Urban legends are kind of like mind viruses; even though we know they are probably not true, we often can’t resist the urge to retell them (and thus infect other gullible “hosts”) because they make for such good storytelling. Most urban legends have some basis in fact, which only makes them harder to stamp out. Unfortunately, many pointers and tips about Java performance tuning are a lot like urban legends -- someone, somewhere, passes on a “tip” that has (or had) some basis in fact, but through its continued retelling, has lost what truth it once contained. This month, Brian Goetz examines some of these urban performance legends and sets the record straight.

Have you heard about the old lady who tried to dry off her rain-soaked dog in the microwave (false)? Or the aircraft carrier commander who insisted that the lighthouse yield the right of way (false)? Or how alligators came to live in the sewers of New York (false)? Or that a postal truck has the right-of-way over local emergency vehicles -- including police cars, fire trucks, and ambulances -- by virtue of its federal status (also false)? Now, tell the truth -- how many times have you retold or forwarded these stories to others, even if you were unconvinced, or even downright skeptical, of their veracity?

Urban legends persist because there’s something about them that makes them just plausible enough to get retold. Unfortunately, urban legends are not confined to stories of flushing baby alligators down the toilet. There’s a lot of bad advice floating around among programmers, such as what makes Java programs perform well or poorly, and a lot of it is about as scientifically accurate as the alligator story. But it’s plausible enough that it gets retold, and most of us never bother to question or experimentally verify these theories.

This month, I’m going to look at several general statements about Java performance tuning that share many characteristics of urban legends. Some of them have some basis in fact, but all have been inappropriately promoted to the status of performance gospel.
Urban performance legend #1: Synchronization is really slow

True or false: synchronized methods are fifty times slower than equivalent unsynchronized methods? This little gem shows up in Dov Bulka's otherwise decent book on low-level performance tuning, and has been repeated in many other sources. Like all urban legends, it has some basis in fact. In all fairness to Bulka, at some point in the distant past -- perhaps the days of JDK 1.0 -- it was probably true that if you ran the microbenchmark in Listing 1, it showed that testSync took 50 times longer to run than testUnsync.

Even if it was once true, though, it certainly isn't anymore. JVMs have improved tremendously since JDK 1.0. Synchronization is implemented more efficiently, and the JVM is sometimes able to identify that a synchronization is not actually protecting any data and can be eliminated. But more significantly, the microbenchmark in Listing 1 is fundamentally flawed. First of all, microbenchmarks rarely measure what you think they're measuring. In the presence of dynamic compilation, you have no idea what bytecode the JVM decides to convert into native code or when, so you can't really compare apples to apples.

In addition, you have no idea what the compiler or JVM is optimizing away -- some Java compilers will completely optimize away calls to unsyncMethod, because it does nothing, and others may also optimize away syncMethod, or the synchronization on syncMethod, because it also does nothing. Which does your compiler optimize away, and under what circumstances? You don't know, but it almost certainly distorts the measurements.

Regardless of the actual numbers, concluding that unsynchronized method calls are X times faster than synchronized ones from a benchmark of this type is just plain foolish. Synchronization is likely to add a constant overhead to a block of code, not slow it down by a constant factor. How much code is in the block will dramatically affect the "ratio" computed by Listing 1. The synchronization overhead as a percentage of the time to execute an empty method is a meaningless number.

Once you start comparing the runtime of real synchronized methods to their unsynchronized counterparts on modern JVMs, you'll find that the overhead is nothing near the alarmist "50 times" that is so often bandied about. Read Part 1 of the series Threading lightly, "Synchronization is not the enemy" (see Related topics), for some rough and unscientific measurements of the overhead of synchronization. To be sure, there is some overhead to uncontended synchronization (and much more for contended synchronization), but synchronization is not the sewer-dwelling, performance-eating alligator that so many fear.

Listing 1. Flawed microbenchmark for measuring synchronization overhead

```java
public static final int N_ITERATIONS = 10000000;

public static synchronized void syncMethod() {
}

public static void unsyncMethod() {
}

public static void testSync() {
    for (int i=0; i<N_ITERATIONS; i++)
        syncMethod();
```

Java theory and practice: Urban performance legends
The "synchronization is slow" myth is a very dangerous one, because it motivates programmers to compromise the thread-safety of their programs to avoid a perceived performance hazard. In fact, they often think they are being very clever by doing so. It was the fear of this very myth that inspired developers and writers to promote the clever-seeming, but fatally flawed "double-checked locking" idiom, which appears to eliminate synchronization from a common code path, but in fact compromises the thread-safety of your code. Thread-safety problems are time bombs in your code just waiting to go off, and when they do, they will go off at the worst possible time -- when your program is under heavy load. Legitimate performance concerns are a bad reason to compromise the thread-safety of your programs; fear of performance myths is an even worse reason.

Urban performance legend #2: Declaring classes or methods final makes them faster

I discussed this myth in October's column (see Related topics), so I won't rehash it in great detail here. Many articles have recommended making classes or methods final, because it makes it easier for the compiler to inline them and therefore should result in better performance. It's a nice theory. Too bad it's not true.

This myth is even more interesting than the synchronization myth, because there's no data to support it -- it just seems plausible (at least the synchronization myth has a flawed microbenchmark to support it). Someone must have decided that it must work this way, told the story with confidence, and once the story got started, it was spread far and wide.

The danger of this myth, just like the synchronization myth, is that it leads developers to compromise good object-oriented design principles for the sake of a nonexistent performance benefit. Whether to make a class final or not is a design decision that should be motivated by an analysis of what the class does, how it will be used and by whom, and whether you can envision ways in which the class might be extended. Making a class final because it is immutable is a good reason to do so; making a complex class final because it hasn't been designed for extension is also a good reason. Making a class final because you read somewhere that it will run faster (even if it were true) is not.
Urban performance legend #3: Immutable objects are bad for performance

It is common to use immutable objects (such as `String`) to describe data that changes -- when the data changes, you create a new object instead of mutating its state. The performance trade-offs between mutable and immutable objects are complicated. Depending on how the object is used in your program, you might find that immutable objects actually offer a performance advantage (because you don't have to defensively copy them). On the other hand, you may find that they impose a significant performance penalty (because you are modeling frequently changing data and therefore creating new objects). Then again, you might not be able to measure a difference.

The "immutable objects are slow" myth is rooted in the more general performance principle that creating many temporary objects is bad for performance. While creating temporary objects certainly does entail additional work for the allocator and garbage collector, JVMs have gotten much better at mitigating the performance impact of temporary object creation. The performance impact of object creation, while certainly real, is not as significant in most programs as it once was or is still widely believed to be.

Consider the difference between an immutable `StringHolder` class and a mutable one, as shown in Listing 2. In one case, if you wanted to change the string it contained, you would create a new instance of `StringHolder`; in the other, you would call a setter method on the existing `StringHolder` to set the contained string. To take a concrete example, suppose you were wrapping a string with a delimiter. How would these two approaches compare in performance?

Listing 2. Immutable versus mutable StringHolder class

```java
// mutable
stringHolder.setString("/" + stringHolder.getString() + "/");

// immutable
stringHolder = new StringHolder("/" + stringHolder.getString() + "/");
```

If you thought that the extra `StringHolder` object creation made a big difference in the net performance, you'd be wrong. The mutable approach does plenty of object creation, too. To perform the string concatenation, a `StringBuffer` object is created, which entails creating a `char` array, and then a `String` object is created to describe the final character array. If the resulting string is longer than the default buffer size used by `StringBuffer`, the internal character array will be reallocated, resulting in one or more additional object creations. So the mutable approach involves at least three object creations, while the immutable approach involves one more. There is a potential performance difference, but it's hardly the difference between doing no object creation and doing lots of it.

There is also a significant, but difficult-to-measure, interaction with garbage collection here, which may also have an effect on performance. Modern generational garbage collectors perform significantly better when new objects reference old objects, rather than the other way around. Creating a new immutable holder object yields exactly this configuration; mutating an existing container object to reference the newly created string does the opposite.
Just like the first two myths, this one encourages programmers to sacrifice good object-oriented design principles for the sake of a performance benefit. Immutable objects are simpler and less error-prone to write, maintain, and use than mutable ones. Should you give these benefits up for the sake of performance? Maybe, but only once you are convinced you have a performance problem, you know its cause, and you know that breaking the immutability of one particular class will help you reach your performance goals. In the absence of demonstrated performance problems and stated performance goals, you should err on the side of program correctness, not higher performance.

Lessons learned

There are several common themes across all of these performance legends. All of them are rooted in performance claims made in the very early days of Java technology, before significant effort was invested in improving the performance of the JVM. Some of them were true when they were first stated, but the performance of JVMs has improved considerably since then. While we certainly shouldn't ignore the performance impact of synchronization and object creation, we should not elevate them to the status of constructs to be avoided at all cost.

Optimize with concrete performance goals in mind

For each of these myths, the danger is the same: compromising good design principles -- or worse, program correctness -- to achieve a questionable performance benefit. Optimization always carries risk, such as breaking code that already works, making code more complicated and therefore introducing more potential bugs, limiting the generality or reusability of the code, introducing constraints, or just making the code harder to understand and maintain. Until there is a demonstrated performance problem, it is usually best to err on the side of clarity, clean design, and correctness. Save optimizations for situations where performance improvements are actually needed, and employ optimizations that will make a measurable difference.

Performance advice has a short shelf-life

The performance of any given technique is not intrinsic to the technique -- it is also influenced by the environment in which it runs, and program execution environments are changing constantly. Compilers get smarter; processors get faster; libraries get updated; garbage collection and scheduling algorithms change; and the relative speed and cost of processors, cache, main memory, and I/O devices change over time. If technique A was ten times faster than technique B ten years ago, don't assume too much about their relative performance today. Performance observations simply have a short shelf life. When confronted with performance advice, question whether it might be out of date before you accept it as fact.

Given that so much performance advice gets stale quickly, be more skeptical of the effectiveness of performance tips you hear, and be more conservative about applying them to working code. Ask first whether the change will actually bring about a performance improvement in your application, and whether your application is in need of a performance boost at all.

And if you get an e-mail stating that Bill Gates will send you ten dollars for every person you forward his message to, please don't send it to me.
Related topics

- Don't miss the complete *Java theory and practice series* by Brian Goetz. The following columns have particular relevance to this one:
  - "Performance management" (March 2003) covered the importance of integrating performance management into your development cycle.
  - "Is that your final answer?" (October 2002) explored the use and misuse of the final keyword.
  - "To mutate or not to mutate?" (February 2003) described the benefits of immutability
- Part 1 of the three-part series *Threading lightly, "Synchronization is not the enemy"* (developerWorks, July 2001), contains some rough measurements of the overhead of uncontended synchronization on various platforms.
- Dov Bulka's book, *Server-Side Programming Techniques* (Addison-Wesley, June 2000), contains useful tips and stories about low-level Java performance tuning, but be skeptical of the current validity of the numbers, as they were calculated on much older JVMs.
- Read up on the dangers of the double-checked locking idiom (*JavaWorld*, February 2001).
- The urban legends reference page at www.snopes.com debunks and analyzes many urban legends.