Faster Implementation of Java Exceptions (*please!*)

R. Nigel Horspool
nigelh@csr.uvic.ca

University of Victoria, Canada

Premise

Exceptions with the try-catch control construct are useful for *much* more than recovering from a run-time error.

Typical uses would seem to be ...

- better debugging messages for Java code in development,
- returning error results (e.g. InputFileFormatError),
- as a wrapper for someone else's code, trapping exceptions at a place where they are understood.

But why not also use exceptions as a standard construct? There are other possible uses.

Use of Exceptions for Loop Exit?

- Earlier this year I presented a talk on automatically transforming loops to use exceptions to exit. For example,

```
Node n = firstElement;
int length = 0;

while(n != null) {
   n = n.nextElement;
   length++;
}
```

Original version of loop

1. Use of Exceptions for Loop Exit?

- Earlier this year I presented a talk on automatically transforming loops to use exceptions to exit. For example,

Transformed version of loop

```
Node n = firstElement;
int length = 0;
try {
   for( ; ; ) {
      n = n.nextElement;
      length++;
   }
}
catch(NullPointerException e) {
}
```

and why not?
(When the list is long enough, this code is faster than the conventional loop interpretively executed by the JVM. If compiled, there should be no difference.)

2. Exiting from Deep Recursion

- Backtracking search algorithms

  e.g., when we are many levels deep and have found the desired answer, execute this to return to the top level:

```
throw new SolutionFound( ... );
```

2. Exiting from Deep Recursion

- Backtracking search algorithms

  e.g., when we are many levels deep and have found the desired answer, execute this to return to the top level:

```
throw new SolutionFound( ... );
```

- Recovery from syntax errors in recursive descent parsing

  e.g., return from expression level parsing to statement level parsing by executing this:

```
throw new BadExpression("missing operator");
```

3. As a Switch on Class Type?

- This may pervert the purpose of exceptions but the throw-catch construct can be used as as a switch on class type.

```
if (n instanceof BinaryNode) {
   ...
} else if (n instanceof EmptyNode) {
   ...
} else if (n instanceof LeafNode) {
   ...
}
```

becomes:

```
try {
   throw n;
} catch(BinaryNode b) {
   ...
} catch(EmptyNode e) {
   ...
} catch(LeafNode l) {
   ...
}
```

4. Other Uses of Exceptions?

Here is a common idiom

```
if (a instanceof ClassB) {
   b = (ClassB)a;
}
```

It is interesting because the cast implicitly re-performs an *instanceof* test.
Maybe rewrite as this?

```
try {
   b = (ClassB)a;
}
catch(ClassCastException e) {
}
```

and probably there are other uses for exceptions.

Why Don't We Use Exceptions in These Ways?

Apart from the belief that exceptions are only provided for handling errors, there is also the difficulty that an exception takes a *very long time* to process.

(The example loop to find the length of a list requires more than 800,000 iterations to pay off with Sun's JVM, even though the number of bytecode instructions in the loop is reduced from 6 to 5.)

How are Java Exceptions Implemented in the JVM?

- Each class file contains an *Exception Table*.

from	to	target	type
22	46	89	`NullPointerException`
37	42	104	`IOException`

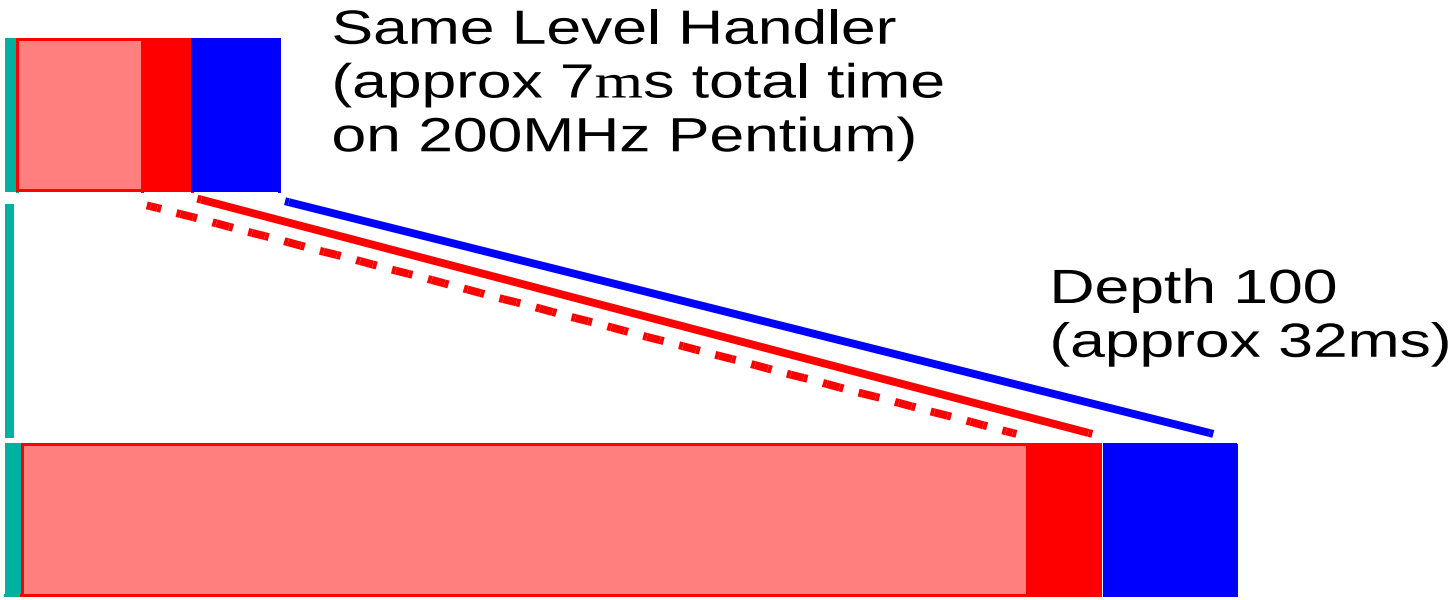
How are Java Exceptions Implemented in the JVM?

