

Article

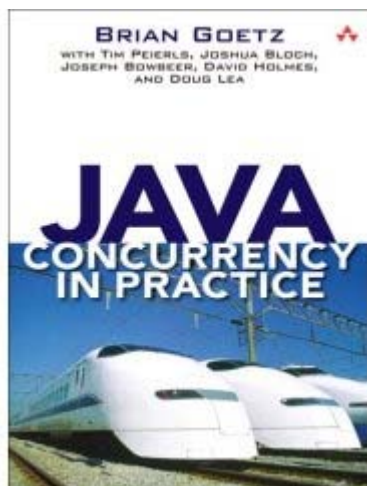
Writing Better Code: A Conversation With Sun Microsystems Technology Evangelist Brian Goetz

By Janice J. Heiss, March 2007

[Articles Index](#)

In recent years, few people have written more about the Java platform than has Sun Microsystems technology evangelist Brian Goetz. Since 2000, he has published some 75 articles on best practices, platform internals, and concurrent programming, and he is the principal author of the book [Java Concurrency in Practice](#), a 2006 Jolt Award Finalist and the best-selling book at the 2006 JavaOne conference. Prior to joining Sun in August of 2006, he was a consultant for 15 years for his software firm, Quiotix, where, in addition to writing about Java technology, he spoke frequently at conferences and gave presentations on threading, the Java programming language memory model, garbage collection, Java technology performance myths, and other topics. In addition, he has consulted on kernel internals, device drivers, protocol implementations, compilers, server applications, web applications, scientific computing, data visualization, and enterprise infrastructure tools. He's participated in a number of open-source projects, including the Lucene text search and retrieval system, and the FindBugs static analysis toolkit.

At Sun, he serves as a consultant on a wide range of topics that extend from Java concurrency to the needs of Java developers, and he contributes to the development of the Java platform. We met with him to get his thoughts on Java technology performance challenges, [Java Platform, Standard Edition 6](#) (Java SE 6), common performance hazards, the challenges of moving from C to Java programming, and ways to write better code.



What should developers understand about concurrency?



They should understand that concurrency is hard but not impossible. People tend to go from one extreme to the other, from thinking "This is easy -- I just synchronize everything" to "This is impossibly hard." I disagree with both. With concurrency, developers should realize that they have to be careful, think hard, and have someone review their code. Concurrent coding is harder than sequential coding, but with proper preparation, developers can do it.



A posting on a [java.sun.com](#) forum raises a question related to concurrency: "Chapter 14 of the Java Language Specification defines (among other things) the synchronized statement:

```
synchronized (expression) { /*do stuff*/ }
```

"It says: ' If the value of the Expression is null, a `NullPointerException` is thrown.' "

The poster suggests extending the Java language to allow null expressions here and not synchronize the block. Some `java.util` collection classes are thread-safe, and some are not -- each class documents this. Likewise, authors of new classes have to choose between implementations. Instead, why not have one class be used in both ways in the same application? Why not allow an instance to select at

runtime whether it behaves in a thread-safe manner or not? The idea would be to increase generality and usefulness, reduce maintenance, and avoid unnecessary class proliferation. Generics did this for classes relative to types.

Your response?

A

Although I think the poster has a nice idea in mind -- increasing the generality of code -- the overwhelmingly likely outcome if this were to be implemented would be to hide errors. Concurrency errors are hard enough to find now -- we don't need any more cases in which the runtime silently swallows them. Instead, we need more tools for identifying concurrency errors.

But this question is really a performance question masquerading as a design question -- the unstated implication being that making a class thread-safe imposes a big performance cost. This perception, that synchronization is really expensive, is another performance myth, one that was true 10 years ago but is no longer true. Every major JDK release has reduced the overhead of synchronization over the previous version. Yes, there's still overhead, but people worry too much about that and not enough about writing clean, correct code.

Wrong Intuitions About Performance Problems

Q

Four years ago you said, "Developers love to optimize code and with good reason. It is so satisfying and fun. But knowing when to optimize is far more important. Unfortunately, developers generally have horrible intuition about where the performance problems in an application will actually be." Do you still believe this?

