- Reading 6, Part 2: Exceptions
- Exceptions for signaling bugs
- Exceptions for special results
- Checked and unchecked exceptions
- Throwable hierarchy
- Exception design considerations
- Abuse of exceptions
- Next: Summary

Reading 6, Part 2: Exceptions

Now that we're writing specifications and thinking about how clients will use our methods, let's discuss how to handle *exceptional* cases in a way that is safe from bugs and easy to understand.

A method's *signature* — its name, parameter types, return type — is a core part of its specification, and the signature may also include *exceptions* that the method may trigger.

Exceptions for signaling bugs

You've probably already seen some exceptions in your Java programming so far, such as `ArrayIndexOutOfBoundsException` (thrown when an array index `foo[i]` is outside the valid range for the array `foo`) or `NullPointerException` (thrown when trying to call a method on a `null` object reference). These exceptions generally indicate **bugs** in your code, and the information displayed by Java when the exception is thrown can help you find and fix the bug.

`ArrayIndexOutOfBounds-` and `NullPointerException` are probably the most common exceptions of this sort. Other examples include:

- `ArithmeticException` , thrown for arithmetic errors like integer division by zero.
- `NumberFormatException` , thrown by methods like `Integer.parseInt` if you pass in a string that cannot be parsed into an integer.

Exceptions for special results

Exceptions are not just for signaling bugs. They can be used to improve the structure of code that involves procedures with special results.

An unfortunately common way to handle special results is to return special values. Lookup operations in the Java library are often designed like this: you get an index of -1 when expecting a positive integer, or a `null` reference when expecting an object. This approach is OK if used sparingly, but it has two problems. First, it's tedious to check the return value. Second, it's easy to forget to do it. (We'll see that by using exceptions you can get help from the compiler in this.)

Also, it's not always easy to find a 'special value'. Suppose we have a `BirthdayBook` class with a lookup method. Here's one possible method signature:

```
class BirthdayBook {
    LocalDate lookup(String name) { ... }
}
```

(`LocalDate` is part of the Java API.)

What should the method do if the birthday book doesn't have an entry for the person whose name is given? Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was *obvious* that no program written in 1960 would still be running at the end of the century. (They were wrong, by the way.)

Here's a better approach. The method throws an exception:

```
LocalDate lookup(String name) throws NotFoundException {
    ...
    if ( ...not found... )
        throw new NotFoundException();
    ...
```

and the caller handles the exception with a `catch` clause. For example:

```
BirthdayBook birthdays = ...
try {
    LocalDate birthdate = birthdays.lookup("Alyssa");
    // we know Alyssa's birthday
} catch (NotFoundException nfe) {
    // her birthday was not in the birthday book
```

}

Now there's no need for any special value, nor the checking associated with it.

Read: **Exceptions** in the Java Tutorials.

Checked and unchecked exceptions

We've seen two different purposes for exceptions: special results and bug detection. As a general rule, you'll want to use checked exceptions to signal special results and unchecked exceptions to signal bugs. In a later section, we'll refine this a bit.

Some terminology: *checked* exceptions are called that because they are checked by the compiler:

- If a method might throw a checked exception, the possibility must be declared in its signature. `NotFoundException` would be a checked exception, and that's why the signature ends `throws NotFoundException`.
- If a method calls another method that may throw a checked exception, it must either handle it, or declare the exception itself, since if it isn't caught locally it will be propagated up to callers.

So if you call `BirthdayBook`'s `lookup` method and forget to handle the `NotFoundException`, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur will be handled.

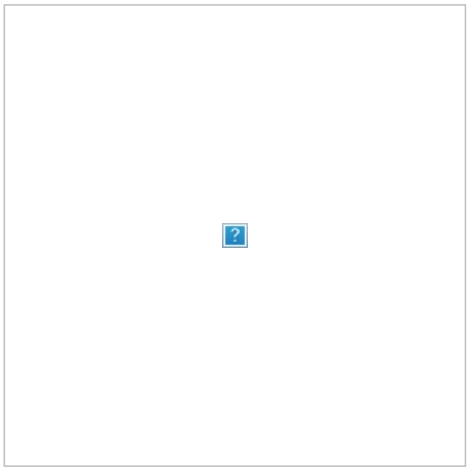
Unchecked exceptions, in contrast, are used to signal bugs. These exceptions are not expected to be handled by the code except perhaps at the top level. We wouldn't want every method up the call chain to have to declare that it (might) throw all the kinds of bug-related exceptions that can happen at lower call levels: index out of bounds, null pointers, illegal arguments, assertion failures, etc.

