

Robust Java benchmarking, Part 2: Statistics and solutions

Introducing a ready-to-run software benchmarking framework

Skill Level: Advanced

[Brent Boyer \(ellipticgroupinc@gmail.com\)](mailto:ellipticgroupinc@gmail.com)

Programmer
Elliptic Group, Inc.

24 Jun 2008

Program performance is always a concern, even in this era of high-performance hardware. This article, the second in a two-part series, covers the statistics of benchmarking and offers a framework you can use to benchmark Java™ code ranging from self-contained microbenchmarks to code that calls a full application.

[Part 1](#) of this two-part article guides you around the many pitfalls associated with benchmarking Java code. This second installment covers two distinct areas. First, it describes some elementary statistics that are useful for coping with the measurement variations inevitably seen in benchmarking. Second, it introduces a software framework for doing benchmarking and uses it in a series of worked examples to illustrate important points.

Statistics

It would be convenient if you only needed to perform one execution-time measurement and could use a single measurement to compare the performance of different code. Sadly, the popularity of this approach does not trump its invalidity — there are just too many sources of variation to let you trust a single measurement. In [Part 1](#), I mentioned clock resolution, complicated JVM behavior, and automatic resource reclamation as noise sources; these are just a few of the factors that can randomly or systematically bias benchmarks. Steps can be taken to try to mitigate some of them; if you understood them well enough, you might even be able to

perform *deconvolution* (see [Resources](#)). But these remedies are never perfect, so ultimately you must cope with the variations. The only way to do this is to take many measurements and use statistics to make reliable conclusions. Ignore this section at your own peril, for "He who refuses to do arithmetic is doomed to talk nonsense" (see [Resources](#)).

Supplementary material

A companion site for this article includes the full sample code package and a supplement that explores statistical issues in detail. See [Resources](#) for a link to the companion site.

I'll present just enough statistics to address these common performance questions:

- Does task A execute faster than task B?
- Is this conclusion reliable, or was it a measurement fluke?

If you make several execution-time measurements, the first statistic that you are likely to compute is a single number that summarizes its *typical value* (see [Resources](#) for links to Wikipedia's definitions of the statistical concepts in this article). The most common such measure is the *arithmetic mean*, commonly known as the *mean* or the *average*. It is the sum of all the measurements divided by the number of measurements:

$$\text{mean}_x = \text{Summation}_{i=1,n} (x_i) / n$$

A supplement to this article available on a companion Web site (see [Resources](#)) includes a discussion of other measures besides the mean; see its *Alternatives to the mean* section.

Using the mean of several measurements to quantify performance is certainly more accurate than using a single measurement, but it can be insufficient for determining which task executes faster. For example, suppose that the mean of task A's execution time is 1 millisecond and task B's mean is 1.1 milliseconds. Should you automatically conclude that task A is faster than task B? You might not if you also knew that task A's measurements ranged from 0.9 to 1.5 milliseconds, while task B's measurements ranged from 1.09 to 1.11 milliseconds. So, you also need to address the *measurement spread*.

The most common statistic for describing measurement spread (or scatter) is the *standard deviation*:

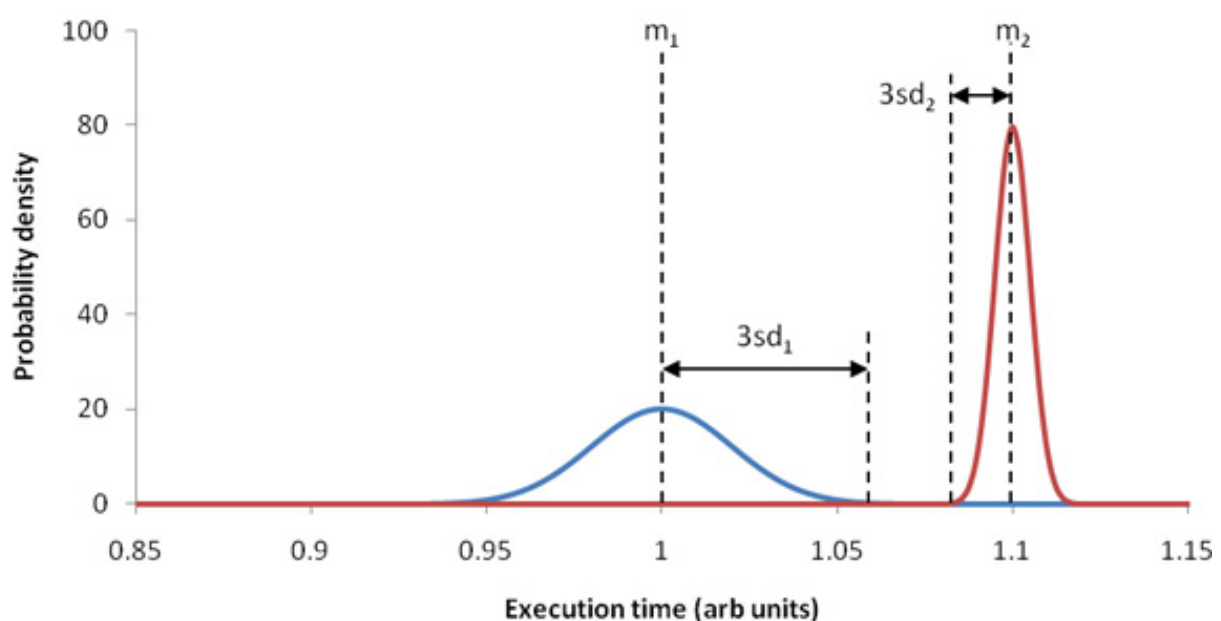
$$\text{sd}_x = \text{sqrt}\{ \text{Sum}_{i=1,n} ([x_i - \text{mean}_x]^2) / n \}$$

How does the standard deviation quantify measurement scatter? Well, it depends on what you know about the measurements' *probability density function* (PDF). The

stronger the assumptions you make, the finer the conclusions you can draw. The article supplement's *Relating standard deviation to measurement scatter* section explores this in more detail and concludes that in the benchmarking context, a reasonable rule of thumb is that *at least 95% of the measurements should lie within three standard deviations of the mean*.

