# Exception Handling: Common Problems and Best Practice with Java 1.4

Dr. Andreas Müller and Geoffrey Simmons

Sun Microsystems GmbH, Sun Java Center,
Sonnenallee 1, 85551 Kirchheim-Heimstetten,
Andreas.H.Mueller@Sun.COM, Geoffrey.Simmons@Sun.COM

**Abstract.** Many real-world Java applications fall short of Java's excellent exception handling capabilities. In this article we point out several design and programming mistakes encountered in industry projects. We give advice on best practice concepts of Java exception handling and cover enhanced exception handling features in Java 1.4.

## 1 Motivation

Java exception handling is superior to most languages' and platforms' capabilities:

- The JVM rarely crashes, even as a result of serious problems in the code or runtime environment and thus in most cases successfully delivers valuable error information.
- Stack trace information support is built into the exception classes and does not need proprietary extensions.
- Checked exceptions are a way to emphasize through compiler support which exceptions of an API could and should be treated in the caller's code.

In summary, Java offers excellent capabilities to build reliable software which simplifies error diagnosis and maintenance. Unfortunately, in industrial practice this potential is rarely used to its full extent. Misunderstanding of Java exceptions and negligence at the design and programming level waste some of Java's advantages. In the following sections we point out some common problems encountered in real-life applications.

## 2 Some Common Problems

### 2.1 Problem 1: Empty Catch Blocks

Empty catch blocks found in Java applications in many cases look like the following code snippets:

```
DateFormat format = DateFormat.getDateInstance(SHORT);
...
private final static String DEFAULT_DATE_STRING=”01.01.1900”;
...
try {
    Date defaultDate=format.parse(DEFAULT_DATE_STRING);
    ...
} catch( ParseException pexc ) {
   // Cannot happen: if it works once, it will always work!
}
```

The idea behind this code is that the only argument to the `parse` method is a constant chosen by the programmer to fit the used `DateFormat`, and therefore the `ParseException` will never be thrown in practice; hence, no code to deal with this kind of exception is needed.

Although this reasoning might appear sound at first glance, it is not, for two reasons:

- During maintenance of this code a programmer might switch to a different `DateFormat` and forget to change the constant accordingly
- The above code depends on the `Locale`. It might therefore be broken by installation on a different machine without any code changes.

In both cases a `ParseException` will be thrown and caught silently and the primary problem will remain undetected. If `defaultDate` is uninitialized at that time this will lead to a follow-up `NullPointerException`. Otherwise the program will continue with a wrong value of `defaultDate`. In both cases the original problem will be difficult to identify. Therefore empty catch blocks in most[1] cases are a serious threat to maintainability and reliability, two main criteria of software quality.

The problem with empty catch blocks arises because the exception is *checked*, i.e. not inherited from `RuntimeException`. Programmers must either declare it in a `throws` clause or provide a `catch` block. Both choices seem inadequate for an exception that is not expected to occur. In the best practice chapter, we will show how this problem can be solved using unchecked exceptions in an adequate manner.

### 2.2 Problem 2: Meaningless Throws Clauses

Many projects define a common base class for application-specific exceptions:

```
public class OurAppBaseException extends Exception{…}
```

This can be a good idea in order to define common behaviour of all exceptions. It is a design bug, however, to replace specific throws clauses in all methods of the application by blanket declarations such as this:

```
public void method1 throws OurAppBaseException  {...}
public String method2 throws OurAppBaseException {
    ...
    } catch( FileNotFoundException e ) {
         throw new OurAppBaseException(e.getMessage( ));
    }
}
```

[1] They might be justified if an exception is harmless and doesn’t need treatment (e.g. InterruptedException), but never if an exception supposedly cannot occur.

It is the purpose of checked exceptions to provide specific information to the caller about exceptions that might be thrown in a method, and to have exception handling formally checked by the compiler. The above practice undermines that purpose while formally maintaining the syntax of checked exceptions. To fully appreciate this argument note that if `OurAppBaseException` were unchecked (inherited from `RuntimeException`), all throws clauses could be dropped from all methods without changing their semantics. This is a legitimate choice (compare C++ !) but it should not be mixed up with adequate usage of checked exceptions.

