Analyzing Exception Usage in Large Java Applications

Darrell Reimer and Harini Srinivasan

IBM Research, 19 Skyline Drive, Hawthorne, NY, USA 10532
{dreimer, harini}@us.ibm.com

Abstract. The Java programming language provides a way of identifying when semantic constraints of the program are violated using its *exception* mechanism. Whenever a semantic constraint in the program is violated, control flow is transferred from the point where the exception happened (throw site) to the point specified by the programmer (catch site). While this is indeed a robust and portable mechanism of handling semantic errors and exceptional program behavior, the mechanism is often misused and/or abused. In our experience working with large J2EE applications, we have encountered several inappropriate exceptions usage patterns that have made maintainability of these applications extremely difficult. Proper exception usage is necessary to minimize time from problem appearance to problem isolation and diagnosis. This article discusses some common trends in the use of exceptions in large Java applications that make servicing and maintaining these long running applications extremely tedious. The paper also proposes some solutions to avoid or correct these misuses of exceptions.

1 Introduction

The Java Virtual Machine [12] uses exceptions to signal semantic errors in a program. In particular, whenever a semantic error occurs, the JVM raises an exception. It is the responsibility of the application programmer to both (a) identify when such exceptions can happen, i.e., when semantic errors can happen and (b) catch these exceptions in a manner that helps identify them during program execution. Exceptions can also be used to *remedy* an incorrect execution behavior of the application.

Proper handling of exceptions is extremely important to be able to manage and service large J2EE applications. For example, consider a J2EE application that handles online banking transactions. Typically, the financial institution would like the application to run in a 24x7 mode to be able to service their customers continuously. The cost of stopping and starting these applications is usually very high for these institutions. Given this scenario, if a failure happens during application program execution, it is extremely important to be able to quickly locate the point of failure. In particular, if an exception is not logged, once a failure occurs, additional logging must be added, the application restarted, and the problem must be reproduced. Restarting such 24x7 applications is highly undesirable. Most of the J2EE API methods whose execution can

result in failures are designed to throw exceptions. Examples of such methods are those that result in interactions with other components of the application architecture, for example, database, network, LDAP etc. The most desirable programming practice that can help in tracing failure points is to catch the specific exceptions thrown by these methods and output some kind of log information indicating the failure. Logging the failure this way helps in understanding what happened during program execution later on. However, in the large J2EE applications we have worked with, we noticed that this exception handling practice is not common.

Before proceeding to talk about exception usage patterns in these applications, we give a brief overview of the exception mechanism in Java (Section 2). This section is not intended to be a tutorial of Java exceptions. The reader is advised to consult the Java language specification and the Java Virtual Machine specification for details on language and implementation semantics of Java exceptions. Section 3 discusses exceptions usage patterns in a handful of large J2EE applications we have worked with. This section is the primary contribution of this paper. In Section 4, we discuss approaches to solve this problem of improper exception handling. Section 5 discusses related work in the area of understanding exceptions usage in Java applications.

2 Overview of Java Exceptions

An exception can occur under one the following circumstances [11]:

- An abnormal execution condition was synchronously detected by the JVM. For example, integer divide by zero, array out of bounds, out of memory error, loading or linking errors.
- A **throw** statement was executed.
- An asynchronous exception occurred because the **stop** method of class Thread was invoked or an internal JVM error occurred.

The Java programming language defines class Throwable and allows the application programmer to extend this class. The Throwable class and its subclasses are collectively referred to as exception classes and instances of these classes are used to represent exceptions. Among these exceptions, all exceptions that are instances of class RunTimeException and its subclasses and exceptions that are instances of class Error and its subclasses need not be checked, i.e., these exceptions need not be explicitly handled by the application. All other exceptions are *checked exceptions* and need to be explicitly handled in the program. The language provides the `try` and `catch` clauses to define exception handlers. During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception. If no such handler is found, then the method uncaughtException is invoked for the

