Exceptions – Consolidated

**Unchecked exceptions** represent exceptional conditions that the application usually cannot anticipate or recover from and so methods are not obliged to establish a policy to handle then.

Errors : IOError, AssertionError, NoClassDefinitionFoundError, NoSuchMethodError

**According James Gosling**, The creater 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”.

**According to Joshua Bloch**,

* Only use Exceptions for exceptional conditions ( Never use for ordinary flow )
* Use checked exceptions for recoverable conditions and runtime exceptions for programming error.
* Avoid unnecessary use of checked exceptions.

**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.

Imagine a method that is tasked with retrieving a user account. The method may not be able to complete for a number of reasons such as the account being locked, having been deleted or the current user may not have permission to access it. These outcomes aren’t exactly exceptional conditions in that they are all bound to happen at some point during the normal use of the software, so it is still correct to use exceptions and we have to use only checked exceptions. Effective Exceptions article by Barry Ruzek, he differentiates faults and contingencies, defining the latter as “an expected condition demanding an alternative response from a method that can be expressed in terms of the method’s intended purpose and advocates using checked exceptions to enable the caller of the method to provide strategy for coping with these expected outcomes.

One way to avoid their unnecessary use is to use a state testing method before calling the method that could potentially throw the exception. That way, if the preconditions is broken, it is a violation of the API and therefore it is perfectly OK to response with an unchecked exceptions, avoiding the need for the exception to be declared in the throws clause or explicitly dealt with in a catch. This is an approach advocated by Joshua Bloch in Effective Java and used in Java API, e.g. iterator.hasNext() being called before iterator.next().

**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.

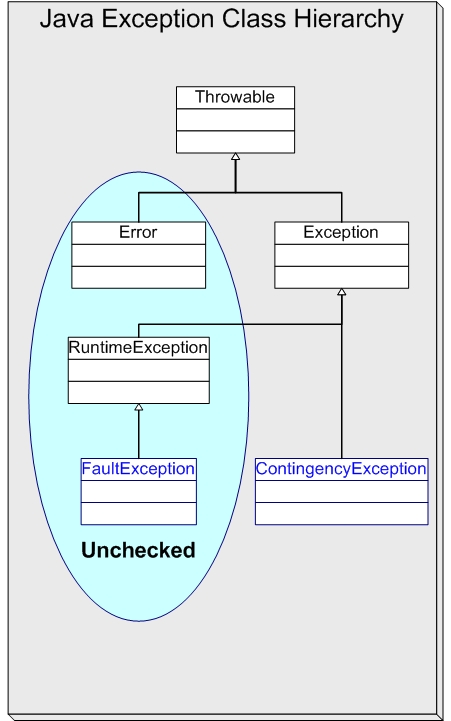
**Preserve encapsulation** – This means you should not throw exceptions that reveal API implementation details. Sometimes it is fine to throw standard Java exceptions from your API. If I am implementing a file management library that builds on Java IO classes it is fine to propagate IOException instances. 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.

**Effective Java Exceptions – By Barry Ruzek**

Some have argued that checked exceptions in the Java language are an experiment that failed. This article argues that the fault does not lie with the Java model, but with Java library designers who failed to acknowledge the two basic causes of method failure. It advocates a way of thinking about the nature of exceptional conditions and describes design patterns that will help your design.



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. At least 63 Java library packages issue this exception, either directly or through one of its dozens of subclasses. An I/O failure is a serious but extremely rare event. On top of that, there is usually nothing your code can do to recover from one. Java programmers found themselves forced to provide for IOException and similar unrecoverable events that could possibly occur in a simple Java library method call.

### **Faults and Contingencies**

Consider a CheckingAccount class within an imaginary banking application. A CheckingAccount belongs to a customer, maintains a current balance, and is able to accept deposits, accept stop payment orders on checks, and process incoming checks. ACheckingAccount object must coordinate accesses by concurrent threads, any of which may alter its state. CheckingAccount'sprocessCheck() method accepts a Check object as an argument and normally deducts the check amount from the account balance. But a check-clearing client that calls processCheck() must be ready for two contingencies. First, the CheckingAccount may have a stop payment order registered for the check. Second, the account may not have sufficient funds to cover the check amount.

So, the processCheck() method can respond to its caller in three possible ways. The nominal response is that the check gets processed and the result declared in the method signature is returned to the invoking service. The two contingency responses represent very real situations in the banking domain that need to be communicated to the check-clearing client. All three processCheck()responses were designed intentionally to model the behavior of a typical checking account.

The natural way to represent the contingency responses in Java is to define two exceptions, say StopPaymentException andInsufficientFundsException. It wouldn't be right for a client to ignore these, since they are sure to be thrown in the normal operation of the application. They help express the full behavior of the method just as importantly as the method signature.

Clients can easily handle both kinds of exception. If payment on a check is stopped, the client can route the check for special handling. If there are insufficient funds, the client can transfer funds from the customer's savings account to cover the check and try again.

The contingencies are expected and natural consequences of using the CheckingAccount API. They do not represent a failure of the software or of the execution environment. Contrast these with actual failures that could arise due to problems related to the internal implementation details of the CheckingAccount class.

Imagine that CheckingAccount maintains its persistent state in a database and uses the JDBC API to access it. Almost every database access method in that API has the potential to fail for reasons unrelated to the implementation of CheckingAccount. For example, someone may have forgotten to turn on the database server, unplugged a network cable, or changed the password needed to access the database.

JDBC relies on a single checked exception, SQLException, to report everything that could possibly go wrong.

**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.

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

**Use unchecked exceptions for fault conditions**

Your fault handling strategy should recognize that methods in the Java library and other APIs may use checked exceptions to represent what could only be fault conditions in the context of your application. In this case, adopt the architectural convention to catch the API exception where it happens, treat it as a fault, and throw an unchecked exception to signal the fault condition and capture diagnostic information.

The specific exception type to throw in this situation should be defined by your architecture. Don't forget that the primary purpose of a fault exception is to convey diagnostic information that will be recorded to help people figure out what went wrong. Using multiple fault exception types is probably overkill, since your architecture will treat them all identically. A good, descriptive message embedded inside a single fault exception type will do the job in most cases. It's easy to defend using Java's generic RuntimeException to represent your fault conditions. As of Java 1.4, RuntimeException, like all throwables, supports exception chaining, allowing you to capture and report a fault-inducing checked exception.