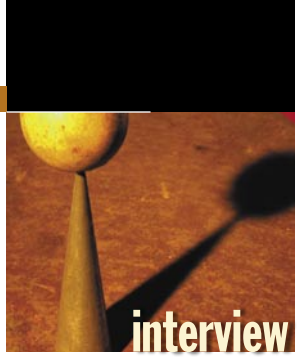



# A Conversation with James Gosling

**James Gosling**

**TALKS ABOUT VIRTUAL MACHINES, SECURITY, AND OF COURSE, JAVA.**

As a teenager, James Gosling came up with an idea for a little interpreter to solve a problem in a data analysis project he was working on at the time. Through the years, as a grad student and at Sun as creator of Java and the Java Virtual Machine, he has used several variations on that solution. "I came up with one answer once, and I have just been repeating it over and over again for a frightening number of years," he says.

Those years included earning a B.Sc. from the University of Calgary, Canada, and a Ph.D. in computer science from Carnegie Mellon University, before joining Sun Microsystems in 1984. At Sun he is best known for creating the original design of Java and implementing its original compiler and virtual machine. Over the years he also wrote the original Unix Emacs, and was the lead engineer of NeWS (network-extensible window system).

Today he is a Sun fellow and chief technology officer of the Developer Products Group. "I have no idea what my job is really," says Gosling, "but I seem to spend most of my time either going to meetings and arguing with people, doing e-mail, and doing a lot of developer evangelism, talking to people all over the world, and generally earning way too many frequent flyer miles."

Gosling discusses his history in interpretive dynamic environments with Eric Allman, chief technology officer and founder of Sendmail Inc. Allman was an early contributor to the Unix effort at the University of California at Berkeley, where he received a master's degree in computer science in 1980. While there, he wrote the syslog, tset, the _me troff macros, and trek, along with Sendmail. Allman designed database user and application interfaces at Britton Lee (later Sharebase). Allman also contributed to a neural network-based speech recognition project at the International Computer Science Institute and was CTO at Sift Inc. He co-authored the "C Advisor" column for *Unix Review* magazine for several years and is a former member of the board of directors of the Usenix Association.

**ERIC ALLMAN** You seem to have had a life theme: this interpretive dynamic environment that includes Emacs—which is arguably nothing more than a Lisp interpreter, syntactic sugar on the outside—NeWS (network-extensible Window System, Oak (Java's predecessor), Java, and so forth. To what extent was this a conscious choice—to say this is what I want to spend my life doing—and how much of it was just that it kept bubbling up all over again?

**JAMES GOSLING** It was pretty much that it seemed to keep bubbling up. It actually goes back further than the


PHOTOGRAPHY BY TOM UPTON

examples you've mentioned. Back when I was a teenager, I worked on this project at the University of Calgary to do data collection for the Isis 2 satellite. I wrote a program for taking data off tapes and producing photographs of them. At the same time I was also working as the local maintainer of a text editor called TECO, which you might remember—with horror, I hope.

**EA** Yes, with horror.

**JG** TECO had this completely trivial little interpreter in it, which I came to know and love. When I was working on this data analysis program, all the people in the project—from the hardware technicians to the physicists—kept asking me to change the program to do this or that or the other thing. I just got fed up one day and had this idea that maybe if I built a little interpreter—kind of like the interpreter that was in TECO—it would quiet them down a little bit and do all the things that they wanted without recompiling this whole big analysis system.

And I did that, and it actually worked. They were writing these little macros in this language that looked an awful lot like TECO's. People were doing everything from writing hardware diagnostics for the satellite to labeling of polar projections of this satellite data. It was just real successful.

I came up with that one answer to that one problem when I was 15 or 16, and I've just reused it ever since.

**EA** You can't keep a good paradigm down. What are you working on now?

**JG** I'm working on a few open source projects that involve RSS (Really Simple Syndication) feeds.

I also spend a lot of time arguing with folks at Sun about what we ought to be doing, which is probably the only actually important thing I do.

**EA** Where did the idea for the Java virtual machine come from?

**JG** Back when I was a grad student at Carnegie Mellon, I had this problem where I needed to have some kind of an architecture-neutral distribution format. We had a bunch of workstations called PERQ machines. The folks who built them were a bunch of hardware guys who didn't want to do software. The only compiler that they could get for free was UCSD (University of California San Diego) Pascal. So they made the hardware interpret UCSD Pascal p-codes.

My thesis advisor, Raj Reddy, asked me to spend the summer trying to figure out how to get the software from these PERQ machines to run on our VAXs.

I started out writing a little hardware emulator, just to understand the p-codes. Then I realized I could actually write a code-generating program that translated from Pascal p-codes to VAX assembly code.

