**Effective Java Exceptions**

by Barry Ruzek   
01/10/2007

**Abstract**

One of the most important architectural decisions a Java developer can make is how to use the Java exception model. Java exceptions have been the subject of considerable debate in the community. 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. Finally, it discusses exception handling as a crosscutting concern in the Aspect Oriented Programming model. Java exceptions are a great benefit when they are used correctly. This article will help you do that.

**Why Exceptions Matter**

Exception handling in a Java application tells you a lot about the strength of the architecture used to build it. Architecture is about decisions made and followed consistently at all levels of an application. One of the most important decisions to make is the way that the classes, subsystems, or tiers within your application will communicate with each other. Java exceptions are the means by which methods communicate alternative outcomes for an operation and therefore deserve special attention in your application architecture.

A good way to measure the skill of a Java architect and the development team's discipline is to look at exception handling code inside their application. The first thing to observe is how much code is devoted to catching exceptions, logging them, trying to determine what happened, and translating one exception to another. Clean, compact, and coherent exception handling is a sign that the team has a consistent approach to using Java exceptions. When the amount of exception handling code threatens to outweigh everything else, you can tell that communication between team members has broken down (or was never there in the first place), and everyone is treating exceptions "their own way."

The results of *ad hoc* exception handling are very predictable. If you ask team members why they threw, caught, or ignored an exception at a particular point in their code, the response is usually, "I didn't know what else to do." If you ask them what would happen if an exception they are coding for actually occurred, a frown follows, and you get a statement similar to, "I don't know. We never tested that."

You can tell if a Java component has made effective use of Java exceptions by looking at the code of its clients. If they contain reams of logic to figure out when an operation failed, why it failed, and if there's anything to do about it, the reason is almost always because of the component's error reporting design. Flawed reporting produces lots of "log and forget" code in clients and rarely anything useful. Worst of all are the twisted logic paths, nested try/catch/finally blocks, and other confusion that results in a fragile and unmanageable application.

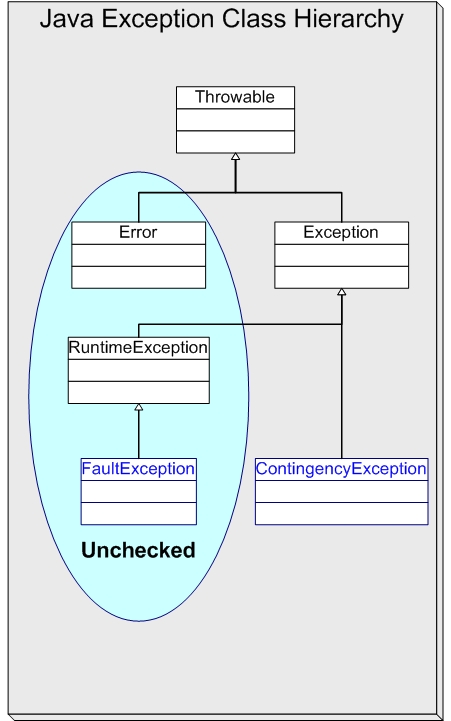
Addressing exceptions as an afterthought (or not addressing them at all) is a major cause of confusion and delay in software projects. Exception handling is a concern that cuts across all parts of a design. Establishing architectural conventions for exceptions should be among the first decisions made in your project. Using the Java exception model properly will go a long way toward keeping your application simple, maintainable, and correct.

**Challenging the Exception Canon**

What constitutes "proper use" of Java's exception model has been the subject of considerable debate. Java was not the first language to support exception-like semantics; however, it was the first language in which the compiler enforced rules governing how certain exceptions were declared and treated. Compile-time exception checking was seen by many as an aid to precise software design that harmonized nicely with other language features. Figure 1 shows the Java exception hierarchy.

In general, the Java compiler forces a method that throws an exception based on java.lang.Throwable including that exception in the "throws" clause in its declaration. Also, the compiler verifies that clients of the method either catch the declared exception type or specify that they throw that exception type themselves. These simple rules have had far-reaching consequences for Java developers world-wide.

