

## Put Java in the fast lane

Follow a performance improvement process to locate bottlenecks in your application

### Summary

This article presents some techniques for locating performance bottlenecks in Java applications and offers suggestions for improving Java performance. Along the way, you'll look at some of the classes in the new `java.nio` package. (August 2, 2002)

By Greg Holling

In my consulting practice, one of the most common complaints I hear about Java is that it's "too slow," "too resource intensive," or that "performance may be a problem." These statements are often made without any basis in fact; many assume that Java's poor performance is a given.

Of course, Java programs *can* be slow, memory intensive, and painful to use; but so can programs written in C++, Visual Basic, Smalltalk, Pascal, Ada, or C#. That doesn't necessarily mean that the language or the runtime environment is bad; it might just mean that a developer wrote the code without considering performance side effects. A well-written Java program usually performs just as well as a program written in any other programming language, and it often performs better, especially with the performance improvements in JDK 1.3 and 1.4.

One important note: *Perceived* performance is the critical measure. It doesn't matter how fast, efficient, or elegant your code is if it doesn't appear to be running or displaying fast enough. Likewise, ugly code can sometimes be blazingly fast—but a maintenance nightmare. We should really strive for code that combines the best of both worlds: it's elegant and performs well enough to satisfy our clients.

To move you a step closer to that goal, in this article, I walk you through a performance improvement process.

To illustrate the process, I include code that uses classes from the new `java.nio` package to provide you with additional clues about how to use them more effectively.

### Performance improvement process

Suppose you've written or inherited a Java application that just isn't performing up to snuff. Your boss is breathing down your neck, and the delivery deadline is next week. What do you do?

I'm assuming you don't want to take the "run-away-as fast-as-you-can" approach. You want to fix the performance problems. Where do you start?

I recommend the following performance improvement process:

1. Decide what performance level is "good enough"
2. Test on all your target platforms
3. If performance is good enough on all target platforms, **stop**; do not pass go, do not collect \$200
4. Profile your application to find the bottlenecks
5. Re-architect or rewrite your code to fix the bottlenecks
6. Return to Step 2

To illustrate the process, let's look at an example. Along the way, I'll fill in more details on the individual steps. The examples include mostly AWT (Abstract Windowing Toolkit) graphics, because performance improvements in graphical programs are easier to see and more interesting to write.

### Apply the process

The following program reads a series of files and counts the occurrences of the letters a to z in the files. It displays a histogram of the letter frequency; the histogram updates after each file is read.

Examine the code below and note to yourself where you could improve it. Then follow along to see if your intuition was correct:

```

import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Frame;
import java.io.IOException;
import java.io.FileInputStream;

public class Letters extends Component {
    long[] countArray = new long[26];
    static char[] letterArray =
    {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
    'p','q','r','s','t','u','v','w','x','y','z'};
    /**
     * Find the number of occurrences of each letter of
the
     * alphabet in the named file. The result is
returned
     * as a 26-element array of long elements.
     * Of course, this will only work for the English
alphabet.
     */
    void countCharacters (String filename)
throws IOException {
        System.out.println ("...reading " + filename);
        FileInputStream fis =
            new FileInputStream (filename);
        int tmp;
        while ((tmp = fis.read()) != -1) {
            char c = Character.toLowerCase((char)tmp);
            int pos = c - 'a';
            if ((pos >= 0) && (pos <= 25)) {
                ++countArray[pos];
            }
        }
        fis.close();
    }
    /**
     * Draw a histogram of the letter frequency.
     * This method is triggered by repaint(), or by
     * window manager repaint events.
     */
    public void paint (Graphics g) {
        long maxCount = 0;
        for (int i=0; i<countArray.length; ++i) {
            if (countArray[i] > maxCount) maxCount =
countArray[i];
        }

        Dimension d = getSize();
        double yScale = ((double)d.height) /
((double)maxCount);
        int barWidth = (int)d.width / countArray.length;
        int x = 0;
        for (int j=0; j<countArray.length; ++j) {
            g.setColor (Color.blue);
            int barHeight = (int)(countArray[j]*yScale);
            g.fillRect (x, d.height-barHeight,
                barWidth, barHeight);
            g.setColor (Color.white);
            g.drawRect (x, d.height-barHeight,
                barWidth, barHeight);
        }
    }
}

