

Object Oriented Design: Exception Handling and Design Techniques

Copyright © 2004-2005 By Thomas J. Clancy
All Rights Reserved

Table of Contents

1 Introduction.....	3
2 Checked vs. Unchecked Exceptions.....	4
2.1 The Big Controversy.....	5
2.2 Checked and Unchecked Exceptions in Practice.....	6
2.3 Why Are Checked (or Unchecked) Exceptions a Hack?.....	10
3 Exception Handling Techniques.....	13
3.1 Do Not Create Black Holes.....	13
3.2 Do Not Catch for the Sake of Catching.....	15
4 Designing Exceptions.....	18

1 Introduction

Getting right to the point, this short paper discusses Exception handling and design techniques in the design of object oriented software and software libraries using the Java programming language as the vehicle for discussing such ideas. Thus it is biased towards that language and the syntax and some of the fundamental concepts regarding its very foundations; for example the notion of checked exceptions and the **finally** block. Be that as it may, some of the ideas in here are meant to apply generally to the idea of object orientation, exception handling and software design and how these three things relate.

In writing this I have assumed that the reader already has a good understanding of programming in Java or C++ or some current modern OO programming language and also a good grasp of what it means to create and handle exceptions using, for example in Java, the **throw**, **try** and **catch** keywords.

2 Checked vs. Unchecked Exceptions

Java has the notion of two kinds of exceptions, checked and unchecked, and while I consider Sun's “experiment” a hack (see why below), which is to say that the checked exceptions and unchecked exceptions are not two distinct types of exception classes, but rather that unchecked exceptions actually derive from checked ones and thus pose some slight but annoying problems of their own, the very idea of the two is nonetheless intriguing.

Checked exceptions are exceptions that are checked during compilation and are those that derive specifically from the class `java.lang.Exception`. An example of one should suffice for use throughout the documentation (please excuse the less than creative name).

```
package org.tclancy.apps.foo;

public class FooCheckedException extends Exception {

    public FooCheckedException() {
        super();
    }

    public FooCheckedException(String message) {
        super(message);
    }

    public FooCheckedException(String message, Throwable cause) {
        super(message, cause);
    }

    public FooCheckedException(Throwable cause) {
        super(cause);
    }
}
```

This example illustrates a simple extension of `java.lang.Exception`. What you should note here is that, for simplicity and for convenience, I overload each of the constructors of the `Exception` base class. When you start to design checked exception classes this is probably a good thing to do at first. Other techniques might dictate that you create just one or two kinds of constructors. It's really up to the requirements of your design, but mostly its a matter of taste and experience.

Unchecked, or runtime exceptions, are exceptions that derive from the `java.lang.RuntimeException` class, which itself derives from `java.lang.Exception`, and are not checked at compile time, and so their handling is not enforced by the compiler. Here is an example of one that looks similar to the checked exception,

```

package org.tclancy.apps.foo;

public class FooUncheckedException extends RuntimeException {

    public FooUncheckedException() {
        super();
    }

    public FooUncheckedException(String message) {
        super(message);
    }

    public FooUncheckedException(String message, Throwable cause) {
        super(message, cause);
    }

    public FooUncheckedException(Throwable cause) {
        super(cause);
    }
}

```

2.1 The Big Controversy

Because the handling of unchecked exceptions is not enforced by the compiler, you can freely throw unchecked exceptions from a method without having to specify them as part of the method “contract” or requirement—in other words your throws clause—and so this leaves the caller of your methods free to either handle these sorts of exceptions or not. And of course this is part of a wider controversy regarding the great experiment by Sun and the Java language, the only real modern object oriented programming language that enforces exception handling via its checked exceptions at compile time.

Some argue that checked exceptions lead to clearer, cleaner code in that it forces the developers to clearly think about handling the exceptions thrown by the classes that they employ in their own code. The idea is that because the compiler complains when you forget to handle an exception that can be thrown by calling some method, like the well seasoned, conscientious, and disciplined engineer that you ought to be you would handle the exception in a manner befitting it. In practice, however, this annoyance has led to more exception swallowing, or rather to the creation of black holes, than it has to encouraging proper exception handling. Does this look familiar?

```

try {
    // do something IOish
    ...
}
catch(java.io.IOException exc) {
    //do nothing
}

```

Thus the other side argue that checked exceptions aren't necessarily very good for this very reason, which is to say, don't trust the developer to do the job right. Throw unchecked exceptions, or in Java, throw `java.io.RuntimeException`-based exceptions and let the developers catch them if they want to. If they don't, the exception will be tossed all the way up and out and the JVM will get the exception. In this way, at least, the exception won't be swallowed up in the bowels of the code before it might have a chance to do some good. Which is to say that rather than the compiler dictating to you at compile time that you should be handling an exception, you can witness first hand what happens during run time when your code fails.