The compiler relaxes its exception checking behavior for two branches of the Throwable inheritance tree. Subclasses of java.lang.Error and java.lang.RuntimeException are exempt from compile-time checking. Of the two, runtime exceptions are usually of greater interest to software designers. The term "unchecked" exception is applied to this group to distinguish it from all other "checked" exceptions.

  
*Figure 1. Java exception hierarchy*

I imagine that checked exceptions were embraced by those who also valued strong typing in Java. After all, compiler-imposed constraints on data types encouraged rigorous coding and precise thinking. Compile-time type checking helped prevent nasty surprises at run-time. Compile-time exception checking would work similarly, reminding developers that a method had potential alternate outcomes that needed to be addressed.

Early on, the recommendation was to use checked exceptions wherever possible to take maximum advantage of the help provided by the compiler to produce error-free software. The designers of the Java library API evidently subscribed to the checked exception canon, using these exceptions extensively to model almost any contingency that could occur in a library method. Checked exception types still outnumber unchecked types by more than two to one in the J2SE 5.1 API Specification.

To programmers, it seemed like most of the common methods in Java library classes declared checked exceptions for every possible failure. 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. Catching these exceptions added clutter to what should be simple code because there was very little that could be done in a catch block to help the situation. Not catching them was probably worse since the compiler required that you add them to the list of exceptions your method throws. This exposes implementation details that good object-oriented design would naturally want to hide.

This no-win situation resulted in most of the notorious exception handling anti-patterns we are warned about today. It also spawned lots of advice on the right ways and the wrong ways to build workarounds.

Some Java luminaries started to question whether Java's checked exception model was a failed experiment. Something failed for sure, but it had nothing to do with including exception checking in the Java language. The failure was in the thinking by the Java API designers that most failure conditions were the same and could be communicated by the same kind of exception.

|  |  |
| --- | --- |
| Pages: **1**, [2](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions2-097044.html), [3](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions3-089471.html) | [Next Page »](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions2-097044.html) |

**Effective Java Exceptions**   
Pages: [1](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html), **2**, [3](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions3-089471.html)

### 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. A CheckingAccount object must coordinate accesses by concurrent threads, any of which may alter its state. CheckingAccount's processCheck() 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 and InsufficientFundsException. 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. Most of what could go wrong has to do with configuring the database, the connectivity to it, and the hardware it resides on. There's nothing that the processCheck() method could do to deal with these situations in a meaningful way. That's a shame, because processCheck() at least knows about its own implementation. Upstream methods in the call stack have an even smaller chance of being able to address problems.

The CheckingAccount example illustrates the two basic reasons that a method execution can fail to return its intended result. They are worthy of some descriptive terms:

**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.

Using this terminology, a stop payment order and an overdraft are the two possible contingencies for the processCheck() method. The SQL problem represents a possible fault condition. The caller of processCheck() ought to have a way of providing for the contingencies, but could not be reasonably expected to handle the fault, should it occur.

### Mapping Java Exceptions

Thinking about "what could go wrong" in terms of contingencies and faults will go a long way toward establishing conventions for Java exceptions in your application architecture.

|  |  |  |
| --- | --- | --- |
| 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 |

Contingency conditions map admirably well to Java checked exceptions. Since they are an integral part of a method's semantic contract, it makes sense to enlist the compiler's help to ensure that they are addressed. If you find that the compiler is "getting in the way" by insisting that contingency exceptions be handled or declared when it is inconvenient, it's a sure bet that your design could use some refactoring. That's actually a good thing.

Fault conditions are interesting to people but not to software logic. Those acting in the role of "software proctologist" need information about faults to diagnose and fix whatever caused them to happen. Therefore, unchecked Java exceptions are the perfect way to represent faults. They allow fault notifications to percolate untouched through all methods on the call stack to a level specifically designed to catch them, capture the diagnostic information they contain, and provide a controlled and graceful conclusion to the activity. The fault-generating method is not required to declare them, upstream methods are not required to catch them, and the method's implementation stays properly hidden—all with a minimum of code clutter.