A

It's truer today than it was four years ago, and more true for Java developers than it was for C. Most performance tuning reminds me of the old joke about the guy who's looking for his keys in the kitchen even though he lost them in the street, because the light's better in the kitchen. We're intimately familiar with the code that we write. The external services that we depend on, whether libraries or external agents, such as databases and web services, are out of sight and out of mind. So when we see a performance problem, we tend to think about places in our code that we can imagine being performance problems. But often, that's not the source of the performance problem -- it's somewhere else in the application's architecture.

Most performance problems these days are consequences of architecture, not coding -- making too many database calls or serializing everything to XML back and forth a million times. These processes are usually going on outside the code you wrote and look at every day, but they are really the source of performance problems. So if you just go by what you're familiar with, you'll be looking for your keys in the kitchen. This is a mistake that developers have always been subject to, and the more complex the application, the more it depends on code you didn't write. Hence, the more likely it is that the problem is outside of your code.

Performance analysis is much harder in the Java programming language than it was in C, where it is more straightforward, because C bears a significant similarity to assembly language. The mapping from C code to machine code is fairly direct. To the extent that it isn't, you can ask the compiler to show you the machine code.

Java applications don't work like C. The runtime constantly modifies the code based on changing conditions and observations. It starts out interpreting the code and then compiles it. It may invalidate the compiled code and recompile it based on information from profiling data or from loading other classes. As a result, the performance characteristics of your code will vary dramatically depending on the environment the code runs in. That makes it harder to say "This code is faster than that code" because you have to account for more context to make a reasonable performance analysis. There are also nondeterministic factors such as the timing and nature of compilation, the interaction of the loaded classes, and garbage collection. So it's harder to do the kind of microperformance optimization with Java code that one can do in C.

At the same time, the fact that the compilation is done at execution time means that the optimizer has far more information to work with than the C compiler does. It knows what classes are loaded and how the method being compiled has actually been used. As a result, it can make far better optimization decisions than a static compiler could. This is great for performance but means it's harder to predict the performance of a given block of code.

"Write Dumb Code"

Q

So how can developers write Java code that performs well?

A

The answer may seem counterintuitive. Often, the way to write fast code in Java applications is to write dumb code -- code that is straightforward, clean, and follows the most obvious object-oriented principles. This has to do with the nature of dynamic compilers, which are big pattern-matching engines. Because compilers are written by humans who have schedules and time budgets, the compiler developers focus their efforts on the most common code patterns, because that's where they get the most leverage. So if you write code using straightforward object-oriented principles, you'll get better compiler optimization than if you write gnarly, hacked-up, bit-banging code that looks really clever but that the compiler can't optimize effectively.

"Often, the way to write fast code in Java applications is to write dumb code -- code that is straightforward, clean, and follows the most obvious object-oriented principles."

So clean, dumb code often runs faster than really clever code, contrary to what developing in C might

Brian Goetz
Technology Evangelist, Sun

have taught us. In C, clever source code turns into the expected idiom at the machine-code level, but it doesn't work that way in Java applications. I'm not saying that the Java compiler is too dumb to translate clever code into the appropriate machine code. It actually optimizes Java code more effectively than does C.

My advice is this: Write simple straightforward code and then, if the performance is still not "good enough", optimize. But implicit in the concept of "good enough" is that you need to have clear performance metrics. Without them, you'll never know when you're done optimizing. You'll also need a realistic, repeatable, testing program in place to determine if you're meeting your metrics. Once you can test the performance of your program under actual operating conditions, then it's OK to start tweaking, because you'll know if your tweaks are helping or not. But assuming "Oh, gee, I think if I change this, it will go faster" is usually counterproductive in Java programming.

Because Java code is dynamically compiled, realistic testing conditions are crucial. If you take a class out of context, it will be compiled differently than it will in your application, which means performance must be measured under realistic conditions. So performance metrics should be tied to indices that have business value -- transactions per second, mean service time, worst-case latency -- factors that your customers will perceive. Focusing on performance characteristics at the micro level is often misleading and difficult to test, because it's hard to make a realistic test case for some small bit of code that you've taken out of context.

Too Much XML



Four years ago, you also said that the overuse of XML was a significant performance problem for Java developers. Have you changed your opinion on that?



Not at all. As I've suggested, in most applications, performance problems today are usually not due to inefficient coding but are the consequences of architectural decisions. For instance, serializing an object to XML, flinging it across the wire, reconstituting it, and turning it back into an object, and operating with it on another system involves a lot of work being performed behind some abstraction barrier.

