**Java Exception Handling**

Java exception handling enables your Java applications to handle errors sensibly. Exception handling is a very important yet often neglected aspect of writing robust Java applications or components. When an error occurs in a Java program it usually results in an exception being thrown. How you throw, catch and handle these exception matters. There are several different ways to do so. Not all are equally efficient and fail safe.

This trail digs deeper into exception handling in Java. The trail covers various do's and dont's of Java exception handling. It also covers a few techniques for efficient and less error prone exception handling. Hopefully you can benefit from some of these texts.

The versions of Java used in this tutorial are Java 6 and Java 7, though most of the techniques here work already from Java 5 and forward. Some even from Java 4.

Below follows a quick introduction to what you can learn in this Java exception handling trail.

## Basic Java Exception Handling

The first 2 texts on Java exception handling covers the basics of Java's exception throwing and catching mechanism, and exception hierarchies. These texts are:

* [Basic Try Catch Finally](http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html)
* [Exception Hierarchies](http://tutorials.jenkov.com/java-exception-handling/exception-hierarchies.html)

## Checked vs. Unchecked Exceptions in Java

Java is one of the few languages to support both checked and unchecked exceptions. In the text [Checked or Unchecked Exceptions?](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html) I discuss the difference between these two types of exceptions. I also end up recommending unchecked exceptions over checked exceptions.

## General Java Exception Handling Advice

The middle 6 texts in this Java exception handling trail covers issues like "fail safe exception" handling, "where to log exceptions", advice on throwing exceptions during validation etc. In other words, various advice on what to remember when designing your application exception throwing and handling.

## Advanced Exception Handling in Java

The two texts [Exception Handling Templates](http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html) and [Exception Enrichment](http://tutorials.jenkov.com/java-exception-handling/exception-enrichment.html) discusses two somewhat more advanced exception handling techniques that can clean up your code a bit. Exception templates moves all the "try-catch" code to one or more reusable exception handling templates. Exception enrichment rids you of having tons of different exceptions in your exception hierarchy. It also helps you avoid long exception stack traces, and get truly unique error codes for each error in your application.

## Most Popular on Java Exception Handling

The most popular texts in this Java exception handling tutorial are the texts on [Exception Handling Templates](http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html) and[Exception Enrichment](http://tutorials.jenkov.com/java-exception-handling/exception-enrichment.html).

## Exception Handling Strategies

I have written a longer tutorial on how to put the techniques described in this tutorial into a coherent exception handling strategy. It is available in its own trail at [Exception Handling Strategies](http://tutorials.jenkov.com/exception-handling-strategies/index.html)

Table of Contents for Java Exception Handling Tutorial

Below is a list of the texts this trail **Java Exception Handling** contains. The list is repeated at the top right of every page in the trail.

|  |  |  |
| --- | --- | --- |
| **Java Exception Handling** | | |
|  |  |  |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Introduction to Java Exception Handling](http://tutorials.jenkov.com/java-exception-handling/index.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Basic Try Catch Finally](http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Try-with-resources](http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Catching Multiple Exceptions](http://tutorials.jenkov.com/java-exception-handling/catching-multiple-exceptions.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Exception Hierarchies](http://tutorials.jenkov.com/java-exception-handling/exception-hierarchies.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Checked or Unchecked Exceptions?](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Exception Wrapping](http://tutorials.jenkov.com/java-exception-handling/exception-wrapping.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Fail Safe Exception Handling](http://tutorials.jenkov.com/java-exception-handling/fail-safe-exception-handling.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Pluggable Exception Handlers](http://tutorials.jenkov.com/java-exception-handling/pluggable-exception-handlers.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Logging: Where to Log Exceptions?](http://tutorials.jenkov.com/java-exception-handling/logging-where-to-log-exceptions.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Validation: Throw Exceptions Early](http://tutorials.jenkov.com/java-exception-handling/validation-throw-exceptions-early.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Validation: Throw Exception or Return False?](http://tutorials.jenkov.com/java-exception-handling/validation-throw-exception-or-return-false.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Exception Handling Templates](http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html) |
|  | http://tutorials.jenkov.com/images/arrow-right-light-medium.png | [Exception Enrichment](http://tutorials.jenkov.com/java-exception-handling/exception-enrichment.html) |

# Basic try-catch-finally Exception Handling in Java

This text summarizes the basics of how try-catch-finally clause error handling works. The examples are in Java, but the rules are the same for C#. The only difference between Java and C# exceptions is that C# doesn't have checked exceptions. Checked and unchecked exceptions are explained in more detail in a different text.

Exceptions are used in a program to signal that some error or exceptional situation has occurred, and that it doesn't make sense to continue the program flow until the exception has been handled. A method may throw an exception for many reasons, for instance if the input parameters are invalid (negative when expecting positive etc.).

## The Call Stack Explained

This text refers to the concept the "call stack" in several places. By the call stack is meant the sequence of method calls from the current method and back to the Main method of the program. If a method A calls B, and B calls C then the call stack looks like this:

A

B

C

When method C returns the call stack only contains A and B. If B then calls the method D, then the call stack looks like this:

A

B

D

Understanding the call stack is important when learning the concept of exception propagation. Exception are propagated up the call stack, from the method that initially throws it, until a method in the call stack catches it. More on that later.

## Throwing Exceptions

If a method needs to be able to throw an exception, it has to declare the exception(s) thrown in the method signature, and then include a throw-statement in the method. Here is an example:

public void divide(int numberToDivide, int numberToDivideBy)

throws BadNumberException{

if(numberToDivideBy == 0){

throw new BadNumberException("Cannot divide by 0");

}

return numberToDivide / numberToDivideBy;

}

When an exception is thrown the method stops execution right after the "throw" statement. Any statements following the "throw" statement are not executed. In the example above the "return numberToDivide / numberToDivideBy;" statement is not executed if a BadNumberException is thrown. The program resumes execution when the exception is caught somewhere by a "catch" block. Catching exceptions is explained later.

You can throw any type of exception from your code, as long as your method signature declares it. You can also make up your own exceptions. Exceptions are regular Java classes that extends java.lang.Exception, or any of the other built-in exception classes. If a method declares that it throws an exception A, then it is also legal to throw subclasses of A.

## Catching Exceptions

If a method calls another method that throws checked exceptions, the calling method is forced to either pass the exception on, or catch it. Catching the exception is done using a try-catch block. Here is an example:

public void callDivide(){

try {

int result = divide(2,1);

System.out.println(result);

} catch (BadNumberException e) {

//do something clever with the exception

System.out.println(e.getMessage());

}

System.out.println("Division attempt done");

}

The BadNumberException parameter e inside the catch-clause points to the exception thrown from the divide method, if an exception is thrown.

If no exeception is thrown by any of the methods called or statements executed inside the try-block, the catch-block is simply ignored. It will not be executed.

If an exception is thrown inside the try-block, for instance from the divide method, the program flow of the calling method, callDivide, is interrupted just like the program flow inside divide. The program flow resumes at a catch-block in the call stack that can catch the thrown exception. In the example above the "System.out.println(result);" statement will not get executed if an exception is thrown fromt the divide method. Instead program execution will resume inside the "catch (BadNumberException e) { }" block.

If an exception is thrown inside the catch-block and that exception is not caught, the catch-block is interrupted just like the try-block would have been.

When the catch block is finished the program continues with any statements following the catch block. In the example above the "System.out.println("Division attempt done");" statement will always get executed.

## Propagating Exceptions

You don't have to catch exceptions thrown from other methods. If you cannot do anything about the exception where the method throwing it is called, you can just let the method propagate the exception up the call stack to the method that called this method. If you do so the method calling the method that throws the exception must also declare to throw the exception. Here is how the callDivide() method would look in that case.

public void callDivide() throws BadNumberException{

int result = divide(2,1);

System.out.println(result);

}

Notice how the try-catch block is gone, and the callDivide method now declares that it can throw a BadNumberException. The program execution is still interrupted if an exception is thrown from the divide method. Thus the "System.out.println(result);" method will not get executed if an exception is thrown from the divide method. But now the program execution is not resumed inside the callDivide method. The exception is propagated to the method that calls callDivide. Program execution doesn't resume until a catch-block somewhere in the call stack catches the exception. All methods in the call stack between the method throwing the exception and the method catching it have their execution stopped at the point in the code where the exception is thrown or propagated.