Newer Java APIs such as the Spring Framework and the Java Data Objects library have little or no reliance on checked exceptions. The Hibernate ORM framework redefined key facilities as of release 3.0 to eliminate the use of checked exceptions. This reflects the realization that the great majority of the exception conditions that these frameworks report are unrecoverable, stemming from incorrect coding of a method call, or a failure of some underlying component such as a database server. Practically speaking, there is almost no benefit to be gained by forcing a caller to catch or declare such exceptions.

**Fault handling in your architecture**

The first step toward handling faults effectively in your architecture is to admit that you need to do it. Coming to this acceptance is difficult for engineers who take pride in their ability to create impeccable software. Here is some reasoning that will help. First, your application will be spending a great deal of time in development where mistakes are commonplace. Providing for programmer-generated faults will make it easier for your team to diagnose and fix them. Second, the (over)use of checked exceptions in the Java library for fault conditions will force your code to deal with them, even if your calling sequences are completely correct. If there's no fault handling framework in place, the resulting makeshift exception handling will inject entropy into your application.

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

Faults are a distraction from your application's real purpose. Therefore, the amount of code devoted to processing them should be minimal and, ideally, isolated from the semantic parts of the application. Fault processing must serve the needs of the people responsible for correcting them. They need to know that a fault happened and get the information that will help them figure out why. Even though a fault, by definition, is not recoverable, good fault handling will attempt to terminate the activity that encountered the fault in a graceful way.

**Use unchecked exceptions for fault conditions**

There are lots of reasons to make the architectural decision to represent fault conditions with unchecked exceptions. The Java runtime rewards programming mistakes by throwing RuntimeException subclasses such as ArithmeticException and ClassCastException, setting a precedent for your architecture. Unchecked exceptions minimize clutter by freeing upstream methods from the requirement to include code for conditions that are irrelevant to their purpose.

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.

You may choose to define your own unchecked exception for the purpose of fault reporting. This would be necessary if you need to use Java 1.3 or earlier versions that do not support exception chaining. It is simple to implement a similar chaining capability to capture and translate checked exceptions that constitute faults in your application. Your application may have a need for special behavior in a fault reporting exception. That would be another reason to create a subclass of RuntimeException for your architecture.

**Establish a fault barrier**

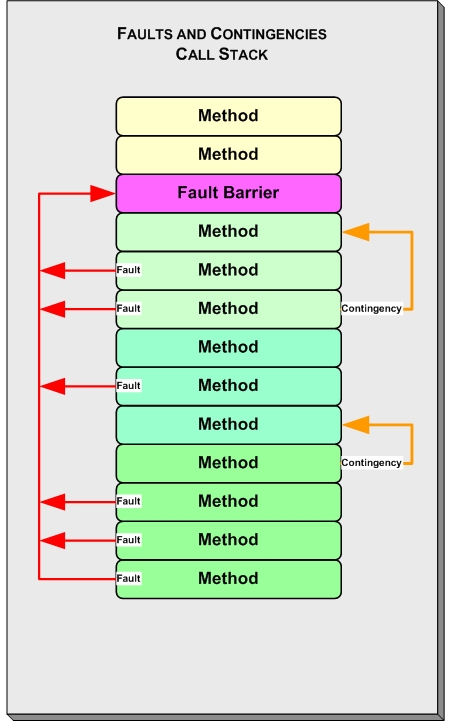
Deciding which exception to throw and when to throw it are important decisions for your fault-handling framework. Just as important are the questions of when to catch a fault exception and what to do afterward. The goal here is to free the functional portions of your application from the responsibility of processing faults. Separation of concerns is generally a good thing, and a central facility responsible for dealing with faults will pay benefits down the road.

