Effective Java Exceptions

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.

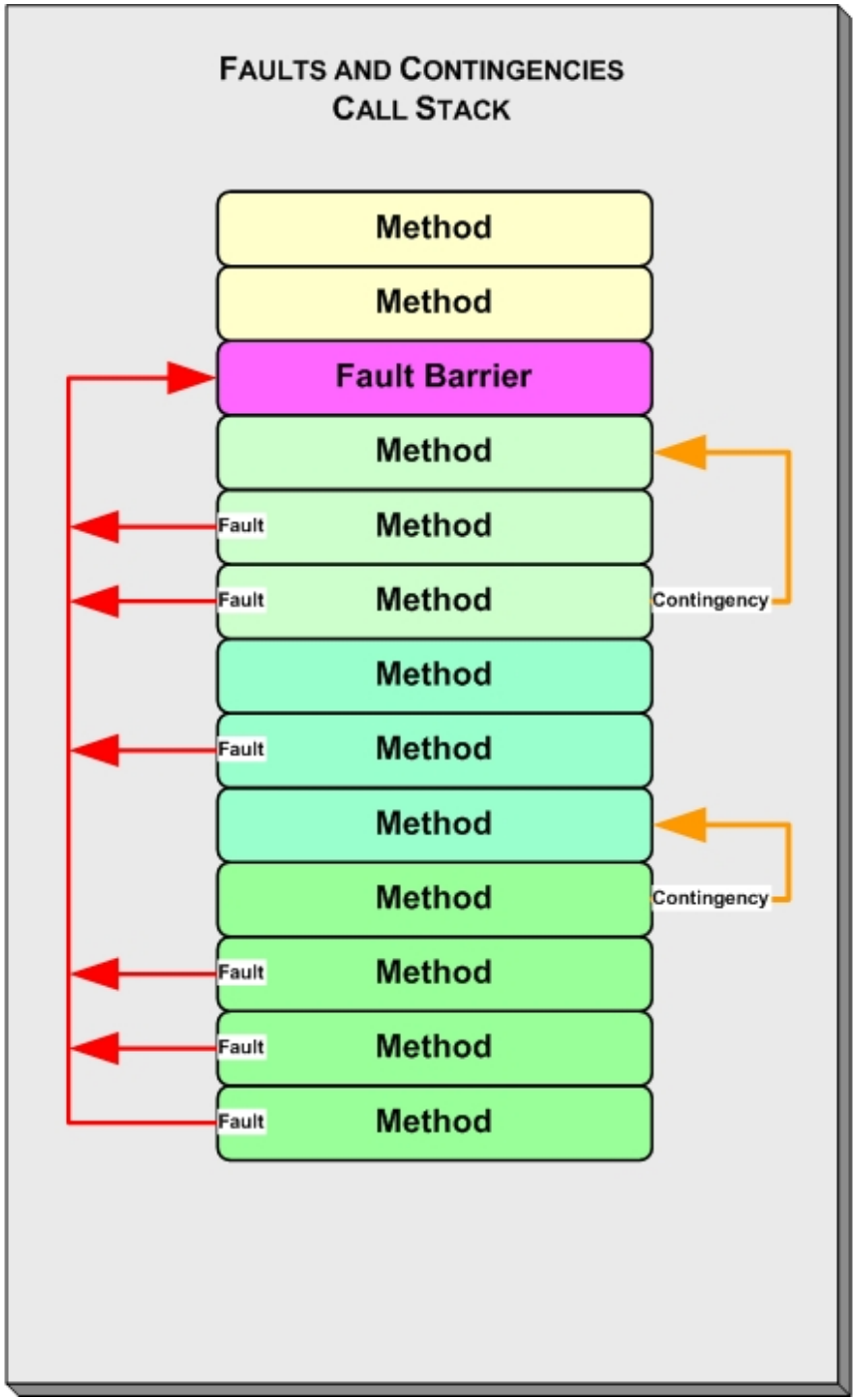

Figure 2. Contingency and fault exceptions

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.