## Example: Catching IOException's

If an exception is thrown during a sequence of statements inside a try-catch block, the sequence of statements is interrupted and the flow of control will skip directly to the catch-block. This code can be interrupted by exceptions in several places:

public void openFile(){

try {

// constructor may throw FileNotFoundException

FileReader reader = new FileReader("someFile");

int i=0;

while(i != -1){

//reader.read() may throw IOException

i = reader.read();

System.out.println((char) i );

}

reader.close();

System.out.println("--- File End ---");

} catch (FileNotFoundException e) {

//do something clever with the exception

} catch (IOException e) {

//do something clever with the exception

}

}

If the reader.read() method call throws an IOException, the following System.out.println((char) i ); is not executed. Neither is the last reader.close() or the System.out.println("--- File End ---"); statements. Instead the program skips directly to the catch(IOException e){ ... } catch clause. If the new FileReader("someFile"); constructor call throws an exception, none of the code inside the try-block is executed.

## Example: Propagating IOException's

This code is a version of the previous method that throws the exceptions instead of catching them:

public void openFile() throws IOException {

FileReader reader = new FileReader("someFile");

int i=0;

while(i != -1){

i = reader.read();

System.out.println((char) i );

}

reader.close();

System.out.println("--- File End ---");

}

If an exception is thrown from the reader.read() method then program execution is halted, and the exception is passed up the call stack to the method that called openFile(). If the calling method has a try-catch block, the exception will be caught there. If the calling method also just throws the method on, the calling method is also interrupted at the openFile() method call, and the exception passed on up the call stack. The exception is propagated up the call stack like this until some method catches the exception, or the Java Virtual Machine does.

## Finally

You can attach a finally-clause to a try-catch block. The code inside the finally clause will always be executed, even if an exception is thrown from within the try or catch block. If your code has a return statement inside the try or catch block, the code inside the finally-block will get executed before returning from the method. Here is how a finally clause looks:

public void openFile(){

FileReader reader = null;

try {

reader = new FileReader("someFile");

int i=0;

while(i != -1){

i = reader.read();

System.out.println((char) i );

}

} catch (IOException e) {

//do something clever with the exception

} finally {

if(reader != null){

try {

reader.close();

} catch (IOException e) {

//do something clever with the exception

}

}

System.out.println("--- File End ---");

}

}

No matter whether an exception is thrown or not inside the try or catch block the code inside the finally-block is executed. The example above shows how the file reader is always closed, regardless of the program flow inside the try or catch block.

Note: If an exception is thrown inside a finally block, and it is not caught, then that finally block is interrupted just like the try-block and catch-block is. That is why the previous example had the reader.close() method call in the finally block wrapped in a try-catch block:

} finally {

if(reader != null){

try {

reader.close();

} catch (IOException e) {

//do something clever with the exception

}

}

System.out.println("--- File End ---");

}

That way the System.out.println("--- File End ---"); method call will always be executed. If no unchecked exceptions are thrown that is. More about checked and unchecked in a later chapter.

You don't need both a catch and a finally block. You can have one of them or both of them with a try-block, but not none of them. This code doesn't catch the exception but lets it propagate up the call stack. Due to the finally block the code still closes the filer reader even if an exception is thrown.

public void openFile() throws IOException {

FileReader reader = null;

try {

reader = new FileReader("someFile");

int i=0;

while(i != -1){

i = reader.read();

System.out.println((char) i );

}

} finally {

if(reader != null){

try {

reader.close();

} catch (IOException e) {

//do something clever with the exception

}

}

System.out.println("--- File End ---");

}

}

Notice how the catch block is gone.

## Catch or Propagate Exceptions?

You might be wondering whether you should catch or propate exceptions thrown in your program. It depends on the situation. In many applications you can't really do much about the exception but tell the user that the requested action failed. In these applications you can usually catch all or most exceptions centrally in one of the first methods in the call stack. You may still have to deal with the exception while propagating it though (using finally clauses). For instance, if an error occurs in the database connection in a web application, you may still have to close the database connection in a finally clause, even if you can't do anything else than tell the user that the action failed. How you end up handling exceptions also depends on whether you choose checked or unchecked exceptions for your application. There is more on that in other texts in the error handling trail.

# Try-with-resources in Java 7

Try-with-resources in Java 7 is a new exception handling mechanism that makes it easier to correctly close resources that are used within a try-catch block.

Here is a list of topics covered in this text:

## Resource Management With Try-Catch-Finally, Old School Style

Managing resources that need to be explicitly closed is somewhat tedious before Java 7.

Look at the following method which reads a file and prints it to the System.out:

private static void printFile() throws IOException {

InputStream input = null;

try {

input = **new FileInputStream("file.txt");**

int data = **input.read();**

while(data != -1){

System.out.print((char) data);

data = **input.read();**

}

} finally {

if(input != null){

**input.close();**

}

}

}

The code marked in bold is where the code can throw an Exception. As you can see, that can happen in 3 places inside the try-block, and 1 place inside the finally-block.

The finally block is always executed no matter if an exception is thrown from the try block or not. That means, that the InputStream is closed no matter what happens in the try block. Or, attempted closed that is. TheInputStream's close() method may throw an exception too, if closing it fails.

Imagine that an exception is thrown from inside the try block. Then the finally block is executed. Imagine then, that an exception is also thrown from the finally block. Which exception do you think is propagated up the call stack?

The exception thrown from the finally block would be propagated up the call stack, even if the exception thrown from the try block would probably be more relevant to propagate.

## Try-with-resources

In Java 7 you can write the code from the example above using the try-with-resource construct like this:

private static void printFileJava7() throws IOException {

try(FileInputStream input = new FileInputStream("file.txt")) {

int data = input.read();

while(data != -1){

System.out.print((char) data);

data = input.read();

}

}

}

Notice the first line inside the method:

try(FileInputStream input = new FileInputStream("file.txt")) {

This is the try-with-resources construct. The FileInputStream variable is declared inside the parentheses after the try keyword. Additionally, a FileInputStream is instantiated and assigned to the variable.

When the try block finishes the FileInputStream will be closed automatically. This is possible becauseFileInputStream implements the Java interface java.lang.AutoCloseable. All classes implementing this interface can be used inside the try-with-resources construct.

If an exception is thrown both from inside the try-with-resources block, and when the FileInputStream is closed (when close() is called), the exception thrown inside the try block is thrown to the outside world. The exception thrown when the FileInputStream was closed is suppressed. This is opposite of what happens in the example first in this text, using the old style exception handling (closing the resources in the finally block).

## Using Multiple Resources

You can use multiple resources inside a try-with-resources block and have them all automatically closed. Here is an example:

private static void printFileJava7() throws IOException {

try( FileInputStream input = new FileInputStream("file.txt");

BufferedInputStream bufferedInput = new BufferedInputStream(input)

) {

int data = bufferedInput.read();

while(data != -1){

System.out.print((char) data);

data = bufferedInput.read();

}

}

}

This example creates two resources inside the parentheses after the try keyword. An FileInputStream and aBufferedInputStream. Both of these resources will be closed automatically when execution leaves the try block.

The resources will be closed in reverse order of the order in which they are created / listed inside the parentheses. First the BufferedInputStream will be closed, then the FileInputStream.

## Custom AutoClosable Implementations

The try-with-resources construct does not just work with Java's built-in classes. You can also implement thejava.lang.AutoCloseable interface in your own classes, and use them with the try-with-resourcesconstruct.

The AutoClosable interface only has a single method called close(). Here is how the interface looks:

public interface AutoClosable {

public void close() throws Exception;

}

Any class that implements this interface can be used with the try-with-resources construct. Here is a simple example implementation:

public class MyAutoClosable implements AutoCloseable {

public void doIt() {

System.out.println("MyAutoClosable doing it!");

}

@Override

public void close() throws Exception {

System.out.println("MyAutoClosable closed!");

}

}

The doIt() method is not part of the AutoClosable interface. It is there because we want to be able to do something more than just closing the object.

Here is an example of how the MyAutoClosable is used with the try-with-resources construct:

private static void myAutoClosable() throws Exception {

try(MyAutoClosable myAutoClosable = new MyAutoClosable()){

myAutoClosable.doIt();

}

}

Here is the output printed to System.out when the method myAutoClosable() is called:

MyAutoClosable doing it!