ThreadGroup that is the parent of the current thread-thus every effort is made to avoid letting an exception go unhandled. [11]. The following is a simple try – catch combination:

```
// in the main method:
try() {
       foo();
} catch (MyException me) {
       System.err.println(me);
} finally {
    clearResources();  // code to close resources.
}
```

The method `foo()` or one of its callees can potentially throw an exception of type `MyException` (or a subclass of `MyException`). `MyException`, in this case, is a checked Java exception. When such an exception is thrown, control is transferred from the call site of `foo()` to the beginning of the catch block and execution proceeds from that point on. When a method such as `foo()` throws an exception, the signature of the method must advertise the specific exception(s) thrown. What happens inside the `catch` clause is still up to the application programmer. It is possible that the exception is rethrown using a `throw` statement, or the exception is logged or some code is executed or nothing at all happens. There is also the `finally` clause to a `try` statement. The semantics of the `finally` block is that it is always executed. Whether a catch clause executes or not, the code within a `finally` is always executed. In the above code snippet, the method clearResources() executes both when the program exhibits normal and exceptional control flow. The Java API itself defines a number of checked and unchecked exceptions.

3. Exception usage in Large Java Applications

From our experience working with large J2EE applications, we have observed the following exception usage patterns that have hindered the maintainability and serviceability of these applications. All these applications are “real-life” i.e., customer applications that have been deployed and in production.

3.1 Swallowed Exceptions

Exceptions should not be ignored through empty catch blocks. In general, every path out of a `catch{}` block should result in the exception being logged or the exception being re-thrown or have some kind of *remedial code* that remedies the exceptional execution behavior. If a handler block has neither logging code nor a rethrow, we refer to the corresponding exception as *swallowed.* The following is an example of a swallowed exception:

```
// example swallowed exception
```

```
try{
    foo();
} catch (MyException me) {
}
```

where, there is no rethrow of an exception or some kind of logging code to record that the exception happened within the catch block. In one very large e-business customer application, the application code had ignored many exceptions, i.e., contained empty catch blocks without any logging information. Consequently, when such a system failed in production, it was extremely hard to track the cause of failures. Ignored exceptions severely impaired the effectiveness of monitoring systems in this application environment. For example, the application had a try block that had an SQL update in it. The corresponding catch was empty. [An SQL update (`executeUpdate()`) method call is part of the J2EE API to access the database component of the application. The method call is usually implemented in a driver that interfaces with C code that exchanges information with the Database system, typically via sockets.] In this case, if the update failed, there will be no record of it for the application server.

However, under certain circumstances, a catch block without any logging code or rethrow is perfectly fine. These are usually cases where exception constructs are used to manage normal program control flow, or certain exceptions caught need not be logged, e.g., `InterruptedException`, or the `catch` block contains *remedial* code. Listed below are a few examples:

```
1. // example OK swallowed exception case
     key = null;
     try {
        key = foo();
     } catch (MyException me) { return key; }
     return key;
```

In the above example, the value of variable `key` is set to null before the `try` statement. When `foo()` throws an exception, the catch block executes but does not have to set the value of `key` to be `null` again. The exception thrown appears to be swallowed because of the absence of logging code or a throw statement within the catch block. Interestingly, in this example, if the programmer did not intend to use the try-catch for normal control flow, the calling method will likely see a `NullPointerException` that is not caught causing debugging nightmares.

```
2. // example OK swallowed exception
   try {
      key = foo();
   } catch(MyException me) {key = ValMaybeNull; }
   return key;
```

In this example, a semantically valid assignment to the variable `key` occurs within the catch block which can potentially execute as normal program control flow. A variation of this example is when `key` is initialized to `null` prior to the try block and the handler has no code in it.

```
3. // example OK swallowed exception – remedial code
     sentFlag = false;
     while(!sentFlag){
      for(int i=0; i<connectRetry && !sentFlag; i++){
           try {
                  send_something();
                  if (success) sentFlag = true;
           } catch (Exception ex){
                  Thread.currentThread().sleep(SLEEP_TIME);
                  retryCount=i; // used to track #failures
           }
      }
     }
```

In the above example, taken from a real J2EE application, whenever a failure occurs during the send, i.e., in method `send_something()`, an exception is thrown. The `while` loop iterates until the method execution succeeds. This is an example where the exception causes remedial code to be executed.