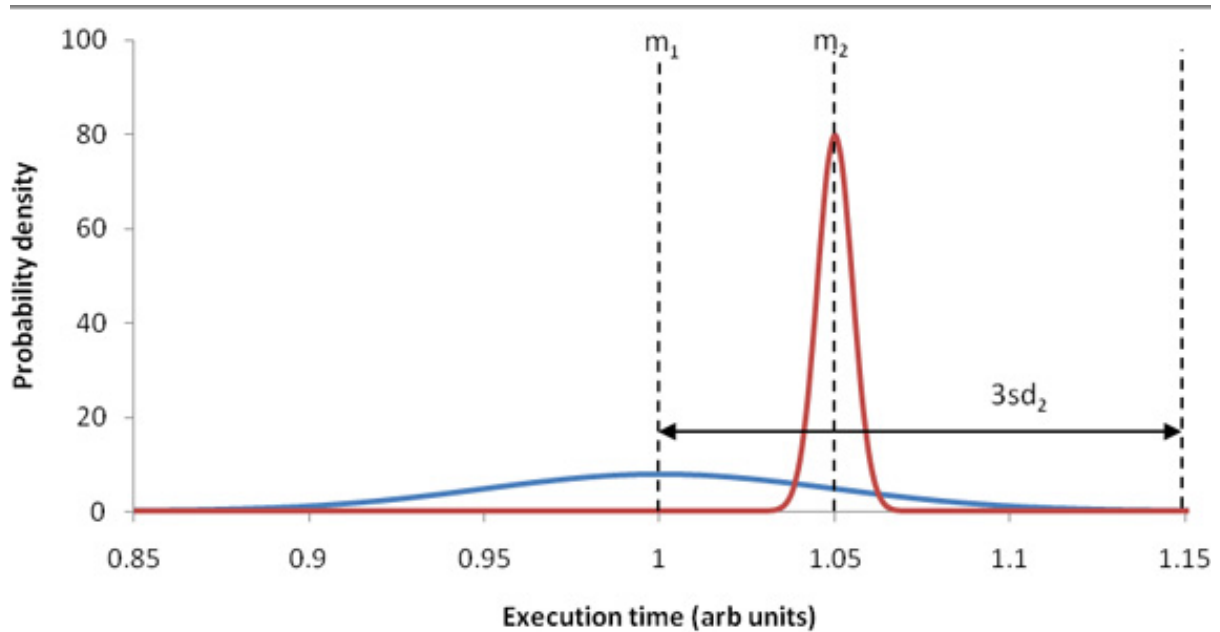
So, how can the mean and standard deviation be used to determine which of two tasks is faster? Invoking the above rule of thumb, the easy case is if the means of the two tasks are separated by more than three standard deviations (choose the larger standard deviation of the two). In that case, the task with the smaller mean is clearly faster almost all of the time, as shown in Figure 1:

Figure 1. Means greater than three standard deviations apart imply clearly distinguishable performance



Unfortunately, determinations become trickier the more two tasks overlap (for example, if the means are one standard deviation apart), as Figure 2 illustrates:

Figure 2. Means less than three standard deviations apart imply performance overlaps



Maybe all that you can do is rank two tasks by their means but note how much they overlap and point out the corresponding degree of weakness in your conclusions.

Reliability

The other question to address is how reliable these mean and standard deviation statistics themselves are. Clearly, they are computed from the measurements, so a new set of measurements would likely yield different values. Assume for now that the measurement process is valid. (Note: it may be impossible to measure the "true" standard deviation. See the *Standard deviation measurement issues* section in the article supplement, which explores this issue.) Then how different would the mean and standard deviation be if the procedure were repeated? Would a new series of measurements yield significantly different results?

The most intuitive way to answer these questions is to construct *confidence intervals* for the statistics. In contrast to a single computed value (a *point estimate*) for the statistic, a confidence interval is a range of estimates. A probability p called the *confidence level* is associated with this range. Most commonly, p is chosen to be 95% and is kept constant during confidence interval comparisons. Confidence intervals are intuitive because their size indicates reliability: short intervals indicate that the statistic is precisely known, while wide intervals indicate uncertainty. For example, if the confidence interval for the mean execution of task A is [1, 1.1] milliseconds, and that for task B is [0.998, 0.999] milliseconds, then B's mean is known with more certainty than A's and is also distinguishably smaller (at the confidence level). See the *Confidence intervals* section in the article supplement for more discussion.

Historically, confidence intervals could only be easily calculated for a few common

PDFs (such as the Gaussian) and simple statistics (such as the mean). In the late 1970s, however, a technique called *bootstrapping* was developed. It is the best general technique for producing confidence intervals. It works for any statistic, not just simple ones like the mean. Furthermore, the nonparametric form of bootstrapping makes no assumptions about the underlying PDF. It never yields unphysical results (for example, negative execution times for a confidence interval lower bound), and it can yield more narrow and accurate confidence intervals than if you make wrong assumptions about the PDF (such as that it is Gaussian). Details on bootstrapping are beyond this article's scope, but the framework I discuss below includes a class named `Bootstrap` that performs the calculations; see its Javadocs and source code for more information (see [Resources](#)).

In summary, you must:

- Perform many benchmark measurements.
- From the measurements, compute the mean and standard deviation.
- Use these statistics to decide that two tasks are either clearly distinguished in speed (means greater than three standard deviations apart), or else that they overlap.
- Compute confidence intervals for the mean and standard deviation to indicate how reliable they are.
- Rely on bootstrapping as the best way to compute confidence intervals.

Introduction to the framework

Up until now, I have discussed the general principles of benchmarking Java code. Now I introduce a "ready to run" benchmarking framework that addresses many of these issues.

Project

Download the project ZIP file from this article's companion site (see [Resources](#)). The ZIP file contains both source and binary files, as well as a simple build environment. Extract the contents into any directory. Consult the top-level `readMe.txt` file for more details.

API

The framework's essential class is named `Benchmark`. It is the only class that most users will ever need to look at; everything else is ancillary. The API for most uses is simple: you supply the code to be benchmarked to a `Benchmark` constructor. The benchmarking process is then fully automatic. The only subsequent step that you typically do is generate a [results report](#).

