A pattern for using exceptions

Fault barrier

Do exceptions matter?

Exceptions matter a lot!

According to some, exceptions are one of the most important ingredients of systems architecture.

When we have interviewed people “in the industry”™, not knowing how to use the exception model of Java is one of the most common sources to problems with developers and the source of a great many bugs.

Why do exceptions matter?

A system consists of classes classes and subsystems (sometimes called tiers). How the tiers and classes communicate with each other is of course important (and tells a lot about the architecture!).

Normally, communication takes place through method calls. Methods are called, and they typically “return” an outcome of the call (or cause a side-effect).

But what about alternative outcomes of a method call?

This is what exceptions are all about! An alternative outcome of a method call.

Exceptions exist for a reason

- Exceptional events do occur in applications
- Often, return values are not enough
- Using special return values could be hidden knowledge (not intuitive)
- Some events are more or less catastrophic
- Some events are expected but not faults per se
- We need an alternative path from a method (return is not always perfect)
- With a structured and thought-through use of exceptions we can achieve a simple, maintainable and correct application design

Signs of problems...

- The amount of code dealing with exceptions start to outweigh the amount of code doing normal stuff
- You don’t know what would happen with the application if an exception you are handling actually occurs
- You do a lot of “catch and log (or worse: System.err.println)” and then nothing else
- No one in the project can explain how Exceptions will be used consistently throughout the application (what model is used)

There are a lot of checked exceptions in the API

Java's API uses checked exceptions *a lot*.

Only IOException occurs in 817 classes on my system, (OpenJDK Runtime Environment (IcedTea 2.6.7) (7u111-2.6.7-0ubuntu0.14.04.3)) or, put differently 157 packages contain code which throws IOException.

```
$ grep -r 'throws IOException'| awk '{print $1;}' | sort -u \
   | rev | cut -d '/' -f2- | rev | sort -u | wc -l
157
```

So what? Just try-catch!

The problem with all the checked exceptions being thrown from the API code, is that IOException declared to be thrown forces you to write a handler (which probably can't do so much about the problem) for events that should occur very rarely, or declare that your method too throws IOException (which is not problematic).

We'd prefer not to reveal such implementation details about IO to the callers of our methods. Why make calling code aware of our IO operations, and also make them deal with the unlikely event that they fail?

Calling methods will be faced with the same dilemma: handle or declare...

What do do, then?

Define an exceptions strategy for your application and make sure every developer knows about it and understands it.

The goal is to formulate a consistent way to deal with exceptions throughout the application.

One pattern you can use for such a strategy is the fault barrier pattern.

Faults and contingencies

A central concept of the fault barrier pattern is to consider two major categories of exceptional events:

Fault - unrecoverable errors to be handled in one place early in the call chain (close to main) by the so called fault barrier.

Contingencies - rare but expected conditions which are not real faults but rather alternative return paths from methods.

Faults are represented by runtime exceptions, contingencies by checked exceptions.

What’s the idea behind that?

The idea is (among other things) to let the programmers focus on contingencies (represented by checked exceptions) which are things they must prepare for and handle or declare and pass on to someone who can handle them.

Faults are defined as bad things from which we can’t recover, so they are represented as runtime exceptions which are simply thrown and will bubble up the call chain all the way to the fault barrier.

The fault barrier will notify the correct people, log stuff and exit gracefully.

How do we make faults runtime exceptions?

All exceptions that are deemed to be critical and unrecoverable are wrapped in a RuntimeException (which could be your own exception class extending RuntimeException or simply java.lang.RuntimeException) and re-thrown.

The developers must talk with each other and agree upon what constitutes a fault and what is not recoverable.

If an IOException is deemed to be a fault (maybe one super important file has gone missing), that IOException is to be caught immediately and wrapped inside a Runtime exception which is thrown from the catch-clause.

What are contingencies, then?

Contingencies are the things we can expect (but shouldn't be very common) and which need to be acted on by some code. To allow the methods to have return values for normal outcomes, contingencies are represented as checked exceptions declared to be thrown where they are discovered.

It is common to use your own custom exception classes for this, so that a library can be self-contained (contingencies are part of packages which are logically related to the event).

It is allowed to wrap other exceptions inside a contingency, for debugging and diagnostic purposes. You can make your exceptions rich (adding methods and stuff to them).

Another pattern related to all this

Another pattern (often part of the fault barrier pattern) is to always wrap low-level exceptions in custom exceptions belonging to the tier or module of the application which run into them.

This is to hide low level implementation decisions from client code.

If a method working on statistics runs into an SQLException, that's no business of the client code. Or put differently, the client code shouldn't be coupled with the JDBC API.

Better to wrap the SQLException (if it isn't critical) inside a StatisticsException (which you have made up yourself) so that it's part of the protocol.

How to look at exceptions

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

Source: http://www.oracle.com/technetwork/articles/entarch/effective-exceptions2-097044.html

Goals of the exception strategy

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

Source:

http://www.oracle.com/technetwork/articles/entarch/effective-exceptions2-097044.html

What not to do

Don’t do this:

```
// Oh, crap this method throws some checked exceptions
try{
  thatMethod();
}catch(Exception e){ // what can I do? }

// Honestly, do you recognize that from your own code?
```

Now, no-one knows what happened or that it happened!

This doesn’t help

```
// Oh, crap this method throws some checked exceptions
try{
  thatMethod();
}catch(Exception e){
  // Better log it
  System.err.println("Bad things happen to bad people" +
                        e.getMessage());
}

// Honestly, do you recognize that from your own code?
```

Now, only people who read stderr knows what happened and that it happened…

That’s it for now

Questions?