MyAutoClosable closed!

As you can see, try-with-resources is a quite powerful way of making sure that resources used inside a try-catch block are closed correctly, no matter if these resources are your own creation, or Java's built-in components.

# Catching Multiple Exceptions in Java 7

In Java 7 it was made possible to catch multiple different exceptions in the same catch block. This is also known as multi catch.

Before Java 7 you would write something like this:

try {

// execute code that may throw 1 of the 3 exceptions below.

} catch(SQLException e) {

logger.log(e);

} catch(IOException e) {

logger.log(e);

} catch(Exception e) {

logger.severe(e);

}

As you can see, the two exceptions SQLException and IOException are handled in the same way, but you still have to write two individual catch blocks for them.

In Java 7 you can catch multiple exceptions using the multi catch syntax:

try {

// execute code that may throw 1 of the 3 exceptions below.

} catch(**SQLException | IOException e**) {

logger.log(e);

} catch(Exception e) {

logger.severe(e);

}

Notice how the two exception class names in the first catch block are separated by the pipe character |. The pipe character between exception class names is how you declare multiple exceptions to be caught by the same catchclause.

# Exception Hierarchies

In Java and in C# exceptions can be categorized into hierarchies. The hierarchy is created by having one (or more) exception extend another exception. The first exception becomes a subclass of the second. In Java FileNotFoundException is a subclass of IOException. Here is how a custom exception looks in Java code:

public class MyException extends Exception{

//constructors etc.

}

As you can see there isn't much to it. The advantage of exception hierarchies is that if you decide to catch (using try-catch) a certain exception in the hierarchy, then you will automatically also catch all subclasses of that exception too. In other words, you will catch all exceptions from that certain exception and down the hierarchy. In the example with FileNotFoundException, if you catch IOException which is the superclass of FileNotFoundException, you will also catch FileNotFoundException.

## Multiple Catch Blocks

As you may know already it is possible to have several catch blocks for the same try-block. This is usually the case if the code inside the try-block throws more than one type of exception. But, multiple catch blocks can also be used in the case where all the exceptions thrown inside the try-block are the same type or subclasses of that type. This code illustrates that:

try{

//call some methods that throw IOException's

} catch (FileNotFoundException e){

} catcy (IOException e){

}

Remember that the first catch-block a thrown exception matches will handle that exception. In this example all IOExceptions are being handled by the catch(IOException e) except for FileNotFoundException. The fact that FileNotFoundException is a subclass of IOException gives us the choice of either treating all IOExceptions the same, or catch some of IOExceptions subclasses individually, as is done in the code example above. If the catch(FileNotFoundException e) block is removed any FileNotFoundException will be caught by the catch(IOException e) block, since FileNotFoundException is a subclass of IOException.

## Throws Clauses

If a method can throw either a certain exception A, or any subclasses of A (Asub), then it is enough to declare in the method declaration that the method throws A. It is then allowed to throw subclasses of A from the method too. Here is an example:

public void doSomething() throws IOException{

}

You are allowed to declare the subclasses in the throws clause of the method, even if you don't really need to. It can make the code easier to read and understand for the next developer to look at it. Here is an example:

public void doSomething() throws IOException, FileNotFoundException{

}

As long as the superclass of any declared exception is also declared thrown, it doesn't have any effect on the code to include the throwing of the subclass. In the example above it has no real effect that FileNotFoundException is declared thrown when IOException is also declared. When you catch IOException you also catch FileNotFoundException. It is still possible to handle the two exceptions with each their own catch-block as shown earlier, even if only the superclass is declared thrown.

## Designing Exception Hierarchies

When designing exception hieararchies for an API or application it is a good idea to create a base exception for that API or application. For instance, in Mr. Persister, our Java Persistence / ORM API the base exception is called PersistenceException. This base exception makes it possible to catch and handle all exceptions thrown by Mr. Persister in the same catch block.

If you need more granularity on the exceptions thrown, for instance because you think the exceptions may be handled differently, then add new exceptions as subclasses of your API or application base exception. That way users have the choice of either catching the specific exceptions, or just the base exception. In Mr. Persister we could add a ConnectionOpenException, QueryException, UpdateException, CommitException, and ConnectionCloseException as subclasses of PersistenceException. If the users of Mr. Persister wanted to handle a ConnectionOpenException differently than a QueryException or a ConnectionCloseException, they could catch those exceptions and handle them differently. If not, the user could just catch PersistenceException and handle all exceptions uniformly.

You can subdivide the exceptions more by add more levels to the hierarchy. For instance, you may want to distinguish between an exception thrown because the URL to the database is wrong, and an exception thrown because the database is not running. In that case you could create two exceptions called DatabaseURLException and DatabaseNotRespondingException, which both extends the ConnectionOpenException. Then users can catch the two different exceptions and respond differently to them.

## Summary

In this text we have seen that exception hierarchies can be created by subclassing exception classes. It is a good idea to create a base exception for your API or application, and have all other exceptions subclass this base exception. Individual subclasses makes it possible (but not obligatory) to catch and handle these individual exceptions differently. You should create individual exceptions only for errors that can actually be handled differently.

# Checked or Unchecked Exceptions?

By Jakob Jenkov

In Java there are basically two types of exceptions: Checked exceptions and unchecked exceptions. C# only has unchecked exceptions. The differences between checked and unchecked exceptions are:

1. Checked exceptions must be explicitly caught or propagated as described in [Basic try-catch-finally Exception Handling](http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html). Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.
2. Checked exceptions in Java extend the java.lang.Exception class. Unchecked exceptions extend the java.lang.RuntimeException.

There are many arguments for and against both checked and unchecked, and whether to use checked exceptions at all. I will go through the most common arguments throughout this text. Before I do so, let me just make one thing clear:

Checked and unchecked exceptions are functionally equivalent. There is nothing you can do with checked exceptions that cannot also be done with unchecked exceptions, and vice versa.

Regardless of your choice between checked and unchecked exceptions it is a matter of personal or organisational style. None is functionally better than the other.

## A Simple Example

Before discussing the advantages and disadvantages of checked and unchecked exceptions I will show you the difference in the code they make. Here is a method that throws a checked exception, and another method that calls it:

public void storeDataFromUrl(String url){

try {

String data = readDataFromUrl(url);

} catch (BadUrlException e) {

e.printStackTrace();

}

}

public String readDataFromUrl(String url)

throws BadUrlException{

if(isUrlBad(url)){

throw new BadUrlException("Bad URL: " + url);

}

String data = null;

//read lots of data over HTTP and return

//it as a String instance.

return data;

}

As you can see the readDataFromUrl() method throws a BadUrlException. I have created BadUrlException myself. BadUrlException is a checked exception because it extends java.lang.Exception:

public class BadUrlException extends Exception {

public BadUrlException(String s) {

super(s);

}

}

If storeDataFromUrl() wants to call readDataFromUrl() it has only two choices. Either it catches the BadUrlException or propagates it up the call stack. The storeDataFromUrl() listed above catches the exception. This storeDataFromUrl() implementation propagates the BadUrlException instead:

public void storeDataFromUrl(String url)

throws BadUrlException{

String data = readDataFromUrl(url);

}

Notice how the try catch block is gone and a "throws BadUrlException" declaration is added instead. Now, let's see how it looks with unchecked exceptions. First I change the BadUrlException to extend java.lang.RuntimeException instead:

public class BadUrlException extends RuntimeException {

public BadUrlException(String s) {

super(s);

}

}

Then I change the methods to use the now unchecked BadUrlException:

public void storeDataFromUrl(String url){

String data = readDataFromUrl(url);

}

public String readDataFromUrl(String url) {

if(isUrlBad(url)){

throw new BadUrlException("Bad URL: " + url);

}

String data = null;

//read lots of data over HTTP and

//return it as a String instance.

return data;

}

Notice how the readDataFromUrl() method no longer declares that it throws BadUrlException. The storeDataFromUrl() method doesn't have to catch the BadUrlException either. The storeDataFromUrl() method can still choose to catch the exception but it no longer has to, and it no longer has to declare that it propagates the exception.

## Checked or Unchecked?

Now that we have seen the difference in code between checked and unchecked exceptions, let's dive into the arguments for and against both.

