Checked vs. Unchecked Exceptions: The Debate Is Not Over

Do we need checked exceptions at all? The [debate](http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation) is over, isn't it? Not for me. While most object-oriented languages don't have them, and most programmers think checked exceptions are a Java mistake, I believe in the opposite — unchecked exceptions are the **mistake**. Moreover, I believe multiple exception types are a bad idea too.

Let me first explain how I understand exceptions in object-oriented programming. Then I'll compare my understanding with a "traditional" approach, and we'll discuss the differences. So, my understanding first.

Say there is a method that saves some binary data to a file:

**public** **void** save(File file, **byte**[] data)

**throws** Exception {

// save data to the file

}

When everything goes right, the method just saves the data and returns control. When something is wrong, it throws Exception and we have to do something about it:

try {

save(file, data);

} catch (Exception ex) {

System.out.println("Sorry, we can't save right now.");

}

When a method says it throws an exception, I understand that the method is not **safe**. It may fail sometimes, and it's my responsibility to either 1) handle this failure or 2) declare myself as **unsafe** too.

I know each method is designed with a [single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) in mind. This is a guarantee to me that if method save() fails, it means the entire saving operation can't be completed. If I need to know what the cause of this failure was, I will [un-chain](https://en.wikipedia.org/wiki/Exception_chaining) the exception — traverse the stack of chained exceptions and stack traces encapsulated in ex.

I never use exceptions for flow control, which means I **never recover** situations where exceptions are thrown. When an exception occurs, I let it float up to the highest level of the application. Sometimes I rethrow it in order to add more semantic information to the chain. That's why it doesn't matter to me what the cause of the exception thrown by save() was. I just know the method failed. That's enough for me. Always.

For the same reason, I don't need to differentiate between different exception types. I just don't need that type of hierarchy. Exception is enough for me. Again, that's because I don't use exceptions for flow control.

That's how I understand exceptions.

According to this paradigm, I would say we must:

* always use checked exceptions.
* never throw/use unchecked exceptions.
* use only Exception, without any sub-types.
* always declare one exception type in the throws block.
* never catch without rethrowing; read [more about that here](http://www.yegor256.com/2015/07/09/catch-if-cant-otherwise.html).

This paradigm diverges from many other articles I've found on this subject. Let's compare and discuss.

## **Runtime vs. API Exceptions**

[Oracle says](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) some exceptions should be part of API (checked ones) while some are runtime exceptions and should not be part of it (unchecked). They will be documented in JavaDoc but not in the method signature.

I don't understand the logic here, and I'm sure Java designers don't understand it either. How and why are some exceptions important while others are not? Why do some of them deserve a proper API position in thethrows block of the method signature while others don't? What is the criteria?

I have an answer here, though. By introducing checked and unchecked exceptions, Java developers tried to solve the problem of methods that are too complex and messy. When a method is too big and does too many things at the same time (violates the single responsibility principle), it's definitely better to let us keep some exceptions "hidden" (a.k.a. unchecked). But it's not a real solution. It is only a temporary patch that does all of us more harm than good — methods keep growing in size and complexity.

Unchecked exceptions are a mistake in Java design, not checked ones.

Hiding the fact that a method may fail at some point is a mistake. That's exactly what unchecked exceptions do.

Instead, we should make this fact visible. When a method does too many things, there will be too many points of failure, and the author of the method will realize that something is wrong — a method should not throw exceptions in so many situations. This will lead to refactoring. The existence of unchecked exceptions leads to a mess. By the way, checked exceptions don't exist at all in Ruby, C#, Python, PHP, etc. This means that creators of these languages understand OOP even less than Java authors.

## **Checked Exceptions Are Too Noisy**

Another common argument against checked exceptions is that they make our code more verbose. We have to put try/catch everywhere instead of staying focused on the main logic. Bozhidar Bozhanov even [suggests](http://techblog.bozho.net/checked-and-unchecked-exceptions-in-java/) a technical solution for this verbosity problem.

Again, I don't understand this logic. If I want to do something when methodsave() fails, I catch the exception and handle the situation somehow. If I don't want to do that, I just say my method also throws and pay no attention to exception handling. What is the problem? Where is the verbosity coming from?

I have an answer here, too. It's coming from the existence of unchecked exceptions. We simply can't always ignore failure, because the interfaces we're using don't allow us to do this. That's all. For example, class[Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html), which is widely used for multi-thread programming, has method run() that is not supposed to throw anything. That's why we always have to catch everything inside the method and rethrow checked exceptions as unchecked.

If all methods in all Java interfaces would be declared either as "safe" (throws nothing) or "unsafe" (throws Exception), everything would become logical and clear. If you want to stay "safe," take responsibility for failure handling. Otherwise, be "unsafe" and let your users worry about safety.

No noise, very clean code, and obvious logic.

## **Inappropriately Exposed Implementation Details**

[Some say](http://www.ibm.com/developerworks/library/j-jtp05254/) the ability to put a checked exception into throws in the method signature instead of catching it here and rethrowing a new type encourages us to have too many irrelevant exception types in method signatures. For example, our method save() may declare that it may throw OutOfMemoryException, even though it seems to have nothing to do with memory allocation. But it does allocate some memory, right? So such a memory overflow may happen during a file saving operation.

Yet again, I don't get the logic of this argument. If all exceptions are checked, and we don't have multiple exception types, we just throw Exception everywhere, and that's it. Why do we need to care about the exception type in the first place? If we don't use exceptions to control flow, we won't do this.

If we really want to make our application memory overflow-resistant, we will introduce some memory manager, which will have something like thebigEnough() method, which will tell us whether our heap is big enough for the next operation. Using exceptions in such situations is a totally inappropriate approach to exception management in OOP.

## **Recoverable Exceptions**

Joshua Bloch, in [Effective Java](http://www.amazon.com/gp/product/0321356683/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0321356683&linkCode=as2&tag=yegor256com-20&linkId=QPTA6QN63DE364IM), says to "use checked exceptions for recoverable conditions and runtime exceptions for programming errors." He means something like this:

try {

save(file, data);

} catch (Exception ex) {

// We can't save the file, but it's OK

// Let's move on and do something else

}

How is that any different from a famous anti-pattern called [Don't Use Exceptions for Flow Control](http://c2.com/cgi/wiki?DontUseExceptionsForFlowControl)? Joshua, with all due respect, you're wrong. There are no such things as recoverable conditions in OOP. An exception indicates that the execution of a chain of calls from method to method is broken, and it's time to go up through the chain and stop somewhere. But we never go back again after the exception:

App#run()

Data#update()

Data#write()

File#save() <-- Boom, there's a failure here, so we go up

We can start this chain again, but we don't go back after throw. In other words, we don't **do** anything in the catch block. We only report the problem and wrap up execution. We never "recover"!

All arguments against checked exceptions demonstrate nothing but a serious misunderstanding of object-oriented programming by their authors. The mistake in Java and in many other languages is the existence of unchecked exceptions, not checked ones.