As a result, for an unchecked exception the compiler will not check for `try-catch` or a `throws` declaration. Java still allows you to write a `throws` clause for an unchecked exception as part of a method signature, but this has no effect, and is thus a bit funny, and we don't recommend doing it.

All exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is `null`.

Throwable hierarchy

To understand how Java decides whether an exception is checked or unchecked, let's look at the class hierarchy for Java exceptions.



`Throwable` is the class of objects that can be thrown or caught. `Throwable`'s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception. Any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`.

`Error` is a subclass of `Throwable` that is reserved for errors produced by the Java runtime system, such as `StackOverflowError` and `OutOfMemoryError`. For some reason `AssertionError` also extends `Error`, even though it indicates a bug in user code, not in the runtime. Errors should be considered unrecoverable, and are generally not caught.

Here's how Java distinguishes between checked and unchecked exceptions:

- `RuntimeException`, `Error`, and their subclasses are **unchecked** exceptions. The compiler doesn't require them to be declared in the `throws` clause of a method that throws them, and doesn't require them to be caught or declared by a caller of such a method.
- All other throwables — `Throwable`, `Exception`, and all of their subclasses except for those of the `RuntimeException` and `Error` lineage — are **checked** exceptions. The compiler requires these exceptions to be caught or declared when it's possible for them to be thrown.

When you define your own exceptions, you should either subclass `RuntimeException` (to make it an unchecked exception) or `Exception` (to make it checked). Programmers generally don't subclass `Error` or `Throwable`, because these are reserved

by Java itself.

Exception design considerations

The rule we have given — use checked exceptions for special results (i.e., anticipated situations), and unchecked exceptions to signal bugs (unexpected failures) — makes sense, but it isn't the end of the story. The snag is that exceptions in Java aren't as lightweight as they might be.

Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use, in both method design and method use. If you *design* a method to have its own (new) exception, you have to create a new class for the exception. If you *call* a method that can throw a checked exception, you have to wrap it in a `try - catch` statement (even if you know the exception will never be thrown). This latter stipulation creates a dilemma. Suppose, for example, you're designing a queue abstraction. Should popping the queue throw a checked exception when the queue is empty? Suppose you want to support a style of programming in the client in which the queue is popped (in a loop say) until the exception is thrown. So you choose a checked exception. Now some client wants to use the method in a context in which, immediately prior to popping, the client tests whether the queue is empty and only pops if it isn't. Maddeningly, that client will still need to wrap the call in a `try - catch` statement.

This suggests a more refined rule:

- You should use an unchecked exception only to signal an unexpected failure (i.e. a bug), or if you expect that clients will usually write code that ensures the exception will not happen, because there is a convenient and inexpensive way to avoid the exception;
- Otherwise you should use a checked exception.

Here are some examples of applying this rule to hypothetical methods:

- `Queue.pop()` throws an *unchecked* `EmptyQueueException` when the queue is empty, because it's reasonable to expect the caller to avoid this with a call like `Queue.size()` or `Queue.isEmpty()` .
- `Url.getWebPage()` throws a *checked* `IOException` when it can't retrieve the web page, because it's not easy for the caller to prevent this.
- `int integerSquareRoot(int x)` throws a *checked* `NotPerfectSquareException` when `x` has no integral square root, because testing whether `x` is a perfect square is just as hard as finding the actual square root, so it's not reasonable to expect the caller to prevent it.

The cost of using exceptions in Java is one reason that many Java API's use the null reference as a special value. It's not a terrible thing to do, so long as it's done judiciously, and carefully specified.

Abuse of exceptions

Here's an example from *Effective Java* by Joshua Bloch (Item 57 in the 2nd edition).

```
try {
    int i = 0;
    while (true)
        a[i++].f();
} catch (ArrayIndexOutOfBoundsException e) { }
```

> What does this code do? It is not at all obvious from inspection, and that's reason enough not to use it. … 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 is supposed to be equivalent to:

```
for (int i = 0; i < a.length; i++) {
    a[i].f();
}
```

Or (using appropriate type `T`) to:

```
for (T x : a) {
    x.f();
}
```

The exception-based idiom, Bloch writes:

> … is a misguided attempt to improve performance based on the faulty reasoning that, since the VM checks the bounds of array accesses, the normal loop termination test (`i < a.length`) is redundant and should be avoided.

However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance. On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99.

Much worse than that, the exception-based idiom is not even guaranteed to work! Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array. What happens?

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.

Next: Summary