quickly once the data has been stored in a temporary data structure. For example, some sorting functions copy their input list into an array prior to sorting to reduce the cost of accessing elements in the inner loop of the sort. This is done for performance, but as an added benefit, it ensures that the input list will be untouched if the sort fails.

A last and far less common approach to achieving failure atomicity is to write *recovery code* that 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. This approach is used mainly for durable (disk-based) data structures.

While failure atomicity is generally desirable, it is not always achievable. For example, if two threads attempt to modify the same object concurrently without proper synchronization, the object may be left in an inconsistent state. It would therefore be wrong to assume that an object was still usable after catching a `ConcurrentModificationException`. Errors are unrecoverable, so you need not even attempt to preserve failure atomicity when throwing `AssertionError`.

Even where failure atomicity is possible, it is not always desirable. For some operations, it would significantly increase the cost or complexity. That said, it is often both free and easy to achieve failure atomicity once you're aware of the issue.

In summary, as a rule, any generated exception that is part of a method's specification should leave the object in the same state it was in prior to the method invocation. Where this rule is violated, the API documentation should clearly indicate what state the object will be left in. Unfortunately, plenty of existing API documentation fails to live up to this ideal.

Item 77: Don't ignore exceptions

While this advice may seem obvious, it is violated often enough that it bears repeating. When the designers of an API declare a method to throw an exception, they are trying to tell you something. Don't ignore it! It is easy to ignore exceptions by surrounding a method invocation with a `try` statement whose `catch` block is empty:

```
// Empty catch block ignores exception - Highly suspect!
try {
    ...
} catch (SomeException e) {
}
```

An empty `catch` block defeats the purpose of exceptions, which is to force you to handle exceptional conditions. Ignoring an exception is analogous to ignoring a fire alarm—and turning it off so no one else gets a chance to see if there's a real fire. You may get away with it, or the results may be disastrous. Whenever you see an empty `catch` block, alarm bells should go off in your head.

There are situations where it is appropriate to ignore an exception. For example, it might be appropriate when closing a `FileInputStream`. You haven't changed the state of the file, so there's no need to perform any recovery action, and you've already read the information that you need from the file, so there's no reason to abort the operation in progress. It may be wise to log the exception, so that you can investigate the matter if these exceptions happen often. **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`:**

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // Default; guaranteed sufficient for any map
try {
    numColors = f.get(1L, TimeUnit.SECONDS);
} catch (TimeoutException | ExecutionException ignored) {
    // Use default: minimal coloring is desirable, not required
}
```

The advice in this item applies equally to checked and unchecked exceptions. Whether an exception represents a predictable exceptional condition or a programming error, ignoring it with an empty `catch` block will result in a program that continues silently in the face of error. The program might then fail at an arbitrary time in the future, at a point in the code that bears no apparent relation to the source of the problem. Properly handling an exception can avert failure entirely. Merely letting an exception propagate outward can at least cause the program to fail swiftly, preserving information to aid in debugging the failure.