Task code

Obviously, you must have some code whose execution time you want to benchmark. The only restriction is that the code be contained inside a `Callable` or `Runnable`. Otherwise, the target code can be anything expressible in the Java language, ranging from self-contained microbenchmarks to code that calls a full application.

Writing your task as a `Callable` is usually more convenient. `Callable.call` lets you throw checked `Exceptions`, whereas `Runnable.run` forces you to implement `Exception` handling. And, as I described in Part 1's [Dead-code elimination \(DCE\)](#) section, it is slightly easier to prevent DCE with a `Callable` than a `Runnable`. The one advantage of writing your task as a `Runnable` is that it can minimize object creation and garbage collection overhead; see the *Task code: Callable versus Runnable* section in the article supplement for details.

Results report

The easiest way to get a benchmark results report is to call the `Benchmark.toString` method. This method yields a single-line summary report of only the most important results and [warnings](#). You can obtain a detailed multiline report, with all the results and full explanations, by calling the `Benchmark.toStringFull` method. Otherwise, you can call various accessors of `Benchmark` to generate a custom report.

Warnings

`Benchmark` tries to diagnose some common problems and warn the user in the [results report](#) if it detects such problems. The warnings include:

- **Measurements too low:** The [Resource reclamation](#) section in Part 1 describes how `Benchmark` issues a warning if it determines that garbage collection and object finalization costs were not fully accounted for.
- **Outliers and serial correlation:** The execution-time measurements are subjected to *outlier* and *serial correlation* statistical tests. Outliers indicate that a major measurement error happened. For example, large outliers occur if some other activity started on the computer during measurement; small outliers might hint that DCE occurred. Serial correlation indicates that the JVM has not reached its steady-state performance profile (which should be characterized by execution times having small random variations). In particular, positive serial correlation indicates a trend (either up or down), and negative serial correlation indicates mean reversion (for example, oscillating execution times); neither is good to see.
- **Standard deviation inaccurate:** See the *Standard deviation warnings* section in the article supplement for details.

Simple example

The code fragment in Listing 1 illustrates the points I've discussed so far in this section:

Listing 1. Benchmarking computation of the 35th Fibonacci number

```
public static void main(String[] args) throws Exception {
    Callable<Integer> task =
        new Callable<Integer>() { public Integer call() { return fibonacci(35); } };
    System.out.println("fibonacci(35): " + new Benchmark(task));
}

protected static int fibonacci(int n) throws IllegalArgumentException {
    if (n < 0) throw new IllegalArgumentException("n = " + n + " < 0");
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

In Listing 1, `main` defines the code to be benchmarked as a `Callable` and simply supplies it to a `Benchmark` constructor. This `Benchmark` constructor executes a task many times, first to ensure [code warmup](#), then to gather execution statistics. After the constructor returns, the code context in which it lives (that is, inside the `println`) causes an implicit call to the new `Benchmark` instance's `toString` method, which reports the benchmark's summary statistics. Use of `Benchmark` is usually this simple.

On my configuration, I obtain this result:

```
fibonacci(35): first = 106.857 ms, mean = 102.570 ms (CI deltas: -35.185 us,
+47.076 us), sd = 645.586 us (CI deltas: -155.465 us, +355.098 us)
WARNING: execution times have mild outliers, SD VALUES MAY BE INACCURATE
```

Interpretation:

- The first time that `fibonacci(35)` was called, it took 106.857 milliseconds to execute.
- A point estimate for the mean of the execution time is 102.570 milliseconds. The 95% confidence interval for the mean is about -35/+47 microseconds about the point estimate, namely [102.535, 102.617] milliseconds, which is relatively narrow, so the mean is known with confidence.
- A point estimate for the standard deviation of the execution time is 645.586 microseconds. The 95% confidence interval for the standard deviation is about -155/+355 microseconds about the point estimate, namely [490.121, 1000.684] μ s, which is relatively wide, so it is known with much less confidence. In fact, the warning at the end says that the

standard deviation was [not accurately measured](#).

- The result also warns about outliers. They are ignorable in this case, but if you were worried, you could rerun the code using `Benchmark`'s `toStringFull` method, which would list the values of all the offenders, allowing you to make a judgment call.

Data-structure access times

A better illustration of benchmarking issues, and how `Benchmark` can be used to tackle them, are benchmarks that compare access times for several common data structures.

Measuring just the access time of a data structure is a true microbenchmark. Granted, many caveats apply to microbenchmarks. (For example, they may give little indication of how a full application performs.) But the measurements here redeem themselves because they are tricky to do accurately, and because the many interesting issues that arise (such as cache effects) make them a good example.

Consider the code in Listing 2, which benchmarks array access times (code for the other data structures looks similar):

Listing 2. Array access benchmarking code

```
protected static Integer[] integers;    // values are unique (integers[i] <= i)

protected static class ArrayAccess implements Runnable {
    protected int state = 0;

    public void run() {
        for (int i = 0; i < integers.length; i++) {
            state ^= integers[i];
        }
    }