Some Java books(\*) covering exceptions advice you to use checked exceptions for all errors the application can recover from, and unchecked exceptions for the errors the application cannot recover from. In reality most applications will have to recover from pretty much all exceptions including NullPointerException, IllegalArgumentExceptions and many other unchecked exceptions. The action / transaction that failed will be aborted but the application has to stay alive and be ready to serve the next action / transaction. The only time it is normally legal to shut down an application is during startup. For instance, if a configuration file is missing and the application cannot do anything sensible without it, then it is legal to shut down the application.

(\*) Suns Java Tutorial does for one.

My advice to you is to use either only checked exceptions or only unchecked exceptions. Mixing exception types often results in confusion and inconsistent use. Of course you should be pragmatic. Do what makes sense in your situation.

Below is a list of the most common arguments for and against checked and unchecked exceptions. An argument in favor of one type of exceptions is usually against the other type (pro-checked = con-unchecked, pro-unchecked = con-checked). Therefore the arguments are only listed as either in favour of checked or unchecked exceptions.

1. Pro Checked Exceptions:   
   Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception.
2. Pro Checked Exceptions:   
   Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1).
3. Pro Unchecked Exceptions:   
   Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call.
4. Pro Checked Exceptions:  
   When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them.
5. Pro Unchecked Exceptions:  
   Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface.

Each of the arguments also have counter arguments which will be discussed as I go through the argument in the following sections.

### Argument 1 (Pro Checked Exceptions):

Compiler enforced catching or propagation of checked exceptions makes it harder to forget the handling of that exception.

### Counter-argument:

When being forced to catch or propagate many exceptions developers risk acting sloppily, and just write

try{

callMethodThatThrowsException();

catch(Exception e){

}

and thus effectively ignore the error. 

### Argument 2 (Pro Checked Exceptions):

Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions.

### Counter-argument 1:

It's not any worse than the sloppy exception handling tendency when being forced to handle or propagate checked exceptions.

### Counter-argument 2:

On a recent larger project we decided to go with unchecked exceptions. My personal experience from that project is this: When using unchecked exceptions any method can potentially throw exceptions. Thus I was always reasonably conscious about exceptions no matter what parts of the code I was working on. Not only when checked exceptions were declared.

In addition many of the standard Java API methods that do not declare any checked exceptions may still throw unchecked exceptions like NullPointerException or InvalidArgumentException. Your application will still need to handle these unchecked exceptions. You could argue that the fact that there are checked exceptions makes it easy to forget handling the unchecked exceptions because they are not declared. 

### Argument 3 (Pro Unchecked Exceptions):

Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call. That is. the declared exceptions are aggreated up the methods in the call stack. Example:

public long readNumberFromUrl(String url)

throws BadUrlExceptions, BadNumberException{

String data = readDataFromUrl(url);

long number = convertData(data);

return number;

}

private String readDataFromUrl(String url)

throws BadUrlException {

//throw BadUrlException if url is bad.

//read data and return it.

}

private long convertData(String data)

throws BadNumberException{

//convert data to long.

//throw BadNumberException if number isn't within valid range.

}

As you can see the readNumberFromUrl() needs to declare throwing both the BadUrlException and the BadNumberException that are thrown from the readDataFromUrl() and converData() methods. Imagine how many exceptions would need to be declared at the top level methods of an application with thousands of classes. This can make checked exception propagation a real pain.

### Counter-argument 1:

The exception declaration aggregation rarely happens in real applications. Often developers will use exception wrapping instead. Here is how that could look:

public void readNumberFromUrl(String url)

throws ApplicationException{

try{

String data = readDataFromUrl(url);

long number = convertData(data);

} catch (BadUrlException e){

throw new ApplicationException(e);

} catch (BadNumberException e){

throw new ApplicationException(e);

}

}

As you can see the readNumberFromUrl() method now only declares throwing ApplicationException. The exceptions BadUrlException and BadNumberException are caught and wrapped in a more general ApplicationException. This way exception wrapping avoids exception declaration aggregation.

My personal opinion is, that if all you do is to wrap the exception and not provide any extra information, why wrap it at all? The try-catch block is just extra code that doesn't do anything. It would be easier to just make the ApplicationException, BadUrlException and BadNumberException be unchecked exceptions. Here is an unchecked version of the above code:

public void readNumberFromUrl(String url){

String data = readDataFromUrl(url);

long number = convertData(data);

}

It is still possible to wrap unchecked exceptions if you should want to. Below is a wrapping edition of the unchecked code. Notice how the readNumberFromUrl() method does not declare throwing the ApplicationException even if it throws it.

public void readNumberFromUrl(String url)

try{

String data = readDataFromUrl(url);

long number = convertData(data);

} catch (BadUrlException e){

throw new ApplicationException(

"Error reading number from URL", e);

} catch (BadNumberException e){

throw new ApplicationException(

"Error reading number from URL", e);

}

}

### Counter-argument 2:

Another commonly used technique to avoid exception declaration aggregation up the call stack of an application is to create an application base exception. All exceptions thrown in the application must be a subclass of the base exception. All methods throwing exceptions need only declare to throw the base exception. As you know a method throwing Exception may also throw any subclass of Exception. Here is how that could look:

public long readNumberFromUrl(String url)

throws ApplicationException {

String data = readDataFromUrl(url);

long number = convertData(data);

return number;

}

private String readDataFromUrl(String url)

throws BadUrlException {

//throw BadUrlException if url is bad.

//read data and return it.

}

private long convertData(String data)

throws BadNumberException{

//convert data to long.

//throw BadNumberException if number isn't within valid range.

}

public class ApplicationException extends Exception{ }

public class BadNumberException extends ApplicationException{}

public class BadUrlException extends ApplicationException{}

Notice how the BadNumberException and BadUrlException are no longer declared thrown nor caught and wrapped. They are subclasses of the ApplicationException so they will get propagated up the call stack.

My opinion is the same as with exception wrapping: If all methods in the application just declares throwing the ApplicationException (base exception), why not just make the ApplicationException unchecked and save some try-catch blocks and throws ApplicationExceptions clauses?

### Argument 4 (Pro Checked Exceptions)

When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them. Without declaration you cannot know which exceptions the method may throw. Thus you may not know how to handle them properly. Except of course, if you have access to the code and can see there what exceptions may be thrown from the method.

### Counter-argument:

In most cases you cannot do anything about the exception except showing an error message to the user, write a message to the log, and/or rollback the transaction etc. No matter what exception occurs you will in many situations handle it the same way. Because of this applications often have a few central and general pieces of error handling code. Therefore it is not so important to know exactly what exceptions may be thrown.

### Argument 5 (Pro Unchecked Exceptions)

Checked exceptions declared on methods become part of a the class or interface contract. This makes it harder to add new exceptions to the method later without breaking the contract.

### Counter-argument

This is not a problem if the method uses a base exception. New exceptions can be thrown at will if the method declares throwing the base exception. The only requirement is that the new exceptions thrown are subclasses of the base exception.

Again, what is the value of having all methods that may throw exceptions declare throwing the same base exception? Does it enable you to handle the exceptions any better than if you knew the methods might throw an unchecked exception?

## Summary

I used to be in favor of checked exceptions but recently I have begun to change my mind. Personalities like Rod Johnson (Spring Framework), Anders Hejlsberg (father of C#), Joshua Bloch (Effective Java, item 41: Avoid unnecessary use of checked exceptions) and others have made me rethink the real benefit of checked exceptions. Lately we have tried using unchecked exceptions on a larger project, and they have worked out just fine. The error handling is centralized in a few classes. Here and there we have had to do local error handling instead of propagating the exception to the main error handling code. But it is not in very many places. Our code has become somewhat more readable now that there aren't try-catch blocks all over the code. In other words, there are a lot less no-benefit catch-rethrow try-catch blocks in the code than with checked exceptions. All in all I would recommend using unchecked exceptions. At least give it a try on a project. I have summarized the reasons below:

* Unchecked exceptions do not clutter the code with unnecessary try-catch blocks.
* Unchecked exceptions do not clutter the method declarations with aggregated exception declarations.
* The argument that you easier forget to handle unchecked exceptions is not valid in my experience.
* The argument that it is harder to know how to handle undeclared exceptions is not valid in my experience.
* Unchecked exceptions avoids versioning problems altogether.