- Each class file contains an *Exception Table*.

from	to	target	type
22	46	89	`NullPointerException`
37	42	104	`IOException`

- An exception or a throw statement causes an Exception object to be pushed; the Exception Table is searched for entries applicable to the bytecode line number; the first of the applicable entries with a ‘matching’ type is used; control is transferred to the target line in the bytecode.
- If no matching entry is found, the current frame is exited and the search repeated in the caller’s Exception Table.

And How The Time Is Spent ...



Note: We timed the simplest possible Exception Table; the search time will increase with size and nesting of **`try`**s.

Exception Object Creation – the fillInStackTrace Method

Most of the time is spent performing one routine – **`fillInStackTrace`** (a native method).

The time increases linearly with depth.

What can we do? Possibilities include:

1. Provide a *production mode* of execution where Exception objects are created without their stack trace fields being initialized.

2. Create the stack trace only if it might be used.

   Observations show that **`printStackTrace`** is hardly ever called for user-defined exception types (*see next slide*).

Exception Object Creation

About 50% of catch blocks do not use the Exception object for a Java run-time error.

	Java Errors		User-Defined	
	object used?	trace printed?	object used?	trace printed?
CaffeineMark 3.0	0/7	–	0/0	–
JFlex 1.3	5/14	2/14	2/5	0/5
RabbIT WebProxy 2.0	112/194	60/194	9/9	0/9
IceMail 2.5.1	55/114	20/114	3/3	0/3
JavaCC	90/189	7/189	128/129	0/129

Can we avoid creating the Exception object?

Suppression of Exception Object Creation

```
try {
   ...
}
catch( ExceptionType e ) {
   ...
   // Does e get referenced?
   // Could e.printStackTrace() get called?
   ...
}
```

After simple analysis, we can annotate the Exception Table with flags to indicate the need for an Exception object and to indicate the need for a stack trace.

Reducing Exception Object Overhead for User-Defined Exceptions

The user can eliminate almost all the cost in one of two ways ...

1. Overriding the **`fillInStackTrace`** method

```
public class MyExceptionType extends Exception {
   ...
   public Throwable fillInStackTrace() {
      return null;
   }
}
```

Reducing Exception Object Overhead for User-Defined Exceptions, cont'd

2. Recycling the same Exception object(s)

E.g., for the recursive descent parser example ...

```
static MyExceptionType e =
                  new MyExceptionType("syntax error",0);
...
if (some condition) {
   e.setMessage("parenthesis expected",lineNumber);
   throw e;
}
```

Faster Search of Exception Tables

The JVM search is a linear search, using the **`instanceOf`** test to compare exception types.

- In principle, each class's Exception Table could be implemented as a perfect hash table, so that

  ```
  target = searchTable( location, exceptionType );
  ```

  returns the result in constant time.

- Or, for each **`try`** range, we could precompute a table which can be indexed by the exception type ...

  ```
  target = lookupTable[exceptionType];
  ```

  and we also provide a data structure to map locations to table pointers.

Faster Search of Exception Tables, cont'd

- Current implementations of Java exceptions perform no work when entering/exiting each try block.

  We can push a pointer to a look-up table on entry to a try block and pop it on exit. Then use the tables like this ...

  ```
  target = top->lookupTable[exceptionType];
  ```

  This may be unpopular because it violates the 'user pay' principle.

- There is implicitly a stack of Exception Tables. The stack will grow *very* deep only in the presence of recursion. If this is a problem and if direct lookup tables are not implemented, then searching the stack of tables can be improved by using a cache.

Final Thought

There is a pervasive philosophy, both by the language designers and by the implementers, that exceptions are provided to handle only errors and that errors are rare events. Therefore the efficiency of implementation is unimportant.

This attitude ought to be overturned. Exceptions are simply too useful.

A Closing Quote

“Speed is not an issue here, since we are already fu.”

(A comment in Sun’s JVM exception handling code.)