    public String toString() { return String.valueOf(state); }
}
```

Ideally, the code would do nothing but access the `integers` array. However, in order to access all the elements of `integers` conveniently, I used a loop, which introduces parasitic loop overhead to the benchmarking. Note that I used an old-fashioned explicit `int` loop, rather than the more convenient `for (int i : integers)` enhanced `for` loop, because it is slightly faster, even for arrays (see [Resources](#)). The loop I used here should be a minor defect, because any decent Just-in-time (JIT) compiler will perform loop unrolling, which lessens the impact.

More serious, however, is the code needed to prevent DCE. First, a `Runnable` was chosen for the task class because, as I mentioned earlier, it can minimize object creation and garbage collection overhead, which is critical for a lightweight microbenchmark such as this one. Second, given that this task class is a `Runnable`,

the result of the array access needs to be assigned to the `state` field (and `state` needs to be used in an overriding `toString` method) to prevent DCE. Third, I bitwise XOR the array access with the previous value of `state` to ensure that *every* access is carried out. (Merely doing `state = integers[i]` might trigger a smart compiler to recognize that the entire loop could be skipped, and simply do `state = integers[integers.length - 1]` instead.) These additional four operations (field read, autounboxing of an `Integer` to an `int`, bitwise XOR, field write) are unavoidable overhead that blemish the benchmark: *it is actually measuring more than just array access times*. All the other data-structure benchmarks are similarly tarnished, although for data structures with relatively large access times, the relative effect is negligible.

Figures 3 and 4 present the results for access times on my usual desktop configuration, for two different sizes of `integers`:

Figure 3. Data-structure access times (case: 1024 elements)

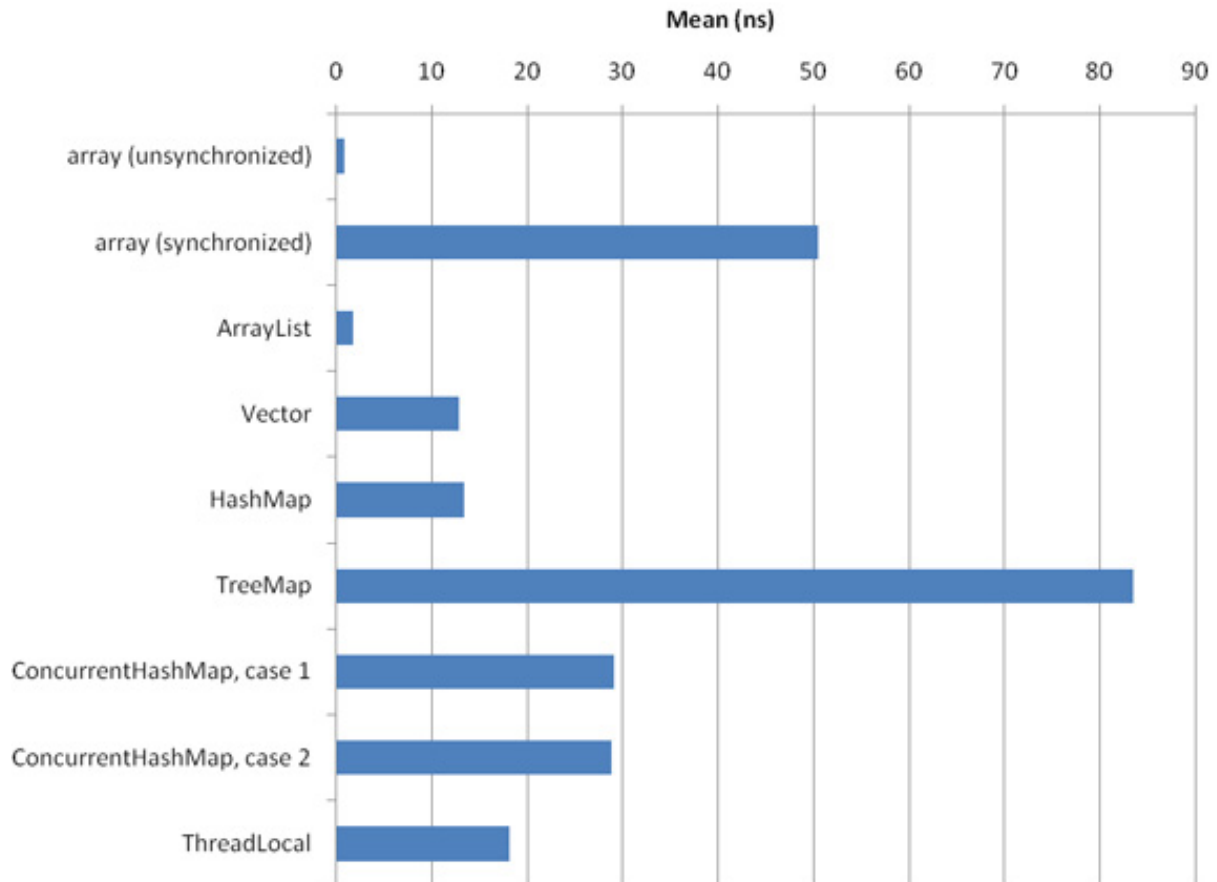
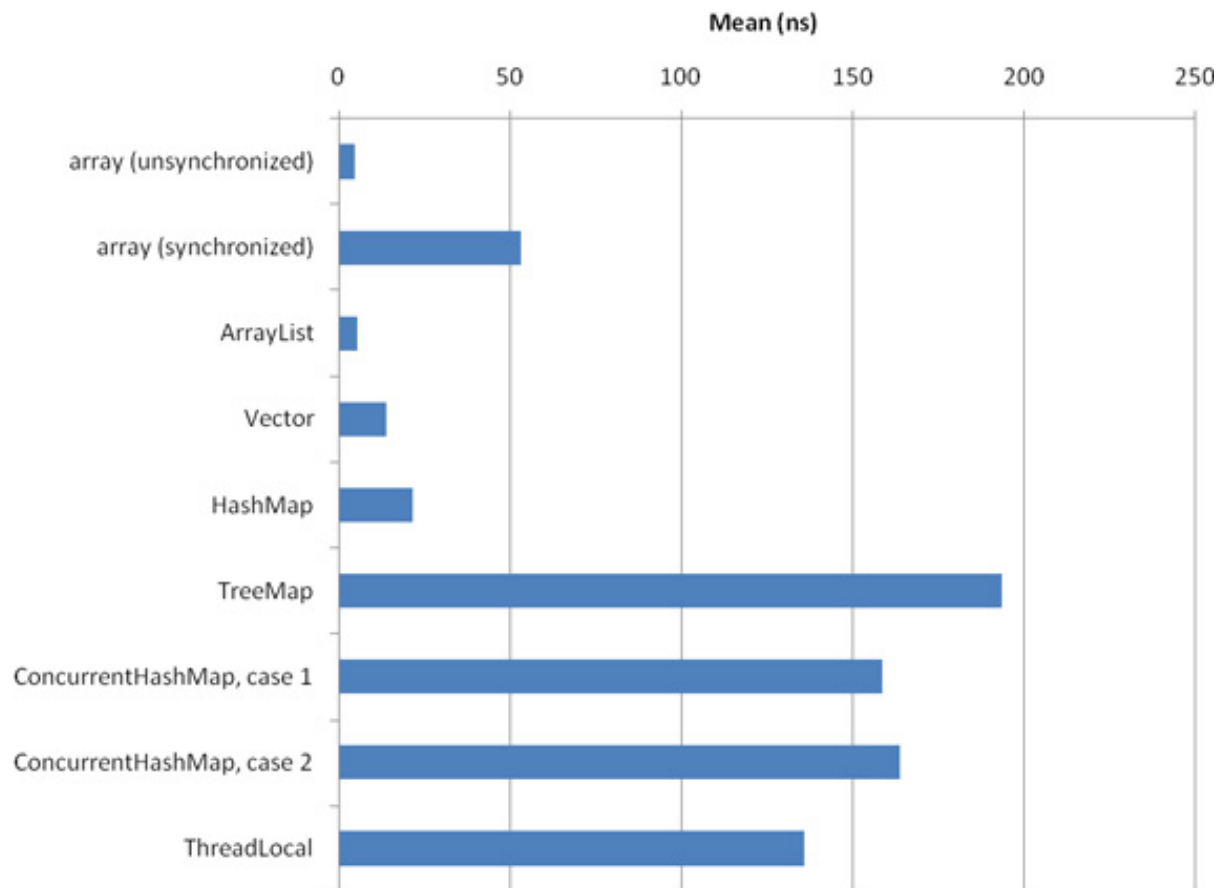


Figure 4. Data-structure access times (case: 1024 × 1024 elements)



Comments on the results:

- Only the point estimates for the mean execution times are shown. (The confidence intervals for the means are always so tight — typical width is ~1000 times smaller than the mean — that they will not show up on these graphs.)
- Only a single thread is used for all these benchmarks; the synchronized classes are under no thread contention.
- The array element access (unsynchronized) code has already been presented in [Listing 2](#).
- The array element access (synchronized) code is identical except that the loop body is: `synchronized (integers) { state ^= integers[i]; }`
- The difference between `ConcurrentHashMap` case 1 and case 2 in Figures 3 and 4 is solely in the `ConcurrentHashMap` constructor used: case 1 specifies `concurrencyLevel = 1`, and case 2 uses the default (which is 16). Because only a single thread is used for all these

benchmarks, case 1 ought to be slightly faster.

- Every benchmark had one or more warnings:
 - Almost all had outliers, but it appears that none were so extreme that their presence affects the results very much.
 - All of the 1024 × 1024 elements results had serial correlation. It is not clear how much of an issue this is; none of the 1024 elements results had serial correlation.
 - The standard deviations were always impossible to measure (typical of microbenchmarks).

These results are probably what you'd expect: an unsynchronized array access is the fastest data structure. An `ArrayList` is second, being almost as fast as a raw array. (Presumably the server JVM is doing an excellent job of inlining direct access to its underlying array.) It is much faster than the synchronized but otherwise similar `Vector`. The next-fastest data structure is `HashMap`, followed by `ThreadLocal` (essentially, a specialized hash table in which the current thread is the key). Indeed, in these tests, `HashMap` is almost as fast as `Vector`, which seems impressive until you consider that `Integers` are being used as keys, and they have a particularly fast `hashCode` implementation (it simply returns the `int` value). Next come the `ConcurrentHashMap` cases, followed by the slowest structure of all, `TreeMap`.