You or your project will have to make your own decisions about whether to use checked or unchecked exceptions, or both. Here is a list of resources that also discusses the decision between checked and unchecked exceptions.

Anders Hejlsberg on checked vs. unchecked exceptions  
<http://www.artima.com/intv/handcuffs.html>   
  
James Gosling on checked exceptions  
<http://www.artima.com/intv/solid.html>   
  
Bill Venners on Exceptions  
<http://www.artima.com/interfacedesign/exceptions.html>   
  
Bruce Eckel on checked exceptions  
<http://www.artima.com/intv/typingP.html>   
  
Designing with Exceptions (Bill Venners - www.artima.com)  
<http://www.artima.com/designtechniques/desexcept.html>   
  
[Effective Java](http://www.amazon.co.uk/exec/obidos/redirect?link_code=ur2&camp=1634&tag=jenkovdevelop-21&creative=6738&path=ASIN/0201310058/qid=1124382514/sr=1-1/ref=sr_1_3_1)http://www.assoc-amazon.co.uk/e/ir?t=jenkovdevelop-21&l=ur2&o=2 (Joshua Bloch - Addison Wesley 2001)   
  
Daniel Pietraru - in favor of checked exceptions  
[Exceptional Java - Checked exceptions are priceless� For everything else there is the RuntimeException](http://littletutorials.com/2008/05/06/exceptional-java-checked-exceptions-are-priceless-for-everything-else-there-is-the-the-runtimeexception/)

# Exception Wrapping

Exception wrapping is wrapping is when you catch an exception, wrap it in a different exception and throw that exception. Here is an example:

try{

dao.readPerson();

} catch (SQLException sqlException) {

throw new MyException("error text", sqlException);

}

The method dao.readPerson() can throw an SQLException. If it does, the SQLException is caught and wrapped in a MyException. Notice how the SQLException (the sqlException variable) is passed to the MyException's constructor as the last parameter.

Exception wrapping is a standard feature in Java since JDK 1.4. Most (if not all) of Java's built-in exceptions has constructors that can take a "cause" parameter. They also have a getCause() method that will return the wrapped exception.

## Why Use Exception Wrapping?

The main reason one would use exception wrapping is to prevent the code further up the call stack from having to know about every possible exception in the system. There are two main reasons for this.

The first reason is, that declared exceptions aggregate towards the top of the call stack. If you do not wrap exceptions, but instead pass them on by declaring your methods to throw them, you may end up with top level methods that declare many different exceptions. Declaring all these exceptions in each method back up the call stack becomes tedious.

The second reason is that you may not want your top level components to know anything about the bottom level components, nor the exceptions they throw. For instance, the purpose of DAO interfaces and implementations is to abstract the details of data access away from the rest of the application. Now, if your DAO methods throw SQLException's then the code using the DAO's will have to catch them. What if you change to an implementation that reads the data from a web service instead of from a database? Then you DAO methods will have to throw both RemoteException and SQLException. And, if you have a DAO that reads data from a file, you will need to throw IOException too. That is three different exceptions, each bound to their own DAO implementation.

To avoid this your DAO interface methods can throw DaoException. In each implementation of the DAO interface (database, file, web service) you will catch the specific exceptions (SQLException, IOException, RemoteException), wrap it in a DaoException, and throw the DaoException. Then code using the DAO interface will only have to deal with DaoException's. It does not need to know anything about what data access technology is used in the various implementations.

# Fail Safe Exception Handling

It is imperative that the exception handling code is fail safe. An important rule to remember is

The last exception thrown in a try-catch-finally block is

the exception that will be propagated up the call stack.

All earlier exceptions will disappear.

If an exception is thrown from inside a catch or finally block, this exception may hide the exception caught by that block. This is misleading when trying to determine the cause of the error.

Below is a classic example of non-fail-safe exception handling:

InputStream input = null;

try{

input = new FileInputStream("myFile.txt");

//do something with the stream

} catch(IOException e){

throw new WrapperException(e);

} finally {

try{

input.close();

} catch(IOException e){

throw new WrapperException(e);

}

}

If the FileInputStream constructor throws a FileNotFoundException, what do you think will happen?

First the catch block is executed. This block just rethrows the exception wrapped in a WrapperException.

Second the finally block will be executed which closes the input stream. However, since a FileNotFoundException was thrown by the FileInputStream constructor, the "input" reference will be null. The result will be a NullPointerException thrown from the finally block. The NullPointerException is not caught by the finally block's catch(IOException e) clause, so it is propagated up the call stack. The WrapperException thrown from the catch block will just disappear!

The correct way to handle this situation is course to check if references assigned inside the try block are null before invoking any methods on them. Here is how that looks:

InputStream input = null;

try{

input = new FileInputStream("myFile.txt");

//do something with the stream

} catch(IOException e){ //first catch block

throw new WrapperException(e);

} finally {

try{

if(input != null) input.close();

} catch(IOException e){ //second catch block

throw new WrapperException(e);

}

}

But even this exception handling has a problem. Let's pretend the file exists, so the "input" reference now points to a valid FileInputStream. Let's also pretend that an exception is thrown while processing the input stream. That exception is caught in the catch(IOException e) block. It is then wrapped and rethrown. Before the wrapped exception is propagated up the call stack, the finally clause is executed. If the input.close() call fails, and an IOException is thrown, then it is caught, wrapped and rethrown. However, when throwing the wrapped exception from the finally clause, the wrapped exception thrown from the first catch block is again forgotten. It disappears. Only the exception thrown from the second catch block is propagated up the call stack.

As you can see, fail safe exception handling is not always trivial. The InputStream processing example is not even the most complex example you can come across. Transactions in JDBC have somewhat more error possibilities. Exceptions can occur when trying to commit, then rollback, and finally when you try to close the connection. All of these possible exceptions should be handled by the exception handling code, so none of them make the first exception thrown disappear. One way to do so is to make sure that the last exception thrown contains all previously thrown exceptions. That way they are all available to the developer investigating the error cause. This is how our Java persistence API, Mr Persister, implements transaction exception handling.

By the way, the [try-with-resources](http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html) features in Java 7 makes it easier to implement fail safe exception handling.

# Pluggable Exception Handlers

Pluggable exception handlers is a technique that allows the users of a component to customize the exception handling of that component. Instead of handling, enriching, wrapping and/or logging the exception, the exception handling is delegated to an exception handler. Here is how an exception handler interface could look in code:

public interface ExceptionHandler {

public void handle(Exception e, String errorMessage);

}

And here is a class that uses it:

public class Component{

protected ExceptionHandler exceptionHandler = null;

public void setExceptionHandler(ExceptionHandler handler){

this.exceptionHandler = handler;

}

public void processFile(String fileName){

FileInputStream input = null;

try{

input = new FileInputStream(fileName);

processStream(input);

} catch (IOException e){

this.exceptionHandler.handle(e,

"error processing file: " + fileName);

}

}

protected void processStream(InputStream input)

throws IOException{

//do something with the stream.

}

}

Notice how the catch clause in the processFile method does not wrap, rethrow or log caught IOException's. The exceptions are passed to the ExceptionHandler instance's handle() method. The exception handler can then decide what to do with it. Should the exception be ignored? Logged? Wrapped in a different exception? Or just rethrown as it is? The exception handler can decide for itself. Note however, that the exception handler cannot throw checked exceptions that were not declared in the method where the exception was caught. For instance, the processFile method does not declare any checked exceptions. Thus, the ExceptionHandler's handle() method cannot throw any checked exceptions. However, the ExceptionHandler can throw all the unchecked exceptions it wants (RuntimeException and subclasses of it).

## Where to use Pluggable Exception Handlers?

Pluggable exception handlers are most effective in situations where the exceptions occurring can be handled sensibly in different ways. For instance, when validating an XML document, or an HTML form, you may not always want to stop the validation at the first validation error. In some situations you might want to continue validation to catch all validation exceptions thrown, and show them all to the user at the same time. This saves the user from having to correct an error, validate, correct error, validate over and over again. All errors can be caught and corrected in one iteration.

## Implementation

There is no standard ExceptionHandler interface in Java. You will have to define one yourself, matching the needs of your component or application. Exactly how many parameters the ExceptionHandler's handle method should take depends on what the component is trying to do. As a minimum though, I would recommend taking the caught exception and a text explaining what tried to do at the time of the error.