3.2 Single catch block for multiple exceptions

If exceptions are caught in the same block, it should be possible to identify which exception was handled by the exception handler by the logging. However, several times, we have encountered the following (undesirable) code in these applications:

```
// exceptions are not handled individually
try {
   foo();
   bar();
} catch (Exception e) {
   System.out.println("catching exception" + e);
}
```

The above example also points out the case where exceptions are *subsumed*. The following is preferred for debugging, where `foo()` can throw the specific exception `MyException` and `bar()` can throw the exception `MyException1`.

```
try {
  foo();
  bar();
} catch (MyException me) {
  System.out.println("MyException raised " + me);
} catch (MyException1 me1) {
  System.out.println("MyException1 raised " + me1);
}
```

Within a try block, exceptions of the same type should not be raised at multiple program points. If not, it will be difficult to identify within the catch block which program point (call site) raised the exception resulting in debugging difficulties. For example,

```
// multiple program points raising same exception
try{
   foo();
   bar();
   Object obj = baz();
} catch (MyException me1) {
    System.out.println("me1 raised by bar");
} catch (MyException me) {
    System.out.println("me raised by baz or foo");
}
```

where both methods `foo()` and `baz()` can throw an exception of type `MyException`.

Likewise, if an exception is being re-thrown, the application should avoid mapping multiple exceptions to the same exception since this hides problem sources from debuggers.

3.3 Exceptions not handled at appropriate level

Another coding pattern that makes debugging difficult is when exceptions are not handled close to the source of the exception. If exceptions are propagated a long way up the call chain, the error message and handling will become less meaningful and debugging much more difficult.

3.4 Log verbosity in catch blocks

A common coding style in a handler that can do as much harm as good is:

```
log("some exception happened");
e.printStackTrace();
```

For example, consider a system under a load surge – some resource in the system becomes overloaded, and *temporarily* fails, resulting in several exceptions getting thrown. Normally, this would just affect the requesting users, but if the exception handling is overly heavy (e.g. lots of I/O, and getting the stack trace), it just adds more load to the system, causing a cascading failure that feeds on itself.

3.5 Application Statistics

The table below shows some statistics on a handful of J2EE applications that we analyzed. The results show the number of swallowed exceptions as defined in this section, after filtering out cases that have `return` statements in the catch blocks and catch blocks that do not have to log exception information. The applications A1-A5 listed in this table are all J2EE customer applications and hence the names are hidden. The application PetStore is a J2EE sample application published by Sun Microsystems. The #classes, #methods and #handlers columns report the numbers in just the application code, not including the J2EE and J2SE libraries. The last row shows the results on JDK1.3.1 rt.jar J2SE library classes.

Application	# classes	#methods	#handlers	#swallowed	#after filtering
A1	1724	19387	9951	182	162
A2	781	8509	492	74	35
A3	666	9355	16770	80	47
A4	1151	10458	3373	97	27
A5	2202	30964	16215	444	350
PetStore	353	2001	428	22	11
rt.jar	5484	46723	3670	974	723

The following table gives additional results on false positives for some of the applications, i.e., what goes on inside the handlers for some of the applications:

Application	#handlers	# handlers with calls	#handlers with re-throws	#handlers with loads/stores
A1	9951	9363	0	27
A4	3373	3119	0	18
A5	16215	15487	0	54
PetStore	428	372	0	1
rt.jar	3670	2242	0	77

Note that a majority of the handlers had calls in them and in the applications we analyzed, we did not come across exception re-throws. When we examined application A1 in more detail and looked at the calls made inside the handlers, we noticed that a majority of these calls were not to logging code, but calls to application code doing business logic of some kind.