Although the results aren't shown here, I have done these same exact benchmarks on a totally different machine (a SPARC-Enterprise-T5220 running at 1.167GHz with 32GB RAM using SunOS asm03 5.10; the same JVM version and settings as my desktop were used, but I now configured `Benchmark` to measure CPU times, not elapsed times, because the tests are all single-threaded and Solaris supports this so well). The relative results are the same as those above.

The only major anomaly in the results I've presented above are the synchronized array access times: I would have thought that they would be comparable to a `Vector`, but they are consistently more than three times slower. Best guess why: some lock-related optimizations (such as lock elision or lock biasing) failed to kick in (see [Resources](#)). This guess seems to be confirmed by the fact that this anomaly is not present in Azul's JVM, which uses customized lock optimizations.

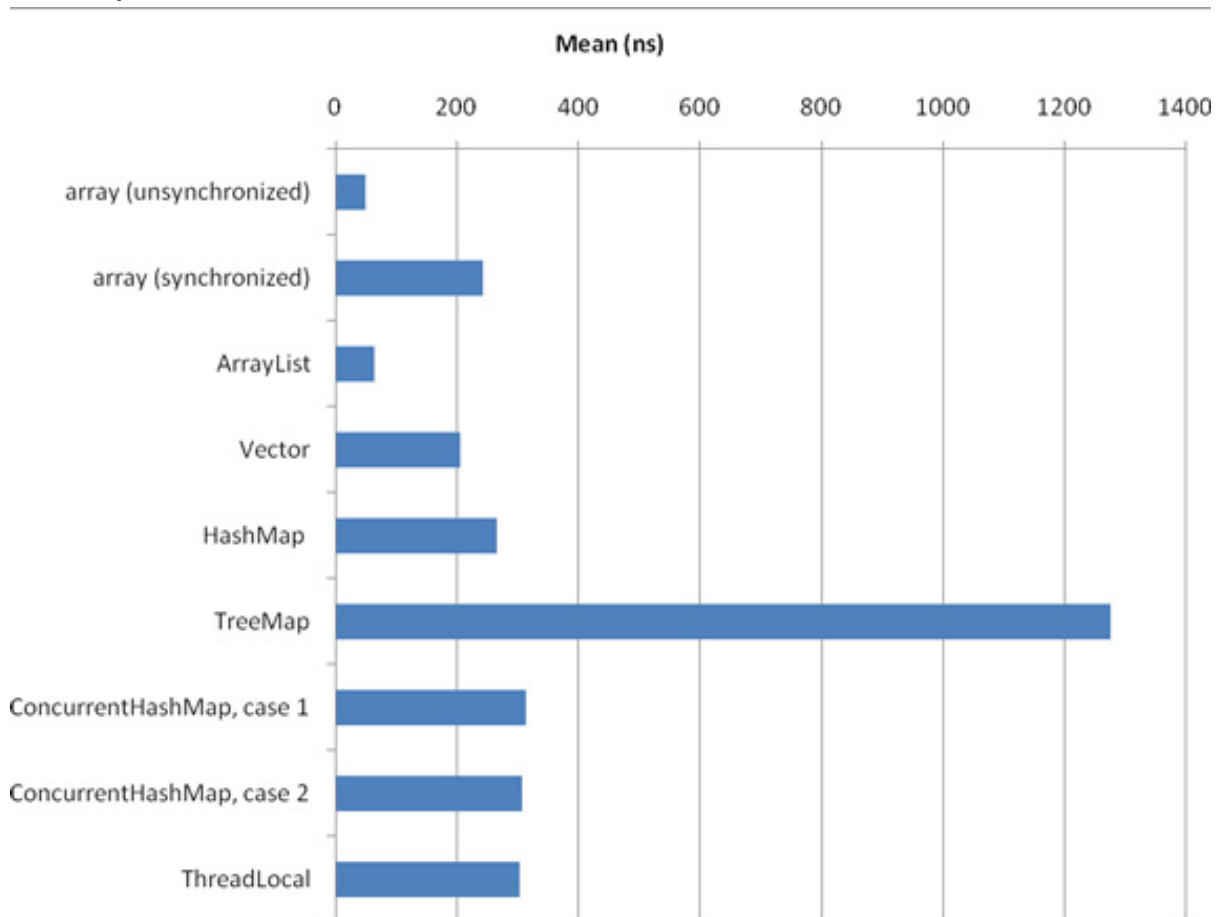
A minor anomaly is that the `ConcurrentHashMap` case 1 is only faster than case 2 when 1024 × 1024 elements are used and is actually slightly slower when 1024 elements are used. This anomaly is probably due to minor memory-placement effects of the different number of table segments. This effect is not present on the T5220 box (case 1 is always slightly faster than case 2, regardless of the number of elements).

Something that is not an anomaly is the faster performance of `HashMap` relative to `ConcurrentHashMap`. The code in both cases is like [Listing 2](#), except that `state`

`^= integers[i]` is replaced with `state ^= map.get(integers[i])`. The elements of `integers` occur in sequential order (`integers[i].intValue() == i`) and are supplied as keys in that same order. It turns out that the hash preconditioning function in `HashMap` has better sequential cache locality than `ConcurrentHashMap` (because `ConcurrentHashMap` needs better high-bit spreading).

This brings up an interesting point: how much do the results I've presented above depend on the fact that `integers` is being iterated through in order? Could there be memory locality effects that some data structures benefit from more than others? To answer these questions, I reran those benchmarks but picked random elements of `integers` instead of sequential elements. (I did this by using a software linear feedback shift register to generate pseudorandom values; see [Resources](#). It adds about 3 nanoseconds of overhead to each data-structure access.) The results for 1024 x 1024 elements of `integers` are shown in Figure 5:

Figure 5. Data-structure access times (case: 1024 x 1024 elements, random access)



You can see that compared to [Figure 4](#), `HashMap` now has about the same performance as `ConcurrentHashMap` (as I argued that it should). `TreeMap` has

appalling random access. The array and `ArrayList` data structures are still best, but their relative performance has faded (they each have about 10 times worse random-access performance, whereas `ConcurrentHashMap` has only about two times worse random-access performance).

One more point: Figures 3, 4, and 5 plot the *individual* access times. For example, from [Figure 3](#), you can see that the time to access a single element from a `TreeMap` is a little more than 80 nanoseconds. But the tasks all look like [Listing 2](#); namely, each task internally performs multiple data accesses (that is, they loop over each element of `integers`). How do you extract individual action statistics from a task that consists of several *identical* actions?

What I left out of [Listing 2](#) was the parent code, which shows how to handle such tasks. You can use code like Listing 3:

Listing 3. Code for tasks with multiple actions