It can be a good idea to provide a default implementation of your ExceptionHandler interface, and have your components initialized with an instance of it. This saves the user of the component from having to plugin an exception handler in the situations where the desired handling of the exception is the same as the default implementation provides. For instance, a default implementation could just rethrow the exception, or wrap it in a component specific exception.

## Code Examples

Here are a few examples of exception handlers.

public class IgnoringHandler implements ExceptionHandler{

public void handle(Exception e, String message) {

//do nothing, just ignore the exception

}

}

public class WrappingHandler implements ExceptionHandler{

public void handle(Exception e, String message){

throw new RuntimeException(message, e);

}

}

public class CollectingHandler implements ExceptionHandler{

List exceptions = new ArrayList();

public List getExceptions(){ return this.exceptions; }

public void handle(Exception e, String message){

this.exceptions.add(e);

//message is ignored here, but could have been

//collected too.

}

}

# Logging Exceptions: Where to Log Exceptions?

It is often a requirement that exceptions occurring in business applications must be logged, so they can be further examined by a human if necessary. Especially in web or server applications where the console output may not be available for examination. The exception log can be helpful at determining what went wrong, by either containing enough information to reveal the cause of the error, or aid in reproducing the exception.

When designing the logging of an application the question often arise: Where in the code should the exceptions be logged? Basically you have three different options:

1. Bottom Level Logging  
   Logging in the component where the exception occurs
2. Mid Level Logging  
   Logging somewhere in the middle of the call stack, where sufficient information is available (the context of the component call)
3. Top Level Logging  
   Logging centrally at the top of the call stack

The listing below shows a call stack where component A calls B. B calls other components, which in turn call the component F. Component A is the top level component, B etc. are a mid level components, and F is a bottom level component.

A

B

...

F

## Bottom Level Logging

The first option you have is to log the exception in the component where it occurs. If you cannot alter the source of that component, then you will log as close as possible to the component method call throwing the exception. At first glance it may seem like a good design choice to encapsulate logging inside the component that throws the exception. However, this approach has a few drawbacks.

First of all logging has to be coded into every component capable of throwing an exception or calling a third party component that throws exceptions. This is a lot of logging code to write and maintain.

Second, if the component is to be reused across different applications, the component has no way of knowing how exceptions is to be logged in each application. You will have to let the logging strategy be pluggable in the component, and then the logging isn't really encapsulated anyways.

Third the component that throws the exception may not really have sufficient information available to write a detailed and sensible log message. Say you have a component that writes an object to database. An exception is thrown because one of the fields is null. Can the component tell exactly what field was null in the log? Can it tell why this field was null? Perhaps the object was loaded with that field as null, or some action taken by the user caused it to become null. Or perhaps a bug in some code caused the field to become null. Does the component know what user was logged in when the exception ocurred? All this information might be needed for logging. The bottom level component may not have all that information available. The rest of the needed information may be available further up the call stack. This information cannot be logged if the logging is done inside the component, at bottom level.

## Mid Level Logging

As alternative to logging at the bottom level, you can log at the mid level wherever enough information is available to write a satisfactory log message. This keeps the bottom level components free of logging code, but it has some drawbacks too.

At mid level you may not have all details available from the bottom level component that threw the exception. The bottom level component must then supply a detailed error text and perhaps error code, to make sensible logging possible at the mid level.

When logging at the mid level there is still a lot of logging code to write. At every place you catch and log an exception, you will have to insert almost the same logging code. If you later need to change the logging strategy it will take a lot of work. A shortcut would be to inject the logging code using aspect oriented programming, but I won't get into that here.

## Top Level Logging

Logging at the top level means that you have a single, central place in the code that catches all thrown exceptions and logs them. In a Java web application that could be a control servlet or perhaps a servlet filter. In a desktop application perhaps you your event handlers would extend a base event handler capable of catching and logging all exceptions thrown.

The advantage of top level logging is that you only have a single spot in the application where logging code is written and maintained. In addition to being easier to implement, this also makes it easier to postpone the logging implementation until later in the development process if needed. The logging code will only have to be implemented in one place, and most likely won't change the total application call structure.

The disadvantage of top level logging is of course that this single central place doesn't know anything about what went wrong in the bottom level component that threw the exception, or what the mid level code calling that component was trying to do. That makes writing sensible log messages somewhat harder.

## Exception Enrichment

To overcome the lack of details available at the top level (and also to some degree at the mid level), you can enrich the exceptions as they propagate up the call stack towards the top level. Enriching exceptions is done by catching them, adding extra information and rethrowing them again. Rethrowing a caught Java exception will not change the embedded stack trace. The stack trace remains the same as when the exception was thrown first, by the bottom level component.

You will have to code your own exception class in order to make exception enrichment possible. Either that, or you'll have to wrap the caught exception in a new exception and throw the new exception.

## Summary and Recommendation

We have looked at three different logging strategies: Bottom, mid and top level logging.

I recommend using top level logging whereever possible, since it is easiest to code and maintain, and also to add later in the development process if you do not want to be bothered with it in the beginning. You may run into situations where an exception must be caught and delt with at a lower level, but these situations don't happen often in my experience. Most often, when an exception occurs that request or event handling is just skipped, and the exception logged along with sufficient information to determine the cause of the error, or reproduce it at least.

# Validation - Throw Exceptions Early

When receiving input that needs to be validated before it can be used, validate all input before using any of it. You whould not change any state in the application or attached systems until all input data has been validated. That way you avoid leaving the application in a half valid state.

For instance, in a DAO method that inserts a user and an address into two different tables in a database, do like this:

check if user already exists

validate user

validate address

insert user

insert address

Do not do like this:

check if user already exists

validate user

insert user

validate address

insert address

If you do like this, and the address turns out to be invalid you will still have inserted the user in the system. What happens when the user corrects the address and retry registering? He will be told that his user already exists, right?

The problem is not only related to database insertions. In general, validate all input before using any of it, to avoid leaving the application in a half valid (=unknown) state.

# Validation - Throw Exception or Return False?

When validating input data you do not always have to throw an exception if the input is invalid. How to handle invalid data or state sensibly often depends on the condition:

Can the code detecting the error, interact sensibly with the

user, to correct the error?

If the error is caused by input from a user, and that user is able to correct the input and retry, then interacting is often preferrable to throwing an exception. In addition you might want to detect all possible errors in the input before showing them to the user. That way the user can correct all the errors before next retry, instead of repeated iterations of correcting, retrying, correcting, retrying etc. For instance, in a servlet you could do like this:

public void service(HttpServletRequest request,

HttpServletResponse response){

boolean isStreetvalid = validateStreet (request);

boolean isZipCodeValid = validateZipCode(request);

boolean isPhoneValid = validatePhone (request);

if(isStreeValid && isZipCodeValid && isPhoneValid){

//perform action

} else {

//redirect back to HTML form and show the three errors.

}

}

Notice how none of the validation methods throw exceptions. Rather they return true or false. There is no need to throw an exception even if the input data is invalid.

In contrast, inside a DAO class you will most likely not be able to interact with the user to correct the error. A DAO object may not even know if it is called from inside a web service, a servlet, or somewhere else. It has no way of knowing if the input came from another system, a file, or a user. It has no way of knowing how, or even if, the error could be sorted out through interact with the user.

In this case the standard way for the DAO object to signal the error is to throw an exception. Other mechanisms like return codes etc. could be used, but it doesn't change the fact that the DAO cannot handle the error itself. The input data should have been validated before the call to the DAO. Here is how it could look:

public void insertAddress(Address address){

validate(address);

//insert address

}

public void validate(Address address) {

if(isInvalidCity(address.getCity()){

throw new InvalidArgumentException(

"City " + address.getCity() + " is not valid");

}

// more if-statements like the one above,

// validating each field in the address object.

}

In this example the validate() method throws an exception as soon as an validation error is found. The insertAddress method cannot interact with the user to correct the error. Thus it is appropriate to throw an exception to signal the error. Alternatively all errors could be detected before throwing the exception.

# Exception Handling Templates in Java

Before you read this text, it is a good idea to have read the text ["Fail Safe Exception Handling"](http://tutorials.jenkov.com/java-exception-handling/fail-safe-exception-handling.html).