The problem with unchecked exceptions is that you are not forced to declare them in a method's signature (even though the language permits you to) and so there is no language-specific forcing you to declare this behavior as part of your contract to those who would use your class. As stated, you can still declare in a `throws` clause that you are throwing an unchecked exception.

```
public void doFoo() throws FooUncheckedException() {
    // do something
    throw new FooUncheckedException();
}
```

But the compiler still doesn't care at all if you handle it, hence why they are known as “unchecked.”

```
public void testDoFoo() {
    doFoo();
}
```

Because of this, I still believe that checked exceptions are a good thing and that although we tend to get lazy (yes I have included myself among the we), especially when we write experimental, trivial or prototype code, I think that having such enforcement and disciplining ourselves to become better at handling exceptions properly and in a standard way makes for better code. Of course I might be wrong and in the end will probably lose the argument because no other language will bother to include checked exceptions (in a dynamic language, really, why bother?) But for now, Java has them and so I say that they should still be used.

2.2 Checked and Unchecked Exceptions in Practice

When handling an exception, if you make a call to a method that throws a checked

exception, you are either forced to catch it or to declare as part of your method's signature in a throws clause that you plan to let the exception propagate back to the caller of your method.

In the first case we simply handle the exception using a try...catch block, and note that we do nothing very interesting within the catch portion. Normally we might log the exception, attempt in some way to recover from it or, if the exception thrown were from a different package, we might wrap it in a new package or class level exception (more on this below) and throw it instead.

```
public void handleCheckedException() {
    try {
        throwsACheckedException();
    }
    catch(FooCheckedException exc) {
        // uh-oh a black hole!
        System.out.println(exc.toString());
    }
}
```

Had we not surrounded the method call, throwsACheckedException, with the try...catch statements the compiler would have complained with an “Unhandled Exception” compilation error—fortunate for us most IDEs will allow us to correct this automatically.

A second option would be to simply allow the checked exception to pass out of our method which would force the caller of our method to handle the exception in some way.

```
/**
 * This method, when called, will throw a checked exception.
 *
 * @throws FooCheckedException The kind of exception thrown.
 */
public void letCheckedExceptionPassOn() throws FooCheckedException {
    throwsACheckedException();
}

...

public void testLetCheckedExceptionPassOn() {
    try {
        fooApp.letCheckedExceptionPassOn();
        fail("method should have thrown an exception.");
    }
    catch (FooCheckedException e) {
    }
}
```

In the first snippet of code I allow the FooCheckedException instance that is thrown by a

call to the `throwsACheckedException` method to pass out of my `letCheckedExceptionPassOn` method.

In a simple unit test case method I call my method and am forced to catch the exception (or to allow it to pass on out of my test case method) and handle it. In this case I am expecting the exception to be thrown and am treating it as an error if it isn't thrown. What's important to note here is that I explicitly make known (which I am forced to do anyway by convention) to the caller of my `letCheckedExceptionPassOn` method that I throw `FooCheckedException`.

The Java syntax isn't rich enough to indicate to the programmer looking at the method signature why or under what conditions this exception will be thrown. That's one of the reasons why documenting the methods in Java using the standard javadoc nomenclature is so important, and why, in general, documenting source code in any language using whatever documentation standards they provide is so vitally important.

Now, if you should decide, instead, to create and throw runtime exceptions you could do something like the following.

```
public void throwUncheckedException() {
    throw new FooUncheckedException(
        "FooApp.throwUncheckedException threw this.");
}

...

public void testThrowUncheckedException() {
    fooApp.throwUncheckedException();
}
```

`FooUncheckedException` is based on `java.util.RuntimeException` and so, by its very nature, is an unchecked exception. Thus just throwing one from inside the `throwUncheckedException` method gives you the freedom not to declare that you're doing so by not forcing you to say so in a `throws` clause in the method signature, even though you could; the syntax of the language doesn't prohibit this.