```

```

        g.setColor (Color.black);
        g.drawChars (letterArray, j, 1, x, 10);
        x += barWidth;
    }
}
/**
 * The main method. Files to analyze are specified
 * on the command line; their names are in argv.
 */
public static void main (String[] argv)
throws IOException {
    Letters letters = new Letters();
    Frame f = new Frame ("Letter Count");
    f.add (letters);
    f.setSize (300, 200);
    f.setVisible (true);

    long startTime = System.currentTimeMillis();
    for (int i=0; i<argv.length; ++i) {
        letters.countCharacters (argv[i]);
        letters.repaint();
    }
    long endTime = System.currentTimeMillis();
    System.out.println ();
    System.out.println
        ("Elapsed time: " + (endTime - startTime)
+ " msec");
    System.out.println();

    for (int i=0; i<letters.countArray.length; ++i) {
        System.out.print
            (letterArray[i] + "=" +
letters.countArray[i] + ",");
    }
    System.out.println();
}
}

```

What's wrong with this program, and how do we fix it? Let's follow our improvement process.

### Step 1: Decide what performance level is good enough

I'll arbitrarily say that performance is good enough when the program can read and process all HTML documents from package javax.swing in about 10 seconds. In the real world, external, and often political, factors often determine performance targets, which might still be arbitrary, of course.

### Step 2: Test on all your target platforms

I have two workstations handy and run JDK 1.4 on both: a fast Pentium running Windows (Platform A), and a slower one running Linux (Platform B). Testing on these two platforms shows that Platform A reads and processes the HTML documents in about 140 seconds; Platform B does the same in 153 seconds. Ouch. Not nearly good enough.

### Step 3: If performance is good enough, stop

We didn't meet the performance targets, so we continue with optimization.

### Step 4: Profile your application

You can profile your application any number of ways. Many

profilers have GUI (graphical user interface) front ends; prices range from free to thousands of dollars. See *Resources* for a selection. Sun Microsystems' JDK includes the hprof tool for profiling, which I'll use to illustrate the profiling process.

The hprof profiler allows you to track many interesting things about an application, including object creation statistics and method profiles. I'll use method profiling for this example. Object creation information proves useful for locating memory leaks and garbage collection problems, which can offer another valuable exercise. To obtain a list of hprof options, type `java -Xrunhprof:help`. For this example's purposes, I used `java -Xrunhprof:cpu=samples,depth=15 Letters`. That generates an output file (`java.hprof.txt`) containing information about thread lifetime and an estimate of time spent in each method. The estimate generates at program exit by default; you can also receive an intermediate snapshot by typing `<ctrl-Break>` on Windows or `<ctrl-^>` on Unix.

Running the program generates the following output in `java.hprof.txt`:

```
CPU SAMPLES BEGIN (total = 3245) Sun Jul 14 18:18:03
2002
rank  self  accum   count trace method
   1  52.30% 52.30%   1697   25 sun.awt.win-
dows.WToolkit.eventLoop
   2  45.30% 97.60%   1470   41
java.io.FileInputStream.read
   3   0.52% 98.12%    17   45
java.awt.EventQueue.postEventPrivate
...
CPU SAMPLES END
```

The samples rank in order, starting with the methods used most often. You can usually focus on the first 5 to 10 entries. We'll ignore the first entry for now. We can find information about the second entry by looking for the string `TRACE 41` in the profiling output:

```
TRACE 41:
  java.io.FileInputStream.read(FileInputStream.java:Native
method)
  Letters.countCharacters(Letters.java:27)
  Letters.main(Letters.java:71)
```

Hmm...we're spending an awful lot of time calling `FileInputStream.read()` in `countCharacters()`. That seems like a good place to make a change.

If you find a suspicious method with method-profiling techniques, you have a few options:

1. Optimize the method—perhaps use a better algorithm
2. Call the method less often
3. Don't call the method at all

#### Step 5: Re-architect or rewrite your code to fix the bottlenecks

Looking at the code, you see that the input is unbuffered. Buffering could help quite a bit. Adding buffering is easy; just

use a `BufferedInputStream`:

```
...import java.io.BufferedInputStream;
...
void countCharacters (String filename)
throws IOException {
    BufferedInputStream bis =
        new BufferedInputStream(new
FileInputStream (filename));
    int tmp;
    while ((tmp = bis.read()) != -1) {
        char c = Character.toLowerCase((char)tmp);
        int pos = c - 'a';
        if ((pos >= 0) && (pos <= 25)) {
            ++countArray[pos];
        }
    }
    bis.close();
...
}
```

#### Step 6: Return to Step 2

Testing the modified code yields the following:

- Platform A: Times between 3.3 and 5.8 seconds
- Platform B: Times between 18.6 and 19.2 seconds

Not good enough, but better. Let's take another pass at optimization.