So I wrote a hardware emulator for the PERQ machine that did hardware emulation by translating, and I spent a bunch of time trying to figure out why it was that the translation actually worked. One of the things that I noticed was that the code that I was getting at was actually better than the code that was coming out of the C compiler. I was quite floored by how well it worked, and I spent a bunch of time thinking about what it was about p-code that actually made it work, versus trying to do this for some other instruction set.

Then fast-forward a bunch of years, when I was trying to do the project that Java came out of. I had to do this architecture-neutral distribution format, and then I just went ka-ching! You know, this p-code translator thing would actually just drop in there.

And that formed the core of the design, and then I started talking to people like Peter Deutsch who had worked on the Smalltalk VMs. It just sort of came together.

It's interesting that what people often think of as a pure virtual machine for implementing a language did actually start out conceptually as a hardware emulator.

**EA** One of the things that has plagued virtual environments for some time are security issues. Do you think there's something inherent that creates security problems, or is it just bad implementation?

**JG** It's just bad implementation. In fact, I think virtual environments are a wonderful place to create much higher levels of security. In the Java world, there's the Sandbox. A security manager object manages every one of these Sandboxes, and the security manager watches what the code in the Sandbox does and decides whether or not it's legitimate.

You can identify who the fellow is in the Sandbox and use RSA keys or cryptographic this's and thats. All kinds of tools are available, but once you've got this trustworthy Sandbox in which you can keep unknown pieces of code isolated—so that you can allow them to do the work that they need to do, but make sure that they don't stray outside of it—it's a really powerful aspect of these virtual environments. With Microsoft Word macros, Microsoft had a perfect opportunity to do something sensible, to have a really nice architecture that would allow them to interchange dynamic behavior in a reliable way, but it's a completely irresponsible design.

**EA** If somebody were implementing an interpretive environment, what would you tell them to look at in terms of the security issues? Any specific tricks you've learned over the years?

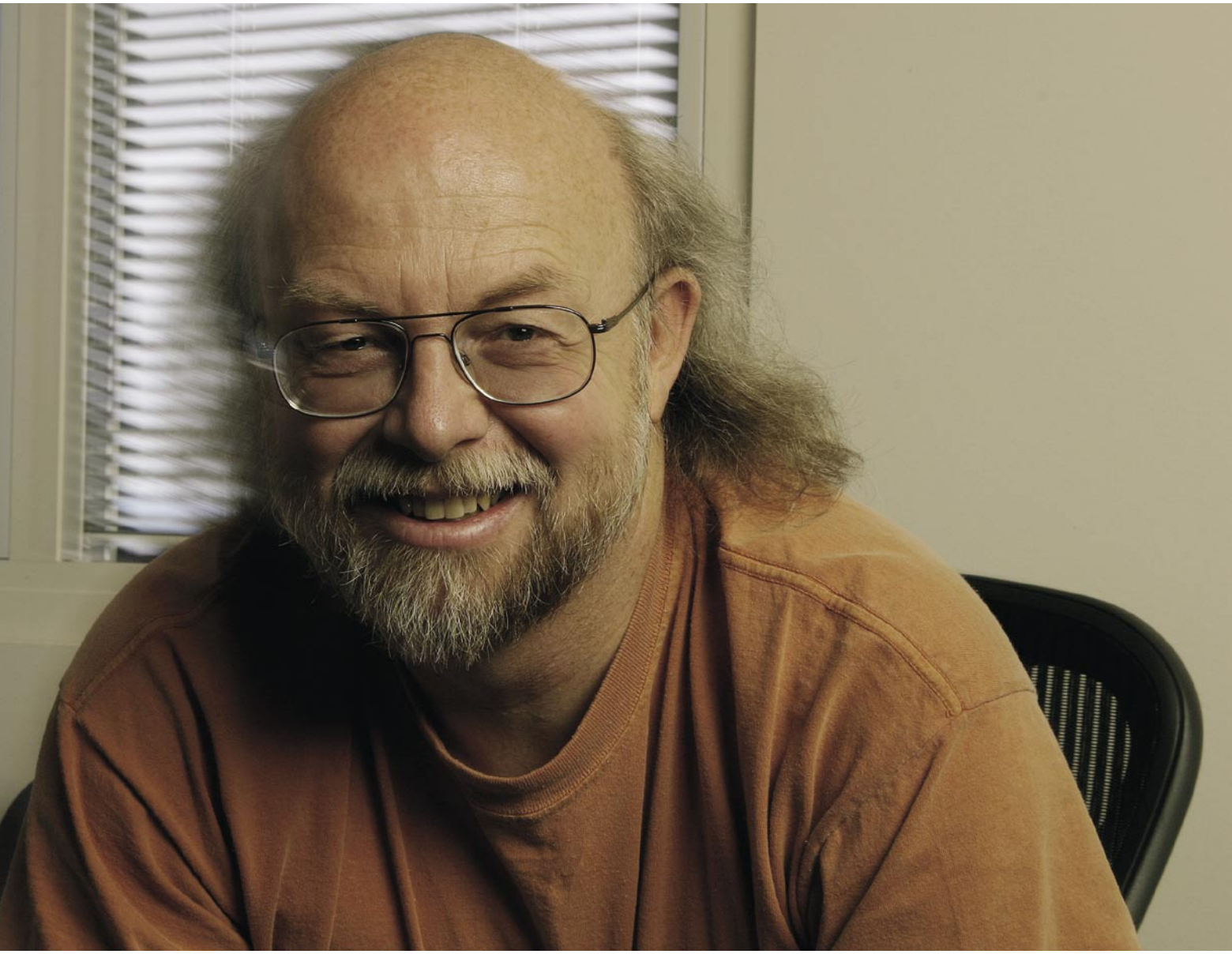
**JG** It's a layered phenomenon. At the lowest level, you have to know that the boundaries around the piece of software are completely known and contained. So, for example, in Java, you can't go outside the bounds of an array. Ever. Period. Turning off array subscripting is not an option.