```
public static void main(String[] args) throws Exception {
    int m = Integer.parseInt(args[0]);

    integers = new Integer[m];
    for (int i = 0; i < integers.length; i++) integers[i] = new Integer(i);

    System.out.println(
        "array element access (unsynchronized): " + new Benchmark(new ArrayAccess(), m));
    // plus similar lines for the other data structures...
}
```

In Listing 3, I use a two-argument version of the `Benchmark` constructor. The second argument, shown in bold in Listing 3, specifies the number of *identical actions* that task consists of, which is `m = integers.length` in this case. See the *Block statistics versus action statistics* section in the article supplement for more details.

Optimal portfolio computation

So far in this article, I have only considered microbenchmarks. Interesting as they may be, measuring the performance of real applications is where benchmarking's true utility lies.

An example that may be of interest to many of you who are investors is Markowitz mean-variance portfolio optimization, which is a standard technique used by financial advisors to construct portfolios with superior risk/reward profiles (see [Resources](#)).

One firm that provides a Java library to do these calculations is WebCab Components (see [Resources](#)). The code in Listing 4 benchmarks the performance of its Portfolio v5.0 (J2SE Edition) library in solving for the *efficient frontier* (see

[Resources](#)):**Listing 4. Portfolio-optimization benchmarking code**

```

protected static void benchmark_efficientFrontier(
    double[][][] rets, boolean useCons, double Rf, double scale
) throws Exception {
    Benchmark.Params params = new Benchmark.Params(false);    // false to meas only first
    params.setMeasureCpuTime(true);

    for (int i = 0; i < 10; i++) {
        double[][] returnsAssetsRandomSubset = pickRandomAssets(rets, 30);
        Callable<Double> task = new EfTask(returnsAssetsRandomSubset, useCons, Rf, scale);
        System.out.println(
            "Warmup benchmark; can ignore this: " + new Benchmark(task, params) );
    }