Correct exception handling code can be tedious to write. Try-catch blocks also clutter the code and makes it harder to read. Look at the example below:

Input input = null;

IOException processException = null;

try{

input = new FileInputStream(fileName);

//...process input stream...

} catch (IOException e) {

processException = e;

} finally {

if(input != null){

try {

input.close();

} catch(IOException e){

if(processException != null){

throw new MyException(processException, e,

"Error message..." +

fileName);

} else {

throw new MyException(e,

"Error closing InputStream for file " +

fileName;

}

}

}

if(processException != null){

throw new MyException(processException,

"Error processing InputStream for file " +

fileName;

}

In this example no exceptions are lost. If an exception is thrown from within the try block, and another exception is thrown from the input.close() call in the finally block, both exceptions are preserved in the MyException instance, and propagated up the call stack.

That is how much code it takes to handle the processing of an input stream without any exceptions being lost. In fact it only even catches IOExceptions. RuntimeExceptions thrown from the try-block are not preserved, if the input.close() call also throws an exception. Isn't it ugly? Isn't it hard to read what is actually going on? Would you remember to write all that code everytime you process an input stream?

Luckily there is a simple design pattern, the Template Method, that can help you get the exception handling right everytime, without ever seeing or writing it in your code. Well, maybe you will have to write it once, but that's it.

What you will do is to put all the exception handling code inside a template. The template is just a normal class. Here is a template for the above input stream exception handling:

public abstract class InputStreamProcessingTemplate {

public void process(String fileName){

IOException processException = null;

InputStream input = null;

try{

input = new FileInputStream(fileName);

**doProcess(input);**

} catch (IOException e) {

processException = e;

} finally {

if(input != null){

try {

input.close();

} catch(IOException e){

if(processException != null){

throw new MyException(processException, e,

"Error message..." +

fileName);

} else {

throw new MyException(e,

"Error closing InputStream for file " +

fileName;

}

}

}

if(processException != null){

throw new MyException(processException,

"Error processing InputStream for file " +

fileName;

}

}

//override this method in a subclass, to process the stream.

**public abstract void doProcess(InputStream input) throws IOException;**

}

All the exception handling is encapulated inside the InputStreamProcessingTemplate class. Notice how the process() method calls the doProcess() method inside the try-catch block. You will use the template by subclassing it, and overriding the doProcess() method. To do this, you could write:

new InputStreamProcessingTemplate(){

public void doProcess(InputStream input) throws IOException{

int inChar = input.read();

while(inChar !- -1){

//do something with the chars...

}

}

}.process("someFile.txt");

This example creates an anonymous subclass of the InputStreamProcessingTemplate class, instantiates an instance of the subclass, and calls its process() method.

This is a lot simpler to write, and easier to read. Only the domain logic is visible in the code. The compiler will check that you have extended the InputStreamProcessingTemplate correctly. You will typically also get more help from your IDE's code completion when writing it, because the IDE will recognize both the doProcess() and process() methods.

You can now reuse the InputStreamProcessingTemplate in any place in your code where you need to process a file input stream. You can easily modify the template to work for all input streams and not just files.

## Using Interfaces Instead of Subclassing

Instead of subclassing the InputStreamProcessingTempate you could rewrite it to take an instance of an InputStreamProcessor interface. Here is how it could look:

public interface InputStreamProcessor {

public void process(InputStream input) throws IOException;

}

public class InputStreamProcessingTemplate {

public void process(String fileName, InputStreamProcessor processor){

IOException processException = null;

InputStream input = null;

try{

input = new FileInputStream(fileName);

processor.process(input);

} catch (IOException e) {

processException = e;

} finally {

if(input != null){

try {

input.close();

} catch(IOException e){

if(processException != null){

throw new MyException(processException, e,

"Error message..." +

fileName;

} else {

throw new MyException(e,

"Error closing InputStream for file " +

fileName);

}

}

}

if(processException != null){

throw new MyException(processException,

"Error processing InputStream for file " +

fileName;

}

}

}

Notice the extra parameter in the template's process() method. This is the InputStreamProcessor, which is called from inside the try block (processor.process(input)). Using this template would look like this:

new InputStreamProcessingTemplate()

.process("someFile.txt", new InputStreamProcessor(){

public void process(InputStream input) throws IOException{

int inChar = input.read();

while(inChar !- -1){

//do something with the chars...

}

}

});

It doesn't look much different from the previous usage, except the call to the InputStreamProcessingTemplate.process() method is now closer to the top of the code. This may be easier to read.

## Static Template Methods

It is also possible to implement the template method as a static method. This way you don't need to instantiate the template as an object every time you call it. Here is how the InputStreamProcessingTemplate would look as a static method:

public class InputStreamProcessingTemplate {

public **static** void process(String fileName,

InputStreamProcessor processor){

IOException processException = null;

InputStream input = null;

try{

input = new FileInputStream(fileName);

processor.process(input);

} catch (IOException e) {

processException = e;

} finally {

if(input != null){

try {

input.close();

} catch(IOException e){

if(processException != null){

throw new MyException(processException, e,

"Error message..." +

fileName);

} else {

throw new MyException(e,

"Error closing InputStream for file " +

fileName;

}

}

}

if(processException != null){

throw new MyException(processException,

"Error processing InputStream for file " +

fileName;

}

}

}

The process(...) method is simply made static. Here is how it looks to call the method:

InputStreamProcessingTemplate.process("someFile.txt",

new InputStreamProcessor(){

public void process(InputStream input) throws IOException{

int inChar = input.read();

while(inChar !- -1){

//do something with the chars...

}

}

});

Notice how the call to the template's process() method is now a static method call.

## Summary

Exception handling templates are a simple yet powerful mechanism that can increase the quality and readability of your code. It also increases your productivity, since you have much less code to write, and less to worry about. Exceptions are handled by the templates. And, if you need to improve the exception handling later in the development process, you only have a single spot to change it in: The exception handling template.

The Template Method design pattern can be used for other purposes than exception handling. The iteration of the input stream could also have been put into a template. The iteration of a ResultSet in JDBC could be put into a template. The correct execution of a transaction in JDBC could be put into a template. The possibilities are endless.

Context reuse and templates are also discussed in the article [Code Reuse: Context and Action Reuse](http://tutorials.jenkov.com/ood/code-reuse-action-and-context-reuse.html).

# Exception Enrichment in Java

Exception enrichment is an alternative to [exception wrapping](http://tutorials.jenkov.com/java-exception-handling/exception-wrapping.html). Exception wrapping has a couple of disadvantages that exception enrichment can fix. These disadvantages are:

* Exception wrapping may result in very long stack traces consisting of one stack trace for each exception in the wrapping hierarchy. Most often only the root stack trace is interesting. The rest of the stack traces are then just annoying.
* The messages of the exceptions are spread out over the stack traces. The message of an exception is typically printed above the stack trace. When several exceptions wrap each other in a hierarchy, all these messages are spread out in between the stack traces. This makes it harder to determine what went wrong, and what the program was trying to do when the error happened. In other words, it makes it hard to determine in what context the error occurred. The error might have occurred in a PersonDao class, but was it called from a servlet or from a web service when it failed?

In exception enrichment you do not wrap exceptions. Instead you add contextual information to the original exception and rethrow it. Rethrowing an exception does not reset the stack trace embedded in the exception.

Here is an example:

public void method2() throws EnrichableException{

try{

method1();

} catch(EnrichableException e){

e.addInfo("An error occurred when trying to ...");

throw e;

}

}

public void method1() throws EnrichableException {

if(...) throw new EnrichableException(

"Original error message");

}

As you can see the method1() throws an EnrichableException which is a superclass for enrichable exceptions. This is not a standard Java exception, so you will have to create it yourself. There is an example EnrichableException at the end of this text.

Notice how method2() calls the addInfo() method on the caught EnrichableException, and rethrow it afterwards. As the exception propagates up the call stack, each catch block can add relevant information to the exception if necessary.

Using this simple technique you only get a single stack trace, and still get any relevant contextual information necessary to investigate the cause of the exception.

## Unique Error Codes

It is sometimes a requirement that each error raised in an application is identified by a unique error code. This can be a bit problematic since some errors are raised inside components that are reused throughout the application. Therefore an exception may seem the same to the component throwing it, but the context in which the exception occurs is different.