### 2.3 Problem 3: Loss of Stack Information

In many cases it is desirable to change an exception's class by catching it and rethrowing a different exception object. One good reason for this, among others, is that in many cases low-level exceptions do not correspond to a method's abstraction level:

```
private void init() throws InitializationException {
   ...
   } catch(  FileNotFoundException fnfexc)  {
     throw new InitializationException(fnfexc.getMessage());
   }
}
```

Even in cases where this is a good idea, it is a problem that valuable information about the original exception will not be propagated. In the example above, the programmer tried to limit the problem by copying the original message text to the new exception. In this case other pieces of information, such as the original exception's class and stack trace, are still lost. Its full information content can only be preserved by a technique called exception chaining (or nesting). Exception chaining is generically supported in Java 1.4, as will be discussed in chapter 3.

### 2.4 Problem 4: Incomprehensible and Incomplete Exception Logs

Log files are an invaluable source of diagnostic information, but many of them, unfortunately, are rather difficult to read and understand. In many cases one single exception leads to multiple entries in the same log file. From the look of the log file, it is hard to tell whether a thunderstorm of exceptions or one single exception has happened in that system. Obviously, programmers want to make sure that exceptions are logged immediately as they occur. Because 'immediately' is not well defined while an exception travels up the call stack, it almost inevitably will be logged several times. We suspect that the eagerness to log an exception immediately is due to a fear of a sudden runtime system crash. In C/C++ environments, follow-up exceptions would frequently kill the process before valuable information could have been saved to disk, and thus lead to fruitless searches for the cause. The panic stemming from such experience is understandable but not appropriate for a Java runtime environment (without JNI, that bridge from Java to the world of sudden deaths).

In contrast, many programmers record very little state information when logging exceptions, on the grounds that they do not want to "clutter up" the log file. As a consequence, more often than necessary the exception log is not sufficient to identify a problem. Instead, time-consuming modifications of the software and attempts to reproduce the error are necessary.

# 3 Enhanced Exception Handling Support in Java 1.4

## 3.1 Exception Chaining

Exception chaining means that the constructor of an exception object takes a 'nested' or 'cause' exception as an argument. Some exception classes have offered such constructors for some time:

```
RemoteException( String s, Throwable ex )
ServletException( String message, Throwable rootCause )
```

The newly created higher level exception object will then maintain a reference to the underlying root cause exception. If the higher level exception is called to print its stack trace, it will include the stack trace of the root exception as part of its own output. Basically, there is no reason, why this concept should be limited to certain exception classes and to a depth of only one nested exception inside one higher level exception. In Java version 1.4 support for it has therefore been built into the uppermost exception base class:

```
Throwable( String message, Throwable cause )
Throwable( Throwable cause )
```

This allows us to nest exceptions of arbitrary class to arbitrary depth, thereby forming an exception chain of arbitrary length. Exception chains contain information about the full call stack, thus allowing a change of type on the exception's way up the call stack without loss of stack information. This concept will be further discussed as 'exception abstraction' in the best practice section.

## 3.2 Assertions

Assertions are new to Java 1.4. They make it possible to improve software quality by abundant use of runtime checks for internal preconditions, postconditions and invariants similar to C assertion macros. See more details in [Ass02]. For a test and deployment period, assertions are enabled by a runtime switch (java –ea) which can also be limited to specific classes and packages. If this switch is omitted for production, all assertions are disabled and their negative performance impact is gone. Assertions are a valuable feature long awaited by many programmers. In the best practice chapter we outline the usage of assertions in contrast to unchecked exceptions.