#### Continue optimizing code

At the above profiling output, the top entry originates from the GUI event loop. The event loop starts as soon as we make a GUI object visible (that is, call `setVisible(true)`), and the loop runs for the application's lifetime. Also notice that the application redraws the graph after every file is read; this proves more significant for large file numbers. Assuming we don't care about viewing the intermediate graph (that is, it isn't a firm requirement), we can speed up the application by moving the graphics display to `main()`'s end. Then the event loop won't start until the last minute in the application, and `paint()` will probably only be called a couple of times.

At this point, we might question whether the graphical display is required at all and whether it needs to run simultaneously with the application. Depending on the answer, we might be able to move the graphics to a different thread, a different application, or even a different machine. For this discussion's purposes, though, we'll assume immediate graphics display is required.

`main()`'s modified version looks like this:

```
public static void main (String[] argv)
throws IOException {
    Letters letters = new Letters();
    long startTime = System.currentTimeMillis();
    ...
    for (int i=0; i<letters.countArray.length; ++i) {
        System.out.print
(letterArray[i] + "=" + letters.countArray[i]
+ ",");
    }
    System.out.println();
}
```

```

    Frame f = new Frame ("Letter Count");
    f.add (letters);
    f.setSize (300, 200);
    f.setVisible (true);
}

```

After making the change, testing shows:

- Platform A: Times between 2.4 and 5 seconds
- Platform B: Times between 11.6 and 12.8 seconds

We've improved performance, but still haven't met our performance guidelines.

Additional profiling shows the following:

```

CPU SAMPLES BEGIN (total = 149) Sun Jul 14 19:53:50
2002
rank  self  accum  count trace method
  1 28.86% 28.86%    43   12
java.io.FileInputStream.readBytes
  2 16.78% 45.64%    25   29 sun.awt.win-
dows.WToolkit.init
  3 16.11% 61.74%    24   30 sun.awt.win-
dows.WToolkit.eventLoop

```

That qualitatively differs from what we saw before. Most of our time is spent in `readBytes()`. What can we do about `readBytes()`?

JDK 1.4 introduced a new set of classes in the `java.nio` package and its subpackages for buffering I/O (input/output). One class in particular looks useful: `MappedByteBuffer`. We can use a `MappedByteBuffer` object to efficiently represent the contents of a file in memory; the class manages all the details of buffering and memory management.

To connect the `MappedByteBuffer` object to the file, we use a `FileChannel` object. `FileChannel` represents a thread-safe connection between a buffer and a file that can read, write, map, and manipulate the file. The `FileChannel` object maintains information about the current position in the file, and also provides for low-level OS-specific optimizations. The JDK 1.4 documentation includes examples of how to use `FileChannel` and `MappedByteBuffer`.

We modify our code to use `FileChannel` and `MappedByteBuffer`:

```

...import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
...
    void countCharacters (String filename)
        throws IOException {
        FileInputStream fis = new FileInputStream(file-
name);
        FileChannel fc = fis.getChannel();

        // Get the file's size and then map it into mem-
ory
        int sz = (int)fc.size();
        MappedByteBuffer bb =
fc.map(FileChannel.MapMode.READ_ONLY, 0, sz);

```

```

        for (int i=0; i<sz; ++i) {
            char c =
Character.toLowerCase((char)bb.get());
            int pos = c - 'a';
            if ((pos >= 0) && (pos <= 25)) {
                ++countArray[pos];
            }
        }
        fc.close();
    }
}

```

...

The change produces the following timing results:

- Platform A: Times between 2.1 and 4.7 seconds
- Platform B: Times between 11 and 13.8 seconds

The results reveal a slight improvement over the previous example; however they might not prove significant from a statistical or perceptual view.

Okay, time to ask the question again: Have we reached our performance goals? Are the documents processing in about 10 seconds? We've come pretty close to that point; however, we'll try one more optimization.

Let's look at `FileChannel.map()`'s documentation, which has an interesting quote near the bottom of its description:

For most operating systems, mapping a file into memory is more expensive than reading or writing a few tens of kilobytes of data via the usual read and write methods. From the standpoint of performance it is generally only worth mapping relatively large files into memory.

The `ByteBuffer` class, which is `MappedByteBuffer`'s superclass, provides buffered input in the `java.nio` package. Let's try using a raw `ByteBuffer` instead of `MappedByteBuffer` to read the data and see what happens.

Two methods in the `ByteBuffer` class can obtain a raw `ByteBuffer`: methods `allocate(int)` and `allocateDirect(int)`. The argument for both methods is the buffer size. `allocateDirect()` creates a direct byte buffer, which performs as much native file I/O as possible. `allocate()` creates a nondirect byte buffer, which trades less native code for more deterministic (but probably slower) behavior. Sun recommends using direct byte buffers for long-lived buffers associated with large files.