In the test method below it, except for the painfully obvious name of the method itself, I have no way of knowing that a call to the method will throw an exception, and it's only by a side-effect of testing the method that I've discovered that, indeed, this method throws, for some reason, an unchecked exception. Had this actually been a real test I would have been lucky enough (and so rarely am I) to have had the insight to actually

include a call to the method in my suite of tests and so would have been able to catch this fact early in the implementation process. Back to the classic argument, you could say that because this was an unchecked exception, finding the problem was inevitable because I would never have thought, due to my pathetic laziness, to just swallow the exception in a black hole during my initial development. Then again, if I am to learn anything by all of this, I should learn that properly handling all exceptions, especially checked exceptions, is vital to creating good code.

Then again Sun would have us believe, and perhaps rightly so, that because runtime exceptions are meant for things at the JVM level like arithmetic checks, pointer references such as when you try to access a null object, and array boundary checks (e.g. array index out of bounds), they tend to be too numerous and frequent and so there really is little benefit in enforcing their capture and handling. This is why there is a distinction.

Checked exceptions are meant to embody application level meaning such as document locking semantics, for example, in a multiuser document processing system, or something as simple as a security exception for when someone enters an incorrect password. You could argue, as I have tried to in the past (fruitlessly, mind you) that because application-level exceptions happen during runtime that the very name “`RuntimeException`” is a poor choice to describe the kind of exceptions that it embodies, but I think that in the case of Java, and perhaps even with C++, while the name may seem ambiguous, its usage at least is clear enough and so we make due with its name.

And so, as a general rule of thumb we should really never derive exceptions from `java.lang.RuntimeException` nor throw plain runtime exceptions (e.g. `throw new RuntimeException()`). Put more succinctly, the writers at Sun have the following rules of thumb to follow when it comes to the Checked vs. Unchecked Exception controversy:

- *A method can detect and throw a `RuntimeException` when it [has] encountered an error in the virtual machine runtime. However, it's typically easier to just let the virtual machine detect and throw it. Normally, the methods you write should throw `Exceptions`, not `RuntimeException`.*
- *Similarly, you create a subclass of `RuntimeException` when you are creating an error in the virtual machine runtime (which you probably aren't). Otherwise you should subclass `Exception`.*
- *Do not throw a runtime exception or create a subclass of*

RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

2.3 Why Are Checked (or Unchecked) Exceptions a Hack?

Actually the question should just be, “Why are unchecked exceptions a hack?” Checked exceptions were, as I understand it, and as I've explained it, an experiment. All modern OO languages that have exception handling with the exception of Java have unchecked exceptions and leave it to the programmer to handle them if necessary.

Often if you want to catch the base exception (more on why this is a good idea below) rather than every single exception that can be thrown by the calls to the methods within a method (which makes sense if all you want to do is log the exception information and carry on) then you should be able to do something like the following:

```
public void testOutputStream() {
    try {
        String s = "This is a test";
        OutputStream out = new FileOutputStream("foo/foo.txt");
        out.write(s.getBytes());
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

What this simple method illustrates is that I should be able to catch any exception (but what I really mean and “fail to realize” is that I want to just catch checked exceptions) derived from `java.lang.Exception` (Java is, after all, an OO language), and print out a stack trace of the exception so that I can examine it should something in my logic be incorrect. In fact I will get an exception, a `FileNotFoundException`, because I don't have a directory called `foo` in my working directory and so `FileOutputStream` could not create my file. I don't need to actually capture the `FileNotFoundException` just as I don't need to capture `java.io.IOException`. I simply have to capture `java.lang.Exception` and examine it to know what exception is thrown.

The distinction here is that *I* am examining it. If I wanted to write code that examined the exception and attempted in some way to recover from the error, then I could have explicitly caught the `FileNotFoundException` and, perhaps, taken another path of execution, or warned the user in some way by presenting a dialog stating something like “...the default path where the `foo.txt` file is created is invalid, please choose or create another path.” Hence exception handling.

Now for why the entire checked/unchecked exception mechanism is hacky. Examining a slightly different snippet of code we begin, I hope, to see the problem.