4. Approaches to handle the problem

How can exception handling be made more effective? Most of the J2EE API methods require the programmer writing these applications to enclose the methods in try – catch blocks. It appears the kinds of improper exception handling that we discussed in the previous section happens due to lack of rigor in writing these applications. For example, consider an application method that (1) gets a database connection using the `javax.sql.DataSource.getConnection()` call, (2) creates one or more database SQL statements using `java.sql.Connection.createStatement()` (3) executes the SQL statements created (that may be updates or queries to the database) using the `java.sql.Statement.executeUpdate()` or `java.sql.Statement.executeQuery()` methods and, (4) finally processes the results from the database using the `java.sql.ResultSet` interface methods. Almost all of these methods throw `SQLException`. It is tempting for the application developer (who does not practice rigor) to either (a) enclose all these methods within a single `try – catch` block that catches an `SQLException` or, (b) enclose each of the

above method calls in a `try – catch` block that catches an `SQLException`, but the handlers do not log any information. We have encountered both these coding patterns in the applications we have worked with. As mentioned in the previous section, both these exception coding patterns make debugging an exceptional condition during program execution extremely tedious. A log of which of the database operations caused the exception to occur will be extremely helpful in debugging not only the application on the Java side, but also any database errors.

Several approaches are possible to handle this "bad coding practice" problem:

- The programming environment (an integrated development environment, IDE) automatically inserts try – catch blocks for method calls whose signature has the `throws` clause in it. In addition, such an IDE could also insert a default print method call that logs some (minimal) information about the exception being caught by the handler. This approach saves a lot of trouble on the programmer's side and also reminds the programmer to log exceptions. However, a drawback of this approach is that, in cases where the try – catch block is used to capture normal control flow (see examples in Section 2), the programmer has to explicitly undo some of the operations of the IDE. While one could argue that try – catch – finally is used for normal control flow only rarely, when actually used, undoing the work of the IDE can be annoying to the developer.
- Another approach is to statically analyze the application program and point out program points in the application where exception handling has not been implemented properly. For example, check the application for swallowed exceptions, multiple exceptions handled by a single catch block, catch block not catching the exact exception thrown but it's supertype etc. Such an analyzer can be integrated as part of an IDE that checks for bad coding patterns. This is the approach that we have used in our tool called *SABER*. SABER does static program analysis (control and data flow) to check for many bad coding patterns, including swallowed exceptions and handlers catching supertype exceptions. The tool is integrated into the WebSphere Studio development environment and reports messages to the programmer in a manner within this IDE that links the error message to the program point where the bad coding pattern appears.
- Finally, is it possible for the JVM to provide more information? Typically, when an exception is thrown, the JVM dumps a stack trace. However, if the exception is rethrown within the catch block, the stack trace will include method calls only from the point where the exception was actually caught. Another factor prohibiting debugging of exceptions is when the JIT is on. Most JVMs do not provide line number information in the stack trace when the method has been JIT-compiled. The optimizing compiler should keep track of this information and convey the line number of the method invoked that caused the exception to occur. This approach will still not be able to provide any other logging information other than the stack trace, which is not always sufficient in debugging the problem.

5. Related Work

Several research papers have looked into proper handling of exceptions and have studied the control flow aspects of exceptions. Robillard and Murphy [5][6] describe their tool called Jex that can be used to illustrate to the programmer the structure of exceptions in application code. Based on the exception control flow, the programmer will be able to identify program points where exceptions are caught accidentally, error handling procedures are not being followed or finer-grained recovery code can be added in the program. Jex analyzes exception control flow and identifies exception subsumption, i.e., wherever a precise exception is not raised, and unhandled exceptions. By presenting the resulting information to the application programmer, the tool allows the developer to encode handlers for exception types that are missing, there by increasing the robustness of the code. Our work is along the same lines as Robillard and Murphy's and, we look at a wider range of exception usage issues *including* subsumed and unhandled exceptions within method bodies. We also present results of exception usage on large real-life applications that are typically developed by multiple development organizations and hence exhibit varying coding styles and conventions.

