## Java Style: Properly handling exceptions

Asked 15 years, 11 months ago Modified 11 years, 8 months ago Viewed 11k times



I keep getting stuck conceptually on deciding an Exception-handling structure for my project.

**17** 

Suppose you have, as an example:



```
public abstract class Data {
  public abstract String read();
```



And two subclasses FileData, which reads your data from some specified file, and StaticData, which just returns some pre-defined constant data.

Now, upon reading the file, an IOException may be thrown in FileData, but StaticData will never throw. Most style guides recommend propagating an Exception up the call stack until a sufficient amount of context is available to effectively deal with it.

But I don't really want to add a throws clause to the abstract read() method. Why? Because Data and the complicated machinery using it knows nothing about files, it just knows about Data. Moreover, there may be other Data subclasses (and more of them) that never throw exceptions and deliver data flawlessly.

On the other hand, the IOException is necessary, for if the disk is unreadable (or some such) an error *must* be thrown. So the only way out that I see is catching the IOException and throwing some RuntimeException in its place.

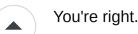
Is this the correct philosophy?

exception java Share Follow edited Jan 8, 2009 at 19:25 asked Jan 8, 2009 at 18:10 **Greg Mattes** Jake **33.9k** • 15 • 76 • 105 15.2k • 22 • 73 • 86

Sorted by:

Highest score (default)

5 Answers







The exception should be at the same level of abstraction where is used. This is the reason why since java 1.4 Throwable supports exception chaining. There is no point to throw FileNotFoundException for a service that uses a Database for instance, or for a service that is "store" agnostic.





It could be like this:



```
public abstract class Data {
   public abstract String read() throws DataUnavailableException;
}
class DataFile extends Data {
    public String read() throws DataUnavailableException {
        if( !this.file.exits() ) {
            throw new DataUnavailableException( "Cannot read from ", file );
         }
         try {
         } catch( IOException ioe ) {
             throw new DataUnavailableException( ioe );
         } finally {
              . . .
         }
 }
class DataMemory extends Data {
    public String read() {
        // Everything is performed in memory. No exception expected.
    }
 }
 class DataWebService extends Data {
      public string read() throws DataUnavailableException {
           // connect to some internet service
           try {
           } catch( UnknownHostException uhe ) {
              throw new DataUnavailableException( uhe );
      }
 }
```

Bear in mind that if you program with inheritance in mind, you should design carefully for the specific scenarios and test implementations with those scenarios. Obviously if it is harder to code an general purpose library, because you don't know how is it going to be used. But most of the times applications are constrained to an specific domain.

Should your new exception be Runtime or Checked? It depends, the general rule is to throw Runtime for programming errors and checked for recoverable conditions.

If the exception could be avoided by programming correctly ( such as NullPointerException or IndexOutOfBounds ) use Runtime

If the exception is due to some external resource out of control of the programmer ( the network is down for instance ) AND there is something THAT could be done ( Display a message of retry in 5 mins or something ) then a checked exception should be used.

If the exception is out of control of the programmer, but NOTHING can be done, you could use a RuntimeException. For instance, you're supposed to write to a file, but the file was deleted and you cannot re-create it or re-try then the program should fail ( there is nothing you can do about it ) most likely with a Runtime.

See these two items from Effective Java:

- Use checked exceptions for recoverable conditions and run-time exceptions for programming errors
- Throw exceptions appropriate to the abstraction

I hope this helps.

Share Follow



If DataUnavailableException is not a RuntimeException in your example, your example (DataMemory to be particular) won't compile. And reading your notes makes me think it should be a Checked Exception, since there is something that can be done about Data Not Available.. – Koray Tugay Dec 14, 2015 at 20:03



If you're not explicitly stating that <code>read()</code> can throw an exception, then you'll surprise developers when it does.



In your particular case I'd catch the underlying exceptions and rethrow them as a new exception class <code>DataException</code> Or <code>DataReadException</code>.



Share Follow



93.2k • 71 • 189 • 238



I'd make DataException hold the IOException within it, too, so the level that handles the exception can drill down if it needs to. – Paul Tomblin Jan 8, 2009 at 18:18

Paul, you don't need to do this explicitly - as of Java 1.4 you can chain exceptions using either the initCause(Throwable) method or by implementing the constructors that include Throwable as an argument. - sk. Jan 8, 2009 at 18:25



4

Throw the IDException wrapped in an exception type that is appropriate to the "Data" class. The fact is that the read method wont always be able to provide the

data, and it should probably indicate why. The wrapping exception may extend RuntimeException and therefore not need to be declared (although it should be



appropriately documented).

Share Follow



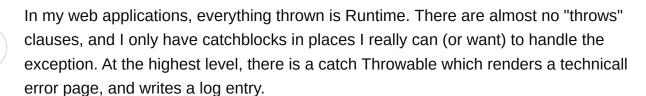
answered Jan 8, 2009 at 18:15

Tom Hawtin - tackline
147k • 30 • 221 • 312



Use Runtime Exceptions, combined with an exploding catch-all at the top. It's a bit scary at first, but gets better once you get used to it.







A Log4J mailer sends me the log entry and 10 log entries preceding it. So when the client calls, I usually already know that there was a problem.

With proper (unit)testing and clean programming, the added cleanliness and readability outweighs the loss of checked exceptions anytime.

Share Follow

this makes sense.

answered Jan 8, 2009 at 19:17





0

You *should* declare some kind of exception on the abstract read() method. The abstract class is effectively declaring an interface - and you already know from your two concrete subclasses that an implementation may well be unable to return successfully due to an exception condition.



Thus declaring some exception to be thrown in the abstract Data.read() method is entirely correct. Don't be tempted to simple declare that it throws IOException, as it shouldn't be tied to and of the specific implementations (else you'd have to declare that it could throw SQLException too in case you ever decide to have a database-reading subclass, SAXException in case you ever have an XML-based reader (that uses SAX), and so on). You'll need a custom exception of your own that adequately captures this at the abstract level - something like the DataException recommended above, or potentially reuse an existing custom exception from the same package if





Does Java allow an overriding method in a subclass to drop the throws declaration? If so, the subclasses that will never throw can be used without having to handle the exception if you're using them by their actual type (and not polymorphically through the parent). Dunno if that works though. – rmeador Jan 8, 2009 at 18:45