```
public void testOutputStream() {
    try {
        String s = "This is a test";
        OutputStream out = new FileOutputStream("foo.txt");
        out.write(s.getBytes());

        // cause an unchecked exception to be thrown
        //
        Object o = null;
        s = o.toString();
    }
    catch (Exception e) {
        // swallow exception.
    }
}
```

In the new method we fix the output stream so that we no longer throw a `FileNotFoundException`, but now we “accidentally” declare an `Object` called `o`, forget to create it and then try to access its `toString` method. This, of course, will throw an unchecked, `NullPointerException`. But because all unchecked exceptions derive from `java.lang.RuntimeException`, and because `java.lang.RuntimeException` is a `java.lang.Exception`, all unchecked exceptions will be caught by our catch block. Also note that in the updated catch block I've decided to swallow the exception in a black hole (see below), and so now not only do I lose any knowledge of my checked exceptions, but I've also inadvertently swallowed up any unchecked exceptions, too. A simple hack to get around Sun's hack follows.

```
public void testOutputStream() {
    try {
        String s = "This is a test";
        OutputStream out = new FileOutputStream("foo.txt");
        out.write(s.getBytes());

        // cause an unchecked exception to be thrown
        Object o = null;
        s = o.toString();
    }
    catch (RuntimeException exc) {
        throw exc;
    }
    catch (Exception e) {
        // swallow exception.
    }
}
```

Ugly, but workable. This will first catch all unchecked exceptions and re-throw them

back up the call stack and allow checked exceptions to pass to the catch below it. Of course we're still swallowing checked exceptions in a black hole, which is a bad thing, but read on for how any why to avoid doing this.

3 Exception Handling Techniques

Although the idea behind exceptions (e.g. throwing and catching) seems simple, often I find that people just don't quite understand what, really, one ought to do with them.

3.1 Do Not Create Black Holes

You create a black hole (and yes you have) when you catch an exception and just drop it; which is to say, not handle it. Or handle it in the most fundamental way such as simply print it out or log it and then do nothing with it. Sometimes, and then only rarely, is just simply logging or printing of the exception information adequate, and then only during the most trivial applications such as when you're writing test or exploratory code, and even then you should question your intents.

Here is an example of poorly written code that simply drops an exception, or creates a black hole, and that can cause a developer endless frustration (funny for the library writer, bad for the library user).

```
package org.tclancy.apps.blackhole;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 *
 * @author tclancy
 */
public class BlackHole {

    public BlackHole() {
    }

    public void writeString(File f, String s) {
        assert(f != null);
        assert(s != null);
        OutputStream o = null;
        if (s.length() > 0) {
            try {
                o = new FileOutputStream(f);
                o.write(s.getBytes());
            }
            catch(FileNotFoundException exc) {
                // A big black hole!!!
            }
            catch(IOException exc) {
                // Another big black hole!
            }
        }
    }
}
```

```

        finally {
            if (o != null) {
                try {
                    o.close();
                }
                catch(IOException exc) {
                    // not necessarily a black hole.
                }
            }
        }
    }
}

```

In my BlackHole class I have a method called writeString that will write a string to a file if it can, which is to say that if it can create and open the file passed to it, the contents of the string also passed to it will be written to the file. Note, however, that I conveniently catch and drop both the java.io.FileNotFoundException and java.io.IOException, thus creating two black holes; if anything were to go wrong, I never report it to the caller. Case in point:

```

package org.tclancy.apps.blackhole;

import java.io.File;

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        BlackHole app = new BlackHole();

        File f = null;

        f = new File("test.out");
        app.writeString(f, "This is a string to append.");

        f = new File("foo/bar/baz/test.out");
        app.writeString(f, "This is a string to append.");

    }

}

```

A simple test illustrates that two calls to the same method with two separate files will both work without incident, meaning that the method won't complain. But while the first call to the writeString method actually creates a file called test.out and writes the contents of the string to it (simply because I had access rights on my local machine and because I didn't include an absolute path to the file), the second call will fail internally but won't let me know. It will return and I will be none the wiser, not realizing that I don't have a relative path, "foo/bar/baz/", from where I am building and running my test program.

In cases like this, when writing a class that uses classes with methods that throw checked exceptions, if you don't know how to recover or in some way handle the exceptions that the methods throw (e.g. `java.io.FileNotFoundException`), declare that you throw this in your method's signature using the `throws` clause and let the caller of your method handle it. It is the caller, after all, who passed you the bad `File` object.

