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.

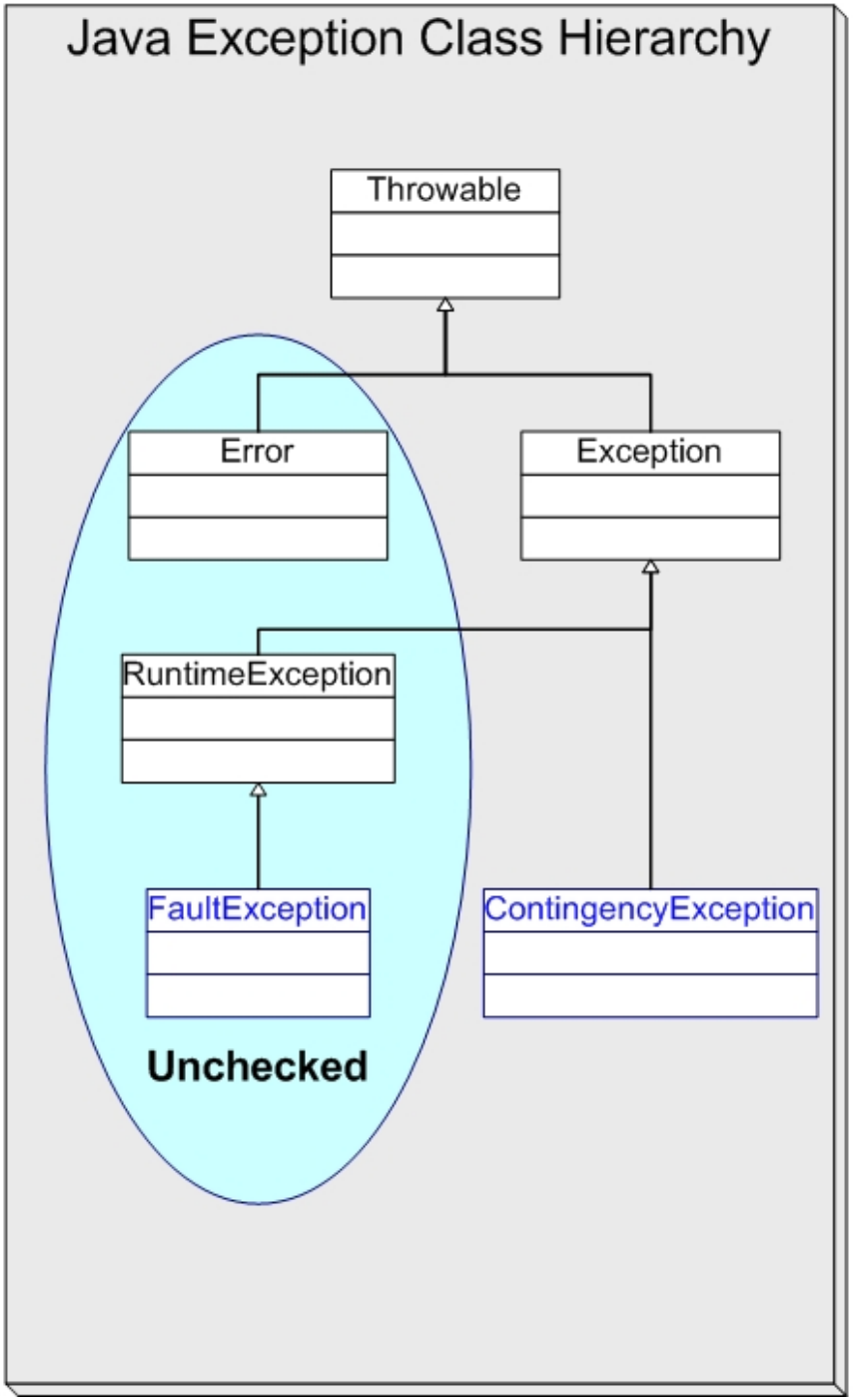

Figure 1. Java exception hierarchy

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.

false ,,,,,,,,,,,,,,