### 3.3 Other exception handling enhancements

Prior to Java 1.4 the only thing you could do with a stack trace was to print it to a stream:

```
public void Throwable.printStackTrace( PrintStream s )
```

Exception stack traces should be included in log files, but most logging APIs require the log entry as a String. So in pre-1.4 systems, the stack trace had to be written to a StringWriter, from which a String had to be extracted. Now with Java 1.4 the stack trace offers an API for programmatic manipulation, eliminating the need for such detours:

```
public StackTraceElement[] Throwable.getStackTrace()
```

This feature will hardly be used in application programming but rather in library packages with special needs to manipulate exceptions. One example of such a library is the new logging package `java.util.logging` which has been incorporated into Java 1.4 [Log02]. For many projects, that have not used a logging package at all or only a minimal home-written logger class, this new logging API offers the opportunity to use more powerful logging facilities. As for other packages the logging of exceptions is an important feature. One advantage of this new logging package as compared to third party products is that some standard Java packages (e.g. RMI) already use it internally and others will follow. Application developers therefore can use a common logging package for system level and application level logging which is easily configured to application specific needs .

## 4 Best Practice Guidelines

### 4.1 Exception Handling as a Key Design Issue

Best practice requires that exception handling and related subjects be taken seriously as key design issues throughout the whole software life cycle. A considerable percentage of the code of the final system (usually more than initially estimated) will be dedicated to exception handling. Exception handling will also determine to a large extent the quality of the software. Therefore it is crucial to treat exception handling proactively instead of fixing things later, usually only after a lot of code has already been written. Here, we limit ourselves to a list of some recommendations which cover all phases of the software development process:

- collect exception handling and logging requirements as systematically as business requirements in early phases
- clarify early how error messages for users (and administrators) are maintained and brought to the user screen at runtime
- design an initial exception hierarchy before implementation starts and decide how checked and unchecked exceptions should be used
- prepare guidelines for how and where exceptions are logged
- monitor quality of exception handling code regularly during implementation

- read log files after tests and make sure you fully understand them before roll-out of the software

### 4.2 Usage of Unchecked Exceptions

There is a misunderstanding that unchecked exceptions have no place in Java programming, although early [Ven98] and more recent publications [Blo01] emphasize that they stand equal beside checked exceptions. Just to make it clear again:

**Exceptions that signal an untreatable situation should be unchecked.**

This concept is present throughout the Java core packages where `NullPointerException`, `IndexOutOfBoundsException` and other program bugs are inherited from `RuntimeException` and thus are unchecked. Program bugs should be fixed, not treated in code. As these exceptions can be thrown anywhere in your code but you don't want to declare them in all your methods, it's a good idea to make them unchecked. Going one step further you should define the following exception for your application:

```
public class ProgrammingException extends RuntimeException
```

This exception class is useful for all higher-level programming bugs. One important example is the empty catch block problem mentioned above:

```
try {
   Date defaultDate=format.parse(DEFAULT_DATE_STRING);
   ...
} catch( ParseException pexc ) {
   // If this exception is thrown, I got something wrong
   throw new ProgrammingException(“bad init value”,pexc);
}
```

Compared to the original claim that the exception 'cannot be thrown', the idea remains the same but is expressed in a much more fault-tolerant way. Note also in this example that exception chaining is used, so that the information associated with the original `ParseException` is not lost.

Besides programming bugs, there are other obvious uses for unchecked exceptions. Installation and configuration problems cannot not be treated in code either. Provide unchecked exception classes for them and use them as in this example:

```
try {
   FileInputStream fin=new FileInputStream(configfilename);
   ...
} catch( FileNotFoundException fnfexc ) {
   throw new InstallationException(“missing file”,fnfexc);
}
```

The `FileNotFoundException` and the `ParseException` above are checked because from the perspective of the Java core packages, they might be treatable depending on the application context. In the example contexts shown, it is clear that they are not treatable. Therefore they are chained into unchecked exceptions which simplifies their further handling and adds information to them. The fact that a programmer linked a certain `FileNotFoundException` to an

`InstallationException` will certainly help the system administrator to analyze the problem once that exception chain has been written to a log file.