A lot of work has gone into optimizing compilers. Then you know there's no way that you can get unbounded memory corruption, and that when you hand a piece of memory off to some suspect piece of code, it can't then use it to get outside of that. So if you've got an interpretive language that supports things like C's unbounded pointers, it's very difficult—or at least expensive—to really bound things that it can touch.

Along with that go things like forging the identity of objects. So in Java, you're allowed to cast things only when the cast is actually legal. You can't cast to a super class or to a subclass. You can't take an image and cast it to a string. You can't lie about the identity of an object.

**EA** Unlike C.

**JG** Unlike C, where you can basically lie about anything. In fact, a lot of standard practice in C is all about



lying about the identity of things. But once you've got an environment where you can't lie—you can't wiggle around things—then you can start building mechanisms that find what things can do, and you can then observe things and have some faith that what you're observing is true. You're not seeing somebody who is tricking you by going around the edges of an interface. Then a lot of the stuff layers up from there, and just falls into place, but the higher-level stuff is pointless unless you've got the lower-level integrity.

**EA** You don't want to build on quicksand?

**JG** Exactly. It's like watching somebody build a building downtown—anywhere—they always dig down to bedrock and then go up. If you don't do that first, it's just going to fall over.

**EA** What kind of overhead does the security manager add?

**JG** It generally doesn't add very much. The security manager is asking, "Is this legal?" usually on the level of a system call. It's something like opening a file, which tends to be a pretty expensive operation anyhow, and asking, "Is this legal?" Since the open-file operation is already doing a bunch of independent legality testing, and often the security manager is in a situation like that, it will do a simple string compare, a yes or a no. It tends to be relatively simple.

**EA** Somehow, I was thinking it was something that was watching the PC (program counter).

**JG** One of the big tricks with security managers is that you don't want to be checking the PC on an instruction-by-instruction basis. That's part of the reason that you want to have the tight memory model. One thing that I didn't describe, which is kind of what makes this work, is that in Java, there's this not-very-well-understood, often not-commonly-even-recognized-that-it-exists thing, called the Verifier. The Verifier is applied to a code frag-

ment when it is loaded, and it's what amounts to a simple theorem prover. It proves some really simple theorems about what your code does through static analysis.

**EA** So, in general, what kind of overhead should one expect in an interpretive environment?

**JG** When you talk about a virtual machine, that doesn't necessarily imply that it's actually interpreted. You can go through some kind of a transformational step and actually end up with machine code that is what is actually executing.

Just-in-time compilers can give you performance that is indistinguishable from hand-coded machine code. In some cases, the just-in-time happens significantly beforehand, so that what gets blown into the ROM on a machine is not necessarily bytecode but can actually be the compiled machine code. It's just that there's a transitive nature of trust, right? You've got this untrusted piece of code that comes in, but then you do an analysis by the Verifier to prove various things about it, and then you transform that bytecode into machine code through a compiler that you trust, and you end up with machine code that you can trust. You can actually use that in a lot of these constrained high-performance environments.

There are very few environments these days where you can't afford a just-in-time compiler. Even in most modern cellphones, the older ones are interpretive Java, but the modern ones are actually not. They actually get fairly amazing performance.

**EA** Do dynamic languages live up to that?

**JG** Some of them do, some don't. With a lot of dynamic languages, people don't actually worry too much about performance.

**EA** I'm wondering how much of the Java work was about getting performance up versus getting the thing working

in the first place?

**JG** A bit of both. Being able to get the performance and the security and the reliability up to a really credible level was absolutely a design goal from the very beginning. It took quite awhile to get there.

**EA** Our audience consists of practitioners. Should they be thinking about writing interpreters instead of just writing the code?

**JG** The performance of many situations when people use these interpretive languages is actually not a big deal, mostly because the individual atomic operations in some of these languages are themselves fairly heavyweight. There are things like filling a polygon or drawing