In the fault barrier pattern, any application component can throw a fault exception, but only the component acting as the "fault barrier" catches them. Adopting this pattern eliminates much of the intricate code that developers insert locally to deal with faults. The fault barrier resides logically toward the top of the call stack where it stops the upward propagation of an exception before default action is triggered. Default action means different things depending on the application type. For a stand-alone Java application, it means that the active thread is terminated. For a Web application hosted by an application server, it means that the application server sends an unfriendly (and embarrassing) response to the browser.

The first responsibility of a fault barrier component is to record the information contained in the fault exception for future action. An application log is by far the best place to do this. The exception's chained messages, stack traces, and so on, are all valuable pieces of information for diagnosticians. The worst place to send fault information is across the user interface. Involving the client of your application in your debugging process is hardly ever good for you or your client. If you are really tempted to paint the user interface with diagnostic information, it probably means that your logging strategy needs improvement.

The next responsibility of a fault barrier is to close out the operation in a controlled manner. What that means is up to your application design but usually involves generating a generic response to a client that may be waiting for one. If your application is a Web service, it means building a SOAP <fault> element into the response with a <faultcode> of soap:Server and a generic <faultstring> failure message. If your application communicates with a Web browser, the barrier would arrange to send a generic HTML response indicating that the request could not be processed.

In a Struts application, your fault barrier can take the form of a global exception handler configured to process any subclass of RuntimeException. Your fault barrier class will extend org.apache.struts.action.ExceptionHandler, overriding methods as needed to implement the custom processing you need. This will take care of inadvertently generated fault conditions and fault conditions explicitly discovered during the processing of a Struts action. Figure 2 shows contingency and fault exceptions.

  
Figure 2. Contingency and fault exceptions

If you are using the Spring MVC framework, your fault barrier can easily be built by extending SimpleMappingExceptionResolver and configuring it to handle RuntimeException and its subclasses. By overriding the resolveException() method, you can add any custom handling you need before using the superclass method to route the request to a view component that sends a generic error display.

When your architecture includes a fault barrier and developers are made aware of it, the temptation to write one-off fault exception handling code decreases dramatically. The result is cleaner and more maintainable code throughout your application.

|  |  |
| --- | --- |
| Pages: [1](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html), **2**, [3](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions3-089471.html) | [Next Page »](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions3-089471.html) |

**Effective Java Exceptions**   
Pages: [1](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html), [2](http://www.oracle.com/technetwork/articles/entarch/effective-exceptions2-097044.html), **3**

**Contingency Handling in Your Architecture**

With fault processing relegated to the barrier, contingency communication between primary components becomes much simpler. A contingency represents an alternative method result that is just as important as the principal return result. Therefore, checked exception type is a good vehicle to convey the existence of a contingency condition and supply the information needed to contend with it. This practice enlists the help of the Java compiler to remind developers of all aspects of the API they are using and the need to provide for the full range of method outcomes.

It is possible to convey simple contingencies by using a method's return type alone. For example, returning a null reference instead of an actual object can signify that the object could not be created for a defined reason. Java I/O methods typically return an integer value of -1 instead of a byte value or byte count to indicate an end-of-file condition. If your method's semantics are simple enough to allow it, alternative return values may be the way to go, since they eliminate the overhead that comes with exceptions. The downside is that the method caller is responsible for testing the return value to see if it is a primary result or a contingency result. The compiler will not insist that the method caller makes that test, however.

If a method has a void return type, an exception is the only way to indicate that a contingency occurred. If a method is returns an object reference, the vocabulary that the return value can express is limited to two values (null and non-null). If a method returns an integral value, it may be possible to express several contingency conditions by choosing values that are guaranteed not to conflict with the primary return values. But now we have entered the world of error code checking, something the Java exception model was developed to avoid.

**Supply something useful**

It made little sense to define different fault reporting exception types, since the fault barrier treats them all the same. Contingency exceptions are quite different, because they are meant to convey diverse conditions to method callers. Your architecture would probably specify that these exceptions should all extend java.lang.Exception or a designated base class that does.