    System.out.println();
    System.out.println("n" + "\t" + "first" + "\t" + "sharpeRatioMax");
    for (int n = 2; n <= rets.length; n++) {
        for (int j = 0; j < 20; j++) {    // do 20 runs so that can plot scatter
            double[][] returnsAssetsRandomSubset = pickRandomAssets(rets, n);
            Callable<Double> task = new EfTask(returnsAssetsRandomSubset, useCons, Rf, scale);
            Benchmark benchmark = new Benchmark(task, params);
            System.out.println(
                n + "\t" + benchmark.getFirst() + "\t" + benchmark.getCallResult());
        }
    }
}

```

Because this is an article on benchmarking, and not portfolio theory, Listing 4 leaves out the code for the `EfTask` inner class. (In words: `EfTask` takes asset historical return data, computes expected returns and covariances from it, solves for 50 points along the efficient frontier, and then returns the point that has the maximal Sharpe ratio; see [Resources](#). This optimal Sharpe ratio is one measure for how good the best portfolio can be for a specific set of assets, identifying the best return on a risk-adjusted basis. See the relevant source file in the article's code download for details.)

This code's purpose is to determine simultaneously both the execution time and portfolio quality as a function of the number of assets — potentially useful information for someone doing portfolio optimization. The `n` loop in Listing 4 makes this determination.

This benchmark presented several challenges. First, the calculations can take a long time, especially once a significant number of assets are being considered. So, I avoided simple code like `new Benchmark(task)`, which carries out 60 measurements (by default). Instead, I chose to create a custom `Benchmark.Params` instance that specifies that only a single measurement should be done. (It also specifies that CPU time measurements should be done, instead of the default elapsed time, simply to illustrate how this can be accomplished. It is okay here to do this because the `WebCab Components` library does not create threads in this context.) However, before any of these single measurement benchmarks are carried out, the `i` loop does several throwaway benchmarks to let the JVM first fully

optimize the code.

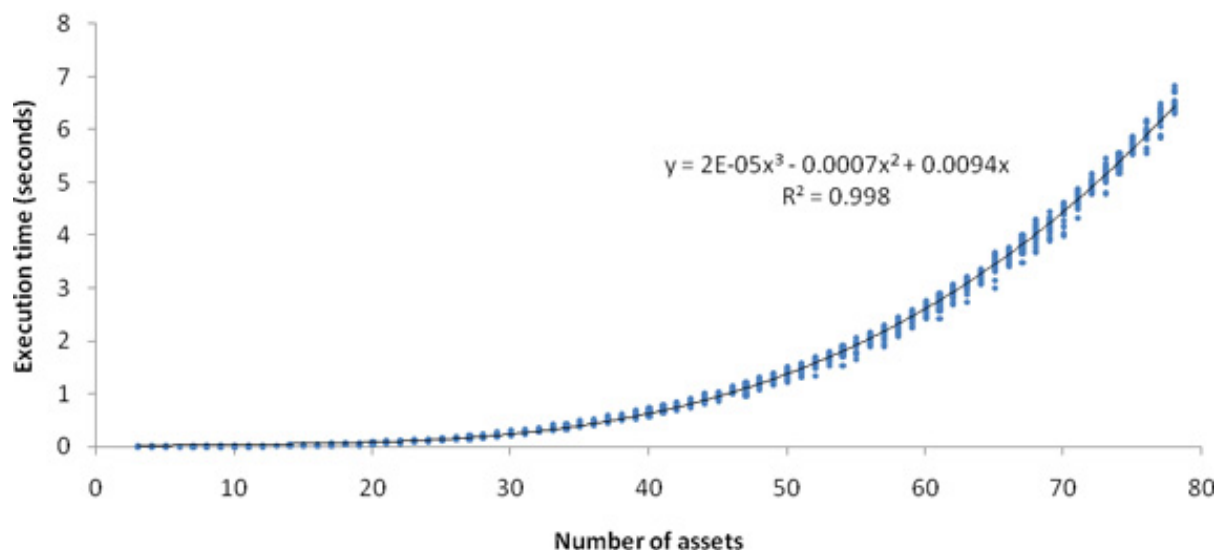
Second, the usual [results report](#) is inadequate for this benchmark's needs, so I generate a customized one that should provide tab-delimited numbers only, so that it can easily be copied and pasted into a spreadsheet for subsequent graphing. Because only a single measurement is taken, the execution time is retrieved using Benchmark's `getFirst` accessor method. The maximum Sharpe ratio for the given set of assets is the return value of the `Callable` task. It is grabbed via Benchmark's `getCallResult` accessor method.

Furthermore, I wanted to get a visual representation of the scatter in the results, so for a given number of assets, the inner `j` loop performs each benchmark 20 times. This is what causes the 20 dots per number of assets in the graphs below. (In some cases, the dots overlap so much that it looks like only a few are present.)

On to the results. The assets I used are the stocks in the current OEX (S&P 100) index. I used the weekly capital gains over the last three whole years (January 1, 2005 through December 31, 2007) for the historical returns; dividends are ignored (if included, they would slightly raise the Sharpe ratios).

Figure 6 is a graph of the execution time as a function of the number of assets:

Figure 6. Portfolio optimization execution time

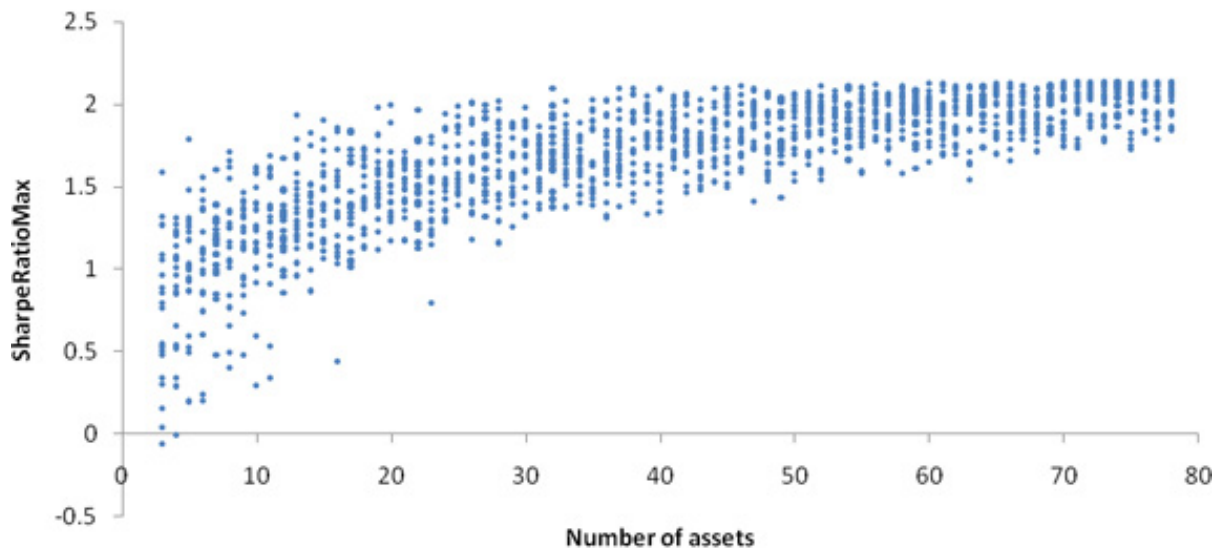


So, the execution time grows cubically as a function of the number of assets. The scatter in these measurements is real and is not benchmarking error: the execution time of portfolio optimization depends on the type of assets considered. In particular, certain types of covariances can require very careful (small step size) numerical calculations, which affects the execution time.

Figure 7 is a plot of the portfolio quality (maximum Sharpe ratio) as a function of the

number of assets:

Figure 7. Portfolio quality



The maximum Sharpe ratio initially rises but soon stops increasing at around 15 to 20 assets. Thereafter, the only effect is that the range in values shrinks toward the maximum as the number of assets increases. This effect too is real: it occurs because as the number of assets considered grows, the odds that you have all the "hot" assets (which will dominate the optimal portfolios) plateaus at 100%. See the *Portfolio optimization* section in the article supplement for a few more thoughts beyond the scope of this article.

Final caveats