### 4.3 Usage of Checked Exceptions

To continue what we have said in the last section:

**checked exceptions signal an exceptional situation which might be treatable.**

Because knowledge inside the called method is incomplete, the decision is left to the caller. A checked exception helps to ensure that the caller accepts this responsibility. This rule can be applied to the following situations:

- Exceptions which signal a problem at the level of a method's purpose (including 'business' level exceptions) are usually checked because by definition, their treatment is out of the method's scope and left to the caller.
- Badly formatted user input can be signalled by checked exceptions, because displaying an exception message and waiting for the user's retry is a kind of treatment.
- Any technical exception inside a server could be converted to a checked `InternalServerException` in the server's API, because the client might try to solve the situation by retrying or by calling a different server instance.

The first item is at the heart of checked exceptions in Java and should be clarified by an example. Let's start from a method which is used to buy shares in a stock trading application:

```
public void buyStock(int numOfShares, StockSymbol stock)
```

If this method can fail because there is not enough money on the cash account it must throw a checked exception:

```
public void buyStock(...) throws NotEnoughCashException
```

If we know from context that this method will only be called after a check of the cash account, this knowledge must be documented by the caller of the method with a `ProgrammingException`:

```
try {
   trader.buyStock(num,symbol);
} catch( NotEnoughCashException necexc ) {
   // Cash should have been checked before
   throw new ProgrammingException("no cash check",necexc);
}
```

As much as an empty catch block, it would be a design bug to eliminate the `NotEnoughCashException` from the method's API due to this knowledge. This would lead to a fragile application with weak diagnostic capabilities.

If the application does not check cash before calling the buyStock() method, all the methods on the call stack need to propagate the `NotEnoughCashException` in their own `throws` clauses until it finally reaches the user interface and a message is displayed to the user. In this case the checked exception needs some reference to an error message. For many practical reasons, it usually is not adequate to display the hardcoded exception message text to the user; the exception message should only be used to communicate information to developers, whereas the display of error

messages to users is a task of the presentation level, and should be implemented at that level. A mechanism for identifying errors with sufficient granularity, so that the right error messages can be selected for them, should be implemented at the level of an application-specific checked exception base class, for example by including a member field for an error code that can be used as an index into a table of messages.
To summarize these ideas, Figure 1 illustrates a sample exception hierarchy that reflects our discussion of checked vs. unchecked exceptions.

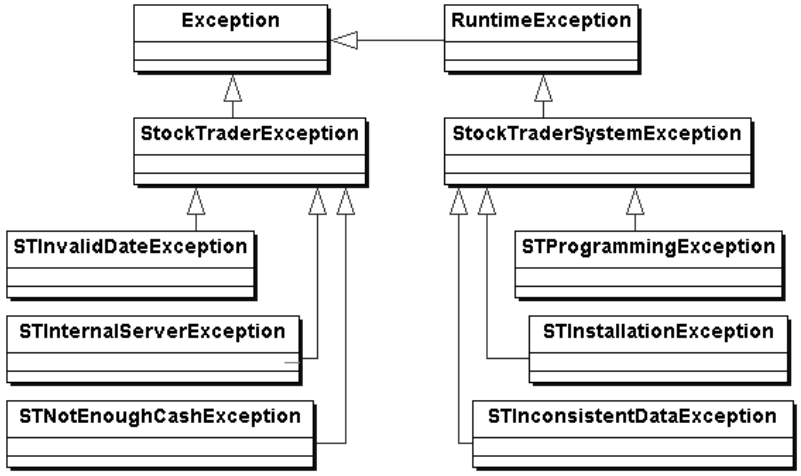


**Fig. 1.** A sample exception hierarchy for a 'StockTrader' application

In most cases an appropriate use of unchecked and checked exceptions will reduce checked exceptions per method to a small number. There is no need for blanket declaration of high level exception classes as in problem 2.

### 4.4 Exception Abstraction and Chaining

As seen in the previous sections, it is useful to change the class of exceptions on their way up the call stack. This is due to the fact that context knowledge changes depending on the method and its abstraction level. Exception chaining as shown in all of the above examples is essential for this concept to work. Loss of call stack information would otherwise be damaging to error analysis. In pre-1.4 systems, exception chaining needs to be implemented by hand, but the benefits justify the effort to do so [Goe01][Blo01]. With improved support for exception chaining in Java 1.4, an obstacle to more widespread use of this concept has been removed. Information rich exception chains should therefore help to further improve the diagnostic strength and quality of Java applications.