Do not forget your exceptions are complete Java types that can accommodate specialized fields, methods, and even constructors that can be shaped for your unique purposes. For example, the InsufficientFundsException type thrown by the imaginary CheckingAccount processCheck() method could include an OverdraftProtection object that is able to transfer funds needed to cover the shortfall from another account whose identity depends on how the checking account is set up.

**To log or not to log**

Logging fault exceptions makes sense because their purpose is to draw the attention of people to situations that need to be corrected. The same cannot be said for contingency exceptions. They may represent relatively rare events, but every one of them is expected to happen during the life of your application. If anything, they signify that the application is working the way it was designed to work. Routinely adding logging code to contingency catch blocks adds clutter to your code with no actual benefit. If a contingency represents a significant event, it is probably better for a method to generate a log entry recording the event before throwing a contingency exception to alert its caller.

**Exception Aspects**

In Aspect Oriented Programming (AOP) terms, fault and contingency handling are crosscutting concerns. To implement the fault barrier pattern, for example, all the participating classes must follow common conventions:

* The fault barrier method must reside at the head of a graph of method calls that traverses the participating classes.
* They must all use unchecked exceptions to signify fault conditions.
* They must all use the specific unchecked exception types that the fault barrier is expecting to receive.
* They all must catch and translate checked exceptions from lower methods that are deemed to be faults in their execution context.
* They must not interfere with the propagation of fault exceptions on their way to the barrier.

These concerns cut across the boundaries of otherwise unrelated classes. The result is minor bits of scattered fault handling code and implicit coupling between the barrier class and the participants (although still a great improvement over not using a pattern at all!). AOP allows the fault handling concern to be encapsulated in a common Aspect applied to the participating classes. Java AOP frameworks such as AspectJ and Spring AOP recognize exception handling as a join point to which fault handling behavior (or advice) can be attached. In this way, the conventions that bind participants in the fault barrier pattern can be relaxed. Fault processing can now reside within an independent, out-of-line aspect, eliminating the need for a "barrier" method to be placed at the head of a method invocation sequence.

If you are exploiting AOP in your architecture, fault and contingency handling are ideal candidates for aspects that apply throughout an application. A full exploration of how fault and contingency handling could work in the AOP world would make an interesting topic for a future article.

**Conclusion**

Although the Java exception model has generated spirited discussion during its lifetime, it provides excellent value when it is applied correctly. As an architect, it is up to you to establish conventions that get the most from the model. Thinking of exceptions in terms of faults and contingencies can help you make the right choices. Using the Java exception model properly will keep your application simple, maintainable, and correct. Aspect Oriented Programming techniques may offer some definite advantages for your architecture by recognizing fault and contingency handling as crosscutting concerns.

**References**

* [Sun's Exception Tutorial](http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html) - all of the basics on Java exceptions
* [Does Java Need Checked Exceptions?](http://www.mindview.net/Etc/Discussions/CheckedExceptions) - Bruce Eckel argues against checked exceptions in Java
* [Exceptional Java](http://www.octopull.demon.co.uk/java/ExceptionalJava.html) - a good discussion of these topics and an architectural exceptions policy to emulate
* [The Exceptions Debate](http://www-128.ibm.com/developerworks/java/library/j-jtp05254.html) - more exceptions background from developerWorks
* [The Apache Struts Web Application Framework](http://struts.apache.org/1.1/index.html) - the source for Struts information
* [The Spring Framework](http://www.springframework.org/) - the source for the information on Spring
* [Wikipedia: Aspect Oriented Programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming) - a good introduction to AOP concepts
* [The AspectJ Project](http://www.eclipse.org/aspectj/) - the source for the information on AspectJ

*Barry Ruzek has been named a Master Certified IT Architect by the Open Group. He has over 30 years of experience developing operating systems and enterprise applications.*