Our code, modified to use a direct byte buffer:

```

...import java.nio.ByteBuffer;
...
    void countCharacters (String filename)
        throws IOException {
        FileInputStream fis = new FileInputStream(file-
name);
        FileChannel fc = fis.getChannel();

        // Get the file's size and then map it into mem-
ory
        int sz = (int)fc.size();
        int bufferSize = 1024;

```

```

        ByteBuffer bb = ByteBuffer.allocateDirect
(bufferSize);
        int nbytes = -1;
        while ((nbytes = fc.read (bb)) != -1) {
            bb.rewind();
            for (int i=0; i<nbytes; ++i) {
                char c =
Character.toLowerCase((char)bb.get());
                int pos = c - 'a';
                if ((pos >= 0) && (pos <= 25)) {
                    ++countArray[pos];
                }
            }
        }
        fc.close();
    }
}
...

```

Timing numbers for this code:

- Platform A: Times between 2.7 and 5.2 seconds
- Platform B: Times between 13.1 and 13.9 seconds

Substituting `allocate()` for `allocateDirect()` decreases performance by another 20 percent. Performance is a tad slower than the previous example, and the code isn't any easier to read. One interesting note, though: I tried the same code on a directory containing about 20 Java source files and received opposite results. Code written with the direct `ByteBuffer` performed about 20 percent faster than code written with `MappedByteBuffer`. This is probably related to extra overhead imposed by buffering and synchronization with native file I/O routines.

So...which optimization is the best? It depends on how the application will be used. According to the requirements listed at the article's beginning, I would pick the `MappedByteBuffer` version. It's the fastest for our test cases and will probably scale well.

The important question: Is performance good enough? It might have been good enough before we tried the last round of optimizations; if so, we just spent extra time optimizing for no apparent benefit. If performance still isn't satisfactory, the next optimization attempt could prove even more difficult and time-consuming.

The reason profiling is so important is that it gives you information about what to fix, and at the same time, gives you information about whether you're done. Sometimes, spending your time changing code will have no noticeable impact on performance and might actually make your code more difficult to read and maintain. Also, the HotSpot VM may surprise you with its optimizations—don't just assume you know how to make Java programs run faster.

The same improvement process will work for memory-related and I/O-related performance problems. Memory-related problems—memory leaks and garbage collection, for example—are often the most difficult to track, because they can prove more difficult to reproduce and observing the program might change memory-related behavior.

## Other ways to improve performance

Once profiling has determined the location of performance problems, deciding what to do can remain tricky. Here are some suggestions for fixing specific performance problems:

- **Method calls:** Optimize the method, or call it less often.
- **Class loading:** Preload classes, or use lazy (on-demand) instantiation. Note: In multithreaded programs, avoid double-checked locking (to find out why, see Brian Goetz's "Double-Checked Locking: Clever, but Broken" (*JavaWorld*, February 2001)).
- **Running out of memory:** Share objects rather than create them, possibly using an object factory or an object pool.
- **String manipulation:** Use `StringBuffer` or `char[]` rather than `String`.
- **Recursion:** Eliminate the recursive method call and convert it to an iterative one.
- **I/O and serialization:** Make sure I/O is buffered; consider writing your own I/O subclasses with unsynchronized methods.
- **Collection classes:** Use a collection class suited to your application. Arrays are sometimes faster to access and modify than collections.

One other approach to performance improvement: Use a byte code optimizer. These programs inspect a class file's byte codes, remove unused code, clean the method-call stack, and so on. The nice thing about optimizers is that you don't have to change any code—you just test to see if the optimizers work. The downside is that optimizers might produce non-portable code or code that doesn't work the way you expect.

## Give your programs a tune up

Perceived performance is more important than real performance. Before you start tuning, determine how to tell when performance is good enough. When tuning is necessary, test and profile. The new 1.4 I/O classes might provide some incremental performance improvement, but don't blindly assume that they will make a dramatic difference. Profile, profile, profile. Use your time effectively. Don't change code based on hunches and don't continue tuning after you've reached satisfactory performance.

### About the author

*Greg Holling is the president and senior consultant for Visionary Computer Consulting. He has more than 20 years of software development experience and has worked in a wide variety of roles, including programmer, system administrator, team leader, project manager, and architect. He has developed software for scientific, GIS, client/server, graphics, medical, insurance, and database applications.*

\*View this article in its entirety at: <http://www.javaworld.com/javaworld/jw-08-2002/jw-0802-performance.html>