### 4.5 Assertions vs. Unchecked Exceptions

Assertions as a way to guarantee internal code consistency can be seen as a competitor to some uses of unchecked exceptions. Would it be useful to replace `ProgrammingException` or `InstallationException` in the examples above by assertions? A code example would be the following:

```
try {
   Date defaultDate=format.parse(DEFAULT_DATE_STRING);
   ...
} catch( ParseException pexc ) {
   // This must never happen!
   assert false : “bad init value”;
}
```

In this and similar cases the answer is no for several reasons:

- assertions provide no means to chain an original exception
- the constant boolean in the assertion indicates an abuse of the concept
- if assertions are disabled we end up with an empty catch block

To summarize, inside a catch block it is too late for assertions. Assertions should be applied well in advance of a particular problem. By redundant checks they safeguard assumptions about the code inside a single software unit of development and testing. From the redundancy of the checks follows that the performance advantage of assertions compared to normal checks is important. Regarding single software units of development and testing, note that a client and a server component often do not form one. It follows that it is a questionable choice to check preconditions of a server API method with assertions. A newly developed client might violate the API while the server runs without assertion checking. After so many caveats we should note that the benefit of assertions is in no doubt for checking postconditions in all kinds of methods and for checking preconditions in private methods, to give just two examples.

### 4.6 Exception Logging

An important part of exception handling is exception logging. This is particularly true for multi-tier and web applications where an exception on its way from the source to the user screen can traverse several virtual machines. In many cases the user who finally receives an error message has neither enough information nor interest to provide a useful error report to the administrators of all the systems contributing to a service. It is therefore essential that each system keep its own log, in which exceptions can be tracked as part of an ongoing quality assurance process.

To make these log files as readable as possible, it is very helpful to log an exception exactly once into a logfile. The right time to log an exception in Java is the latest possible, because the JVM most probably will not crash before and by chaining exceptions gain information content while they travel up the call stack. Exceptions should therefore only be logged when they

- are eventually being treated (caught with no rethrow)
- are leaving a physical tier / virtual machine through a remote call
- are leaving a logical tier that writes its own logfile

If additional 'early' log entries of the same exception are really needed, they should at least use a lower log level for distinction.
For programming exceptions the log should contain as much information about the state of the system as is available. For example, when errors are detected at the web presentation level, the log entry should contain, at least, a dump of the internal states of all of the objects available in the servlet API, incuding the request, response and session objects, including the contents of their attribute tables. The states of all other available objects, including EJB's, should be logged as well. We see no need for restraint here, since the cause for the problem could potentially be anywhere in the system, and a resolution of the problem has highest priority. Although the performance impact might not be negligible, it does not hit the system as long as it functions properly. Of course, a well-structured format for log entries makes extensive diagnostic information easier to examine.

## 5 Summary and Conclusion

In this article we have presented several programming and design mistakes regarding Java exception handling. Such problems emphasize that exception handling always is a key issue in building quality applications. For their solution it is essential to fully understand the concept of checked vs. unchecked exceptions. Checked exceptions supplement unchecked exceptions but do not substitute them. Appropriate use of both exception types together with exception chaining and deliberate logging are key factors for building Java applications with excellent maintainability and self diagnosis. Java 1.4 contributes to that effort with improved support for exception chaining, assertions and logging.

## References


[Ass02] Programming With Assertions, http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html

[Blo01] Joshua Bloch:Effective Java,Programming Language Guide,Addison Wesley,2001

[Goe01] Brian Goetz:Exceptional practices,Part 2, http//developer.java.sun.com/developer/technicalArticles/Programming/exceptions2/

[JLS00] Java Language Specification, http://java.sun.com/docs/books/jls/second_edition/html/exceptions.doc.html

[Log02] Java Logging APIs,http://java.sun.com/j2se/1.4/docs/guide/util/logging/index.html

[Ven98] Bill Venners: Exceptions in Java, http://www.javaworld.com/javaworld/jw-07-1998/jw-07-exceptions.html