Abstractions are great at helping us wrap our heads around complicated problems, because they allow us to restrict ourselves to thinking about one part at a time. But abstraction mechanisms often have costs that we overlook when we focus on system design. So using XML as an interchange format is great for integrating disparate systems. But is the performance cost acceptable when we use it as a generic serialization mechanism? Similarly, remote method calls are also convenient, but can we justify the performance cost in business terms? Sometimes the answer is a resounding yes and sometimes not, but the abstraction barriers invite us to not think about the performance implications of architectural decisions sufficiently. Then, when we do encounter performance problems, we often fall back on tweaking the code -- because the light is better there.

Moving From C to Java Programming

"To get the benefit of Java programming, you have to understand that you are not just programming in C with a different syntax."

Brian Goetz
Technology Evangelist, Sun
Microsystems



What particular problems do C or C++ programmers face when they learn Java programming?



The biggest problem is that they're used to bit-level control over pointers, which they consider a good thing, because they understand what kind of code is going to be generated when they run their program through a compiler. Because C programmers are trained to focus on bit-level microperformance issues, they tend to bring this obsession with them to Java programming and miss the forest for the trees.

I try to convince them that by relinquishing some control, they'll get huge productivity and reliability benefits -- and maybe even better performance as well. Some programmers see the trade-off and think it's

great, while others resist it and code Java programs as if they were coding in C, thereby getting the benefits of neither. The Java language is not just a syntax: There's a design philosophy that goes with managed languages. To get the benefit of Java programming, you have to understand that you are not just programming in C with a different syntax.

Advice for Beginners: Learn the Class Libraries



Do you have any advice for those just learning to program in the Java language?



Coming to Java programming from other languages can be overwhelming, primarily because of the sheer size of the class libraries, which can be intimidating. Many who come from other languages ignore the power of the class libraries.

My advice is to take the time to understand what the class libraries can do for you. You don't have to understand the details of every little feature, but spend some time absorbing the spectrum of what they can do -- because they can make you more productive and make your programs smaller, more reliable, and easier to read and maintain.

Experienced Java programmers would do well to learn what's new with each version of the platform, because each version contains library enhancements that can make their job easier.

Java SE 6 Performance Improvements



You've given many presentations on Java performance myths. How do you assess such myths in light of the release of Java SE 6? Which myths have been slowest to die?



Java SE 6 has a lot of nice performance improvements with many optimizations. When you put them all together, it turns into substantial improvement against benchmarks. Java technology is continuing to deliver on the performance promise of managed runtimes. Some people assume that Java software can never be faster than C, and that at best, it can only be equally fast -- because any program written in Java code could be written in C, where you presumably have bit-level control over everything. But that's comparing apples to oranges.

It is possible for Java code to be faster than C. For example, allocation in the Java language is already much faster than it is in C. Java programming enables optimizations not possible in C because C leaves so many important factors, such as allocation and thread management, to libraries. Ironically, it's the bit-level control over pointers, which most C programmers see as their most powerful weapon, that cripples the C compiler's ability to optimize effectively. By giving up that bit of control, you enable a wealth of optimizations that are not possible in C -- and the Java compiler knows more about optimization than 99.99 percent of programmers do.

Java software has always had the potential to be faster than C. The performance improvements in Java SE 6 make it clear that we're heading toward that goal. We're just beginning to apply the optimizations coming out of the Java compiler research community to production languages.

"If I could wave a magic wand and send out one message about Java programming, it would be this: Trust the JVM. It's smarter than you think. Stop trying to outwit or outsmart it. Tell it what you want, and it will do its damndest to make your application run as fast as it can."

Brian Goetz
Technology Evangelist, Sun
Microsystems

If I could wave a magic wand and send out one message about Java programming, it would be this: Trust the JVM. * It's smarter than you think. Stop trying to outwit or outsmart it. Tell it what you want, and it will do its damndest to make your application run as fast as it can.

The Myth of Expensive Object Allocation



Are there any other myths that you want to touch on?



One older myth that unfortunately persists is that object allocation is expensive. In the J2SE 1.0 and 1.1 days, object allocation was expensive. But garbage collectors have greatly improved, and the cost now of allocating an object in the Java language is less expensive than in C by a factor of four or five, according to data I've seen. The fast-path object allocation for new objects in Java software is on the order of 10 machine instructions, which requires fewer than 10 cycles on most processors. C can't come close to that. In memory management, Java technology is already significantly faster than C, and yet people incorrectly believe that it's expensive and that developers should preallocate and pool their objects.



