;login:

THE MAGAZINE OF USENIX & SAGE

April 2001 • volume 26 • number 2

inside:

PROGRAMMING
Java Performance

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

<glenm@glenmccl.com>

The Cost of Exceptions

The Java language contains features for throwing and catching exceptions, where an exception is an abnormal condition such as an array index out of bounds or access through a null reference. Java exceptions are synchronous, that is, occur as the result of executing a particular instruction, and should be distinguished from UNIX-style signals delivered to a running process. A simple example of exception usage looks like this:

```
public class Simple {
      public static void main(String args[]) {
         int vec[] = new int[10];
         try {
            vec[15] = 37;
         }
         catch (ArrayIndexOutOfBoundsException e) {
            System.err.println(e);
         }
      }
}
```

In this program, an attempt is made to store a value at location 15 in a 10-long array. The attempt is made within a try block that specifies a catch clause for exceptions of type ArrayIndexOutOfBoundsException. If an exception is thrown, either explicitly via a throw statement, or implicitly by the run time system (as in this example), then the catch clauses are tried to see if a matching one can be found, and if so, control is transferred to the clause.

What is the cost of exception handling? Does it have a significant impact on the speed of your Java programs? In this column we'll look at a few examples of exception usage, and make some observations about the costs you should expect. As always, your results may vary from those described here, depending on the hardware you use and the particular Java Virtual Machine (JVM) you have. The JVM used for these examples is Javasoft's Hotspot 1.3.0-C version.

Overhead When Exceptions Are Not Used

How big a price do you pay for the exception feature when no exceptions are thrown? To find out, we'll use an example consisting of an inefficient sort algorithm, with a try block added to the inner loop:

```
public class Sort {
      static void sort(Object vec[]) {
         int len = vec.length;

         for (int i = 0; i < len - 1; i++) {
            for (int j = i + 1; j < len; j++) {

               // set up a try/catch block
               // to handle bad String casts

               try {
                  String si = (String)vec[i];
                  String sj = (String)vec[j];
                     int c = si.compareTo(sj);
                     if (c < 0) {
                        Object t = vec[i];
                        vec[i] = vec[j];
                        vec[j] = t;
```

```
                    }
                }
                catch (ClassCastException e) {
                    System.err.println(e);
                    System.exit(1);
                }
            }
        }
    }

    public static void main(String args[]) {
        final int N = 10000;

        // create an array of object references

        Object vec[] = new Object[N];

        // populate array with String objects
        // of the form "abc1234"

        for (int i = 0; i < N; i++)
            vec[i] = "abc" + i;

        System.out.println(vec[0] + " " + vec[1] + " " + vec[2]);

        // sort

        long start = System.currentTimeMillis();
        sort(vec);
        long elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed);

        System.out.println(vec[0] + " " + vec[1] + " " + vec[2]);
    }
}
```

In this example, an array of Object references is sorted, with an assumption made that the Object references actually refer to String objects. Before vec[i] and vec[j] are compared, they are cast from Object to String. It's possible that the assumption will be false, in which case a ClassCastException is thrown.

With the try block in place, the program uses 16218 units of time, and without the block, 16516 units, a negligible difference. So for this particular example and JVM, there's no cost associated with the try block.

Recycling Exception Objects

Let's get a little deeper into our investigation, and look at another example, one where an exception is repeatedly created and thrown:

```
public class Reuse {
    public static void main(String args[]) {
        final int N = 500000;

        // create an exception object

        Throwable exc = new Throwable();

        long start = System.currentTimeMillis();
        for (int i = 1; i <= N; i++) {
            try {
                // either throw a new'ed object,
                // or a previously created object,
                // or a previous created object
                // with its stack trace filled in

                throw new Throwable();

                //throw exc;

                //exc.fillInStackTrace();
                //throw exc;
            }
            catch (Throwable e) {
            }
        }
        long elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed);
    }
}
```

This particular program requires about 3000 milliseconds to throw/catch 500,000 exceptions on a fast (800 MHz) machine.

We might wonder where the time is going. To find out, we can amend the example slightly. The first thing we do is repeatedly throw an existing exception object, instead of new-ing an object each time. When we do this, the time goes from 3000 down to 200 milliseconds. We might conclude from this that there's a lot of overhead in creating new objects.