```
public void writeString(File f, String s)
    throws FileNotFoundException, IOException {...}
```

Alternatively if the method of your class is performing a more complex, business level action, part of which is to write some information to a file or, perhaps, to a record in a database table, you might wish to throw your own package or class level exception, wrapping the specific exception within it. Your new method signature might then look like the following:

```
public void writeString(File f, String s)
    throws BlackHoleWriteStringException {...}
```

These sorts of exceptions are covered in more detail below.

3.2 Do Not Catch for the Sake of Catching

All too often I see in a method a very large catch net that is there for the sake of simply catching all checked exceptions that are thrown by calls to methods surrounded by the `try` block and usually for no other purpose than to log each one separately as if an exception has no way of telling you what kind of exception it is.

For example, in J2EE applications I tend to see methods that make calls to remote EJBs, similar to the following lines of code.

```
public void testEjb() {
    try {
        TestEJBHome testHome;
        TestEJBRemote testRemote;
        ServiceLocator
            getInstance().getHome("TestEJB", TestEJBHome.class);
        testRemote = testHome.create();

        // do some useful work with remote object.
        //
    }
    catch(ServiceLocatorException ex) {
        log.error("ServiceLocatorException found.", e);
    }
}
```

```

        catch(CreateException ex) {
            log.error("CreateException found.", e);
        }
        catch(RemoteException ex) {
            log.error("RemoteException found.", e);
        }
        catch(Exception ex) {
            log.error("Exception found.", e);
        }
    }
}

```

Or I've seen stuff just as horrible. I suppose this isn't as bad as just swallowing the exception in a black hole. The message is being logged, of course. But you have to wonder why it is that the same work is being done in each case when it could easily have been done once as it is being done in the very last case. The last catch block is doing the very same work. It is stating that an exception is being caught. When it passes the exception object to the logging system, the system is asking the exception object to state information about itself, probably by asking it to print its stack trace. The object's stack trace tells us what kind of exception object it is. Placing the string "RemoteException found" or "ServiceLocatorException found", for example, is superfluous information. It just adds waste to the java code and muddles it, making it more burdensome to read. Doing the following makes it simpler:

```

public void testEjb() {
    try {
        TestEJBHome testHome;
        TestEJBRemote testRemote;
        ServiceLocator
            getInstance().getHome("TestEJB", TestEJBHome.class);
        testRemote = testHome.create();

        // do some useful work with remote object.
        //
    }
    catch(Exception ex) {
        log.error("Exception found.", e);
    }
}

```

Of course this still hides the fact from the caller that a problem occurred. It also captures any unchecked exceptions as well. You could add the hack described above that captures `java.lang.RuntimeException` and just rethrows it. You could also log it and then rethrow it.

As an alternative approach, and probably the proper way to handle it, you could wrap the specific checked exceptions that you want to catch in application (package or class level) exceptions and throw those to the caller of your method.

```
public void testEjb() throws TestEjbException {
    try {
        TestEJBHome testHome;
        TestEJBRemote testRemote;
        ServiceLocator
            getInstance().getHome("TestEJB", TestEJBHome.class);
        testRemote = testHome.create();
        testRemote.doTestOne();
    }
    catch(ServiceLocatorException ex) {
        throw new TestEjbException("Could not locate test service", ex);
    }
    catch(CreateException ex) {
        throw new TestEjbException(
            "Could not create remote test object", ex);
    }
    catch(RemoteException ex) {
        throw new TestEjbException(
            "Could not perform remote operation", ex);
    }
}
```

4 Designing Exceptions

When designing a package of classes, or, really, a class library, you should also carefully design exceptions that the public and protected methods of your classes will throw, if indeed they will throw any, should some exceptional conditions occur. It's deciding what these conditions are that can be tricky. Certain “error” conditions such as reaching the end of file (eof), for example, aren't exceptions and thus aren't treated as exceptions, thus simple checks like looking for -1 as the return code of the **read** method of a stream rather than throwing an exception are standard.