Here is an example:

public void method3() throws EnrichableException{

try{

method1();

} catch(EnrichableException e){

e.addInfo("An error occurred when trying to ...");

throw e;

}

}

public void method2() throws EnrichableException{

try{

method1();

} catch(EnrichableException e){

e.addInfo("An error occurred when trying to ...");

throw e;

}

}

public void method1() throws EnrichableException {

if(...) throw new EnrichableException(

"ERROR1", "Original error message");

}

Notice how method1() adds the code "ERROR1" to the thrown EnrichableException to uniquely identify that error cause. But notice too that method1() is called from both method2() and method3(). Though the error may seem the same to method1() no matter which of method2() and method3() that called it, this may important to know for the developer investigating the error. The error code "ERROR1" is enough to determine where the error occurred, but not in what context it occurred.

A solution to this problem is to add unique context error codes to the exception the same way the other contextual information is added. Here is an example where the addInfo() method has been changed to accommodate this:

public void method3() throws EnrichableException{

try{

method1();

} catch(EnrichableException e){

e.addInfo("METHOD3", "ERROR1",

"An error occurred when trying to ...");

throw e;

}

}

public void method2() throws EnrichableException{

try{

method1();

} catch(EnrichableException e){

e.addInfo("METHOD2", "ERROR1",

"An error occurred when trying to ...");

throw e;

}

}

public void method1() throws EnrichableException {

if(...) throw new EnrichableException(

"METHOD1", "ERROR1", "Original error message");

}

Two new parameters have been added to the addInfo() method and the constructor of the EnrichableException. The first parameter is a code identifying the context in which the error occurred. The second parameter is a unique error code within that context. An error identification for an exception thrown by method1() when called from method2() will now look like this:

[METHOD2:ERROR1][METHOD1:ERROR1]

When method1() is called from method3() the error identification will look like this:

[METHOD3:ERROR1][METHOD1:ERROR1]

As you can see it is now possible to distinguish an exception thrown from method1() via method2() from the same exception thrown from method1() via method3().

You may not always need the extra contextual error codes, but when you do the solution sketched in this section is an option.

## Wrapping Non-Enrichable Exceptions

You may not always be able to avoid exception wrapping. If a component in your application throws a checked exception that is not enrichable, you may have to wrap it in an enrichable exception. Here is an example where method1() catches a non-enrichable exception and wraps it in an enrichable exception, and throws the enrichable exception:

public void method1() throws EnrichableException {

try{

... call some method that throws an IOException ...

} catch(IOException ioexception)

throw new EnrichableException( ioexception,

"METHOD1", "ERROR1", "Original error message");

}

## Unchecked EnrichableException

I used to be in favor of checked exceptions but over the last couple of years my opinion has changed. Now i feel that checked exceptions are more of a nuisance than a help. Therefore I would prefer to make my EnrichableException unchecked, by having it extend RuntimeException.

There is a more thorough discussion of checked and unchecked exceptions in the text ["Checked vs. Unchecked Exceptions"](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html).

## Exception Enrichment and Pluggable Exception Handlers

Like with any other exception type it is possible to use pluggable exception handlers with enrichable exceptions. If you use the unique error codes described earlier these codes must be added as paremeters to the exception handler interface. Here is an example exception handler interface supporting these unique error codes:

public interface ExceptionHandler{

public void handle(String contextCode, String errorCode,

String errorText, Throwable t)

public void raise(String contextCode, String errorCode,

String errorText);

}

Exceptions caught in the program will be passed to the handleException() which will decide what concrete exception to throw instead. In this case an EnrichableException is thrown. If the EnrichableException is unchecked it is not necessary to declare it in the handleException() method.

## An Example EnrichableException

Below is an example of an enrichable exception that you can use as a template for your own enrichable exceptions. You may need to change the class definition to suit your own needs. The exception is designed to use unique error codes as described earlier in this text.

Below the code is an example application that uses the EnrichableException, and a stack trace generated from this application.

import java.util.ArrayList;

import java.util.List;

public class EnrichableException extends RuntimeException {

public static final long serialVersionUID = -1;

protected List<InfoItem> infoItems =

new ArrayList<InfoItem>();

protected class InfoItem{

public String errorContext = null;

public String errorCode = null;

public String errorText = null;

public InfoItem(String contextCode, String errorCode,

String errorText){

this.errorContext = contextCode;

this.errorCode = errorCode;

this.errorText = errorText;

}

}

public EnrichableException(String errorContext, String errorCode,

String errorMessage){

addInfo(errorContext, errorCode, errorMessage);

}

public EnrichableException(String errorContext, String errorCode,

String errorMessage, Throwable cause){

super(cause);

addInfo(errorContext, errorCode, errorMessage);

}

public EnrichableException addInfo(

String errorContext, String errorCode, String errorText){

this.infoItems.add(

new InfoItem(errorContext, errorCode, errorText));

return this;

}

public String getCode(){

StringBuilder builder = new StringBuilder();

for(int i = this.infoItems.size()-1 ; i >=0; i--){

InfoItem info =

this.infoItems.get(i);

builder.append('[');

builder.append(info.errorContext);

builder.append(':');

builder.append(info.errorCode);

builder.append(']');

}

return builder.toString();

}

public String toString(){

StringBuilder builder = new StringBuilder();

builder.append(getCode());

builder.append('\n');

//append additional context information.

for(int i = this.infoItems.size()-1 ; i >=0; i--){

InfoItem info =

this.infoItems.get(i);

builder.append('[');

builder.append(info.errorContext);

builder.append(':');

builder.append(info.errorCode);

builder.append(']');

builder.append(info.errorText);

if(i>0) builder.append('\n');

}

//append root causes and text from this exception first.

if(getMessage() != null) {

builder.append('\n');

if(getCause() == null){

builder.append(getMessage());

} else if(!getMessage().equals(getCause().toString())){

builder.append(getMessage());

}

}

appendException(builder, getCause());

return builder.toString();

}

private void appendException(

StringBuilder builder, Throwable throwable){

if(throwable == null) return;

appendException(builder, throwable.getCause());

builder.append(throwable.toString());

builder.append('\n');

}

[L1:E1][L2:E2][L3:E3]

[L1:E1]Error in level 1, calling level 2

[L2:E2]Error in level 2, calling level 3

[L3:E3]Error at level 3

java.lang.IllegalArgumentException: incorrect argument passed

at exception.ExceptionTest$1.handle(ExceptionTest.java:8)

at exception.ExceptionTest.level3(ExceptionTest.java:49)

at exception.ExceptionTest.level2(ExceptionTest.java:38)

at exception.ExceptionTest.level1(ExceptionTest.java:29)

at exception.ExceptionTest.main(ExceptionTest.java:21)

Caused by: java.lang.IllegalArgumentException: incorrect argument passed

at exception.ExceptionTest.level4(ExceptionTest.java:54)

at exception.ExceptionTest.level3(ExceptionTest.java:47)

... 3 more

public class ExceptionTest {

protected ExceptionHandler exceptionHandler = new ExceptionHandler(){

public void handle(String errorContext, String errorCode,

String errorText, Throwable t){

if(! (t instanceof EnrichableException)){

throw new EnrichableException(

errorContext, errorCode, errorText, t);

} else {

((EnrichableException) t).addInfo(

errorContext, errorCode, errorText);

}

}

public void raise(String errorContext, String errorCode,

String errorText){

throw new EnrichableException(

errorContext, errorCode, errorText);

}

};

public static void main(String[] args){

ExceptionTest test = new ExceptionTest();

try{

test.level1();

} catch(Exception e){

e.printStackTrace();

}

}

public void level1(){

try{

level2();

} catch (EnrichableException e){

this.exceptionHandler.handle(

"L1", "E1", "Error in level 1, calling level 2", e);

throw e;

}

}

public void level2(){

try{

level3();

} catch (EnrichableException e){

this.exceptionHandler.handle(

"L2", "E2", "Error in level 2, calling level 3", e);

throw e;

}

}

public void level3(){

try{

level4();

} catch(Exception e){

this.exceptionHandler.handle(

"L3", "E3", "Error at level 3", e);

}

}

public void level4(){

throw new IllegalArgumentException("incorrect argument passed");

}

}