**Exceptions-2022**



**Checked VS UnChecked Exceptions**

Checked Exceptions should be used for **predictable, but unpreventable** errors that are reasonable to recover from.

**Unchecked Exceptions** should be used for everything else.

1. **Predictable but unpreventable**: The caller did everything within their power to validate the input parameters, but some condition outside their control has caused the operation to fail. For example, you try reading a file but someone deletes it between the time you check if it exists and the time the read operation begins. By declaring a checked exception, you are telling the caller to anticipate this failure.
2. **Reasonable to recover from**: There is no point telling callers to anticipate exceptions that they cannot recover from. If a user attempts to read from non-existing file, the caller can prompt them for a new filename. On the other hand, if the method fails due to a programming bug (invalid method arguments or buggy method implementation) there is nothing the application can do to fix the problem in mid-execution. The best it can do is log the problem and wait for the developer to fix it at a later time.

* **Unchecked exceptions can be considered fail-fast. failure due to the caller**
* **Checked Exception can be considered as fail-safe. Damage control, failure due to callee.**
* **If an error is a programmer error, it must be an Unchecked Exception**.
* **If an error is not a programmer error and the reason is coming from external, it must be a Checked Exception.**

**Checked exceptions** are useful when the programmer did everything right, validated the input, ran tests, and all the code is perfect, but the code connects to a third party webservice that may be down ( or a file you were using was deleted by another external process etc ) . The webservice may even be validate before the connection is attempted, but during the data transfer something went wrong. In that scenario there is nothing that you or your co-workers can do to help it. But still you have to do something and not let the application just die and disappear in the eyes of the user. You use a checked exception for that and handle the exception

**Use the right abstraction level**. For example, a code repository with two different implementations (database and filesystem) should avoid exposing implementation-specific details (class name, method name and other details) by throwing SQLException or IOException. Instead, it should wrap the exception in an abstraction that spans all implementations (e.g. RepositoryException). If you expose the implementation details, it breaks encapsulation.

When an exception occurs, you have to either catch and handle the exception, or tell compiler that you can't handle it by declaring that your method throws that exception

**What is caller and callee?**

**A caller is a function that calls another function; a callee is a function that was called**. The currently-executing function is a callee, but not a caller.

**Rule 1**: Think of **a Unchecked Exception as a testable condition** before code executes. for example…

x.doSomething(); // the code throws a NullPointerException

where x is null... …the code should possibly have the following…

**Rule 2**: Think of a **Checked Exception as an un-testable** condition that may occur while the code executes.

Socket s = new Socket(“google.com”, 80);

InputStream in = s.getInputStream();

OutputStream out = s.getOutputStream();

in the example above, the URL (google.com) may be unavailable to due the DNS server being down

**Errors : IOError, AssertionError, NoClassDefinitionFoundError, NoSuchMethodError**

**According James Gosling**, The creator of Java himself thinks that Checked Exceptions are a good idea because although you can ignore them, you have to willfully do it. You can’t accidentally say “I don’t care”. You have to explicitly say, “I don’t care”.

**Why I think CHECKED Exceptions are OK !**

**Break Encapsulation ?:** One of the main arguments against using checked exceptions is that they break encapsulation by **revealing implementation details**, as well as forcing a **tighter coupling between the method and it callers**. Throw a super class exception, rather than several exception classes.

Effective Exceptions article by Barry Ruzek, he differentiates faults and contingencies.

* **Contingency** - **An expected condition demanding an alternative response from a method** that can be expressed in terms of the method's intended purpose. The caller of the method expects these kinds of conditions and has a strategy for coping with them.
* **Fault** - **An unplanned condition that prevents a method from achieving its intended purpose** that cannot be described without reference to the method's internal implementation.

In his example of a CheckingAccount class in a banking system, stop payment orders or overdrafts are contingencies, while the database being down or a network cable unplugged are faults. Based on this breakdown, contingencies should be handled with checked exceptions and faults with unchecked exceptions.

**If the situation is not likely to be recoverable, throw an unchecked exception, if you can, throw a checked exception.**

**Throw exceptions early**

Exceptions should be thrown as early as possible. As soon as you detect the error condition, the exception needs to be generated.

**Catch exceptions late**

we can say "throw exceptions early" and "catch exceptions late"

Common Java wisdom gives us a few general principles about how to work with exceptions:

1. Don’t use exceptions for flow control and expected business situations, but only for exceptional situations
2. Use unchecked exceptions for programming errors, bugs, situations where the application or the current operation cannot expect to recover from.
3. Use checked exceptions for situations where the called is expected to be able to make a full or partial recovery after the failure.

**Solutions -1** . Normal business scenario, don’t use exceptions, You can simply use an **IF condition to avoid exception** handling.

**Preserve encapsulation** – This means you should not throw exceptions that reveal API implementation details(ClassName and method name and other details). Sometimes it is fine to throw standard Java exceptions from your API. If I am implementing a file management library that builds on Java IO classes it is fine to propagate IOException instances. On the other hand if I implement a data management API that happens to use various data sources like files, database and sockets then probably I want to hide all the implementation and provide my own exception classes.

**Design for usability** – Do not throw a huge number of unrelated exceptions that will force the clients to have multiple catch blocks doing the same thing. In the ideal case you should have for each API or framework a top level exception class defined (possible abstract).

Log exceptions once at highest level possible level or when context details are needed and they can be lost.

**Checked exceptions were embraced by those who also valued strong typing in Java**. For example, the java.io package relies heavily on the checked exception IOException.

A successful fault handling framework has to accomplish four goals:

* Minimize code clutter
* Capture and preserve diagnostics
* Alert the right person
* Exit the activity gracefully

.

**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"!

**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. 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.**

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.

**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.

Again, I don't understand this logic. If I want to do something when method save() 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.

[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.

|  |  |  |
| --- | --- | --- |
|  | **Unpredictable** | **Predictable** |
| **Cont.** | Rare but normal situations due to conditions external to the code, such as a network outage or a file system issue.   * FileNotFoundException * PrinterException * SocketTimeoutException | Situations that are out of the ordinary but not *exceptional* per se.   * IllegalArgumentException * NumberFormatException * MalformedURLException |
| **Fault** | Failures due to external conditions. These are hard for the code to do something about, and in some cases it's even a bad idea to try.   * OutOfMemoryError * StackOverflowError * NoSuchMethodError | Prorgamming errors. These situations are due to bugs and should never have occurred in the first place.   * ArrayIndexOutOfBounds * NullPointerException * ArithmeticException |

| Condition | Contingency | Fault |
| --- | --- | --- |
| Is considered to be | A part of the design | A nasty surprise |
| Is expected to happen | Regularly but rarely | Never |
| Who cares about it | The upstream code that invokes the method | The people who need to fix the problem |
| Examples | Alternative return modes | Programming bugs, hardware malfunctions, configuration mistakes, missing files, unavailable servers |
| Best Mapping | A checked exception | An unchecked exception |