Microbenchmarks should mirror real use cases. For instance, I chose to measure data structure access times because the JDK collections were designed with the expectation that typical applications do a mix of around 85% read/traverse, 14% add/update, and 1% remove. However, beware that changing those mixtures can cause the relative performance to be almost arbitrarily different. Another subtle danger is complexity of the class hierarchy: microbenchmarks often use a simple class hierarchy, but method-call overhead can become significant in complex class hierarchies (see [Resources](#)), so an accurate microbenchmark must mirror reality.

Make sure that your benchmark results are relevant. Hint: a microbenchmark pitting `StringBuffer` versus `StringBuilder` will likely not tell you much about a Web server's performance; higher-level architectural choices (that do not lend themselves to microbenchmarking) are probably vastly more significant. (Although you should automatically use `ArrayList/HashMap/StringBuilder` in almost all of your code anyway, instead of the older `Vector/Hashtable/StringBuffer`.)

Be careful about relying on microbenchmarks alone. For example, if you want to determine the effect of a new algorithm, then measure it not only in a benchmark environment, but also in real application scenarios to see if it actually makes enough of a difference.

Obviously, you can only make general performance conclusions if you have tested a reasonable sample of computers and configurations. Unfortunately, a typical error is benchmarking only on development machines and assuming that the same conclusions hold for all the machines that the code is run on. If you want to be thorough, you need multiple hardware, and even different JVMs (see [Resources](#)).

Do not ignore profiling. Run your benchmark with every kind of profiling tool you can get to confirm that it is behaving as expected (for example, that most of the time is spent in what you think should be the critical methods). This may also be able to confirm that DCE is not invalidating your results.

Ultimately, there is no substitute for really knowing how things operate at a low level in order to draw legitimate conclusions. For example, if you want to see whether the Java language's sine implementation (`Math.sin`) is faster than C's, you will probably find on x86 hardware that Java's sine is much slower. This is because Java correctly avoids the fast but inaccurate x86 hardware helper instructions. Ignorant people doing this benchmark have concluded that C is much faster than the Java language, when all they really determined is that a dedicated (but inaccurate) hardware instruction is faster than a precise software computation.

Conclusion

Benchmarking often requires performing many measurements and using statistics to interpret the results. The benchmarking framework this article presents supports these features and addresses many other issues. Whether you use this framework or — guided by the material I've presented in both parts of the series — create your own, you're now better equipped to help ensure your Java code performs efficiently.

Resources

Learn

- [Companion site for this article](#): You'll find the article supplement and full sample code here.
- Statistics on [Wikipedia](#): Learn more about [deconvolution](#), [typical value](#), [arithmetic mean](#), [measurement spread](#), [standard deviation](#), [confidence intervals](#), [probability density functions](#), [bootstrapping](#), [outliers](#), and [serial correlation](#).
- ["He who refuses to do arithmetic is doomed to talk nonsense"](#): Many pithy quotations are attributed to computer scientist John McCarthy.
- [Use Enhanced For Loop Syntax With Caution](#): Pitfalls of the enhanced `for` loop in embedded development.
- ["Java theory and practice: Synchronization optimizations in Mustang"](#) (Brian Goetz, developerWorks, October 2005): Lock elision (among other concepts) explained.
- ["Biased Locking in HotSpot"](#) (David Dice's Weblog, August 2006): Read about the biased locking scheme in HotSpot.
- [Linear feedback shift register](#): Wikipedia's entry on this engineering concept.
- [Modern portfolio theory](#): Wikipedia explains Markowitz mean-variance portfolio optimization, efficient frontier, and other portfolio-analysis concepts.
- [WebCab Portfolio v5.0 \(J2SE Edition\) library](#): Components benchmarked in some of this article's examples.
- [Sharpe ratio](#): Wikipedia's entry on this measure.
- ["Polymorphism Performance Mysteries Explained"](#) (Dr. Heinz M. Kabutz, The Java Specialists' Newsletter, April 2001): Read why method call overhead can become significant in complex class hierarchies.
- [Sun's, IBM's, and BEA's JVMs](#): Thorough Java benchmarking should be performed on multiple platforms.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Brent Boyer

Brent Boyer has been a professional software developer for more than nine years. He is the head of Elliptic Group, Inc., a software development company in New York, N.Y.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.