Ryder *et al* [10] describe another static analysis tool, JESP, for examining the usage of exceptions. The paper provides empirical results of exception usage on Java benchmarks and discusses the implications of the results on compiler optimizations. We have observed that not all the empirical results (#exception constructs (try, catch, finally), distance between the throw and the corresponding catch, prevalence of user-defined and Java-defined exceptions, #exception classes and the shape of the exception hierarchy) apply to larger Java applications that we have analyzed.

A number of other papers talk about control flow representations of programs written in languages that support exceptions: the Marmot compiler [9], and Choi *et al* [4] for Java, the Vortex compiler [3] that supports Modula-3, Java and the Cecil languages, Chatterjee *et al* [8] that talks about modeling exceptions in an interprocedural control flow graph. Another paper that raises issues related to flow analysis of Java programs in presence of exceptions talks about instruction scheduling in presence of these constructs [7].

Stevens [1,2] studies exception control flow in Java programs and the negative impact of this type of control flow on compiler driven optimizations. He also discusses approaches to reducing this effect using static, whole program analysis on the byte code representation of Java programs.

Romanovsky and Sander [13] talk about misusing exception handling in Ada programs that have a lot in common with how exceptions are misused in Java.

Miller and Tripathi [14] discuss how object-oriented techniques interact with exception handling and which OO features conflict with current exception handling mechanisms.

References

1. Andrew Stevens, Des Watson. The Effect of Java Exceptions on Code Optimisation.

In *European Conference on Object Oriented Programming 2000: Exception Handling in Object Oriented Systems. Workshop Talk*

2. Andrew Stevens *JeX: An Implementation of a Java Exception Analysis Framework to Exploit Potential Optimisations,* University of Sussex, 2001.
3. Jeffrey Dean, Greg DeFouw, David Grove, Vass Litvinov and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. *In ACM Conf. On Object Oriented Programming Systems, Languages and Applications*, pages 83-100, October 1996.
4. Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE99),* Sep 1999.
5. Martin P. Robillard and Gail C. Murphy. Analyzing Exception Flow in Java Programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (Toulouse, France)*, volume 1687 of *Lecture Notes in Comp. Sci.*, pages 322-337. Springer-Verlag, Sep 1999.
6. Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *Proceedings of the ACM SIGSOFT Eight International Symposium on the Foundations of Software Engineering (FSE-8): Foundations of Software Engineering for Twenty-First Century Applications* (San Diego, California, USA), ACM Press, pages 2-10, November 2000.
7. Matthew Arnold, Michael Hsiao, Ulrich Kremer, and Barbara G. Ryder. Instruction scheduling in the presence of java's runtime exceptions. In *Proceedings of Workshop on Languages and Compilers for Parallel Computation (LCPC'99)*, August 1999. http://citeseer.nj.nec.com/arnold99instruction.html
8. Ramakrishna Chatterjee, Barbara Ryder, William A. Landi. Complexity of concrete type inference in the presence of exceptions. In *Lecture Notes in Computer Science*, 1381, pages 57-74. Springer-Verlag, April 1998. *Proceedings of the European Symposium on Programming.*
9. Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgard and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.
10. Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon and Nirav Shah. A Static Study of Java Exceptions using JESP, *In Proceedings of CC 2000*, pp 67-81.
11. James Gosling, *Bill Joy, Guy Steele, Gilad Bracha. The Java Programming Language, Second Edition.* Addison-Wesley, 1997.
12. Tim Lindholm, Frank Yellin. *The Java Virtual Machine Specification. Second Edition.* Addison-Wesley, 1999.
13. A. Romanovsky and B. Sanden. Except for Exception Handling, *Ada Letters*, v. XXI, 3, pp. 19-25, 2001
14. Robert Miller and Anand Tripathi, Issues with Exception Handling in Object-Oriented Systems, *European Conference on Object-Oriented Programming, (ECOOP),* June 1997