This is so, but perhaps in a different way than we might think. When an exception object is created, an internal method fillInStackTrace() is called to record within the object details of the current state of stack frames for the current thread. We can try a third variation on the example to capture the time required for recording the stack trace. When an existing exception object is repeatedly thrown, but fillInStackTrace() is first called, the time goes from 200 to 2700 units. In other words, recycling exception objects does save you a lot of time, but at the expense of accurate stack trace information. The speedup may not be worth it.

Processing Many Catch Clauses

Another possible efficiency issue comes up when you have many catch clauses in a try block. The run time system must go through the clauses to find a matching one. An example of this situation looks like this:

```
class E1 extends Throwable {}
class E2 extends Throwable {}
class E3 extends Throwable {}
class E4 extends Throwable {}
class E5 extends Throwable {}
class E6 extends Throwable {}
class E7 extends Throwable {}
class E8 extends Throwable {}
class E9 extends Throwable {}
class E10 extends Throwable {}

public class Many {
    public static void main(String args[]) {
        final int N = 10000000;
```

```
            // create an exception of type E10

            //Throwable exc = new E1();
            Throwable exc = new E10();

            long start = System.currentTimeMillis();
            for (int i = 1; i <= N; i++) {

                // throw an exception and have it
                // caught by the E10 catch clause

                try {
                    throw exc;
                }
                catch (E1 e1) {
                }
                catch (E2 e2) {
                }
                catch (E3 e3) {
                }
                catch (E4 e4) {
                }
                catch (E5 e5) {
                }
                catch (E6 e6) {
                }
                catch (E7 e7) {
                }
                catch (E8 e8) {
                }
                catch (E9 e9) {
                }
                catch (E10 e10) {
                }
                catch (Throwable e) {
                }
            }

            long elapsed = System.currentTimeMillis() - start;
            System.out.println(elapsed);
        }
    }
```

If an exception of type E1 is thrown, it will match the first clause, while if an E10 exception is thrown, it will match a clause near the end of the list.

If the program throws an E10 exception, the time is 3578 units, while for an E1 exception, the time is 3407, a difference of 5%. So there's a difference here, but not a very large one.

Avoiding Exceptions

Sometimes there are cases where you can avoid exceptions altogether, and save time in doing so. Suppose you need to test whether an object is of String type. There are a couple of ways of doing so. One is to try to cast the object to a String, inside of a try block, and catch the ClassCastException that may be thrown. Another approach uses the "instanceof" operator instead of exceptions.

These two approaches can be programmed as follows:

```
public class Cast {

    // see if Object is a String by trying to cast it

    static boolean isString1(Object obj) {
        try {
            String s = (String)obj;
            return true;
        }
        catch (ClassCastException e) {
            return false;
        }
    }

    // see if Object is a String by use of the instanceof operator

    static boolean isString2(Object obj) {
        return obj instanceof String;
    }

    public static void main(String args[]) {
        final int SCALE = 100;
        final int N1 = 1000000;
        final int N2 = N1 * SCALE;
        boolean b;
        Object obj = new Object();

        // repeatedly call isString1()

        long start = System.currentTimeMillis();
        for (int i = 1; i <= N1; i++) {
            b = isString1(obj);
        }
        long elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed);

        // repeatedly call isString2()

        start = System.currentTimeMillis();
        for (int i = 1; i <= N2; i++) {
            b = isString2(obj);
        }
        elapsed = System.currentTimeMillis() - start;
        System.out.println(elapsed / SCALE);
    }
}
```

Some care needs to be taken in measuring the elapsed times, because the second approach using instanceof runs about 1000 times faster than the first approach that uses exceptions. In this particular example, there's no virtue in using exceptions, because the same problem can be solved in a simpler and faster way.

Summary

We've looked at several examples of performance issues with exception handling, and illustrated some techniques for optimizing performance. As always, it pays to be careful with performance tuning. It's usually best to write code in a natural, clear style, and not to worry too much about squeezing out the last little bit of performance. But when you're desperate to speed up a program, it's good to know where the time is going.