Why do people find it so surprising that allocation is faster in Java applications than in C?



They make outmoded assumptions about how garbage collection works. Because software architects have been thinking about garbage collection for 45 years, garbage collection is far more advanced than people realize.

In order to get the appearance of garbage collection in C++, you use reference counting -- which requires overloading a number of operators, adding overhead to many operations -- plus, you still use `malloc/free` for memory management. People assume that garbage collection in the Java language works the same way, but in fact, it does nothing of the sort.

The garbage collector in contemporary JVMs doesn't touch most garbage at all. In the most common collection scenario, the JVM figures out what objects are live and deals with them exclusively -- and most objects die young. So by the time they get to garbage collection, most objects that have been allocated since the last garbage collection are already dead. The garbage collector avoids a lot of work it would have to do if it were doing it one piece at a time. Similarly, the JVM can optimize away many object allocations. These are just a few examples of how moving memory management out of the libraries and into the platform enables huge performance improvements.

Java SE 6: A Compelling Business Proposition

"J2SE 5.0, with all its cool language features, was primarily a developer-focused release. Java SE 6 is



How would you compare the enhancements in J2SE 5.0 and Java SE 6?



J2SE 5.0, with all its cool language features, was primarily a developer-focused release. Java SE 6

more geared for system administrators, making it a very compelling upgrade from a business perspective."

Brian Goetz
Technology Evangelist, Sun
Microsystems

is more geared for system administrators, making it a very compelling upgrade from a business perspective. J2SE 5.0 provided developers with new techniques that make them more productive, but it takes time for developers to become proficient with new features. With Java SE 6, faster performance provides an immediate benefit to businesses.

Another huge and immediate benefit of Java SE 6 are the [monitoring and management improvements](#) that give developers more insight into what's going on in their application. You can dynamically load a monitoring agent into a running application and analyze your application without having to restart it. All these features make the lives of deployers and system administrators easier.

JSR 223: Scripting Integration



Is there anything that developers are missing about Java SE 6?



I believe the "sleeper" feature of Java SE 6 is [JSR 223](#), the scripting integration JSR. It's an official API for Java applications to call out to scripting languages. Scripting languages can either be interpreted, or they can translate directly to Java software by code.

The proof-of-concept implementation is the JavaScript interpreter engine that's shipping with the JVM, based on the Mozilla Rhino implementation of JavaScript technology. You can call back and forth between JavaScript and Java applications and use the Java class libraries in JavaScript. JSR 223 offers a very nice integration between the Java environments and PHP, Ruby, JavaScript, or whatever your favorite scripting language is.

Because of the tight integration, developers are no longer forced into an either-or choice. They don't have to choose between writing in the Java language or in, say, Python. By making it so easy to call back and forth between Java code and scripting code, you get the best of both worlds. You can develop most of an application in Java code and develop pieces in Python, which may enhance productivity because you can use the right tool at the right point.

It also opens the door to a new generation of extensible applications that allow us to ship applications that have extension points that call out to scripts. So instead of customizing applications by configuring some horrible XML configuration file, which is often underdocumented and hard to do, we'll configure and customize applications by running scripts when needed. We'll insert behavior into these applications as opposed to controlling whatever options the environment gives us.

Applications will be more extensible, easier to customize, and easier for developers to deliver, because we won't have to anticipate everything that every customer might want to do with an application. We can simply provide them with an extension mechanism.

This will benefit developers and customers. There's a lot of talk these days about other languages on the JVM, like Scala, which can run on both the JVM and CLR virtual machine, and JRuby, which is Ruby on the JVM. These languages offer productivity advantages in certain programming domains. Integrating scripting languages with Java applications offers the best of both worlds.

* As used on this web site, the terms "Java Virtual Machine" or "JVM" mean a virtual machine for the Java platform.

See Also

[Brian Goetz Home Page](#)
[Publications by Brian Goetz](#)
[Java Concurrency in Practice](#)
[JSR 223: Scripting Integration](#)
[Monitoring and Managing Java SE 6 Platform Applications](#)
[The JVM Tool Interface \(JVM TI\): How VM Agents Work](#)
[Lucene](#)
[Java Developer Forums: Core Concurrency](#)
[JDK 6 Documentation](#)

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

Submit >

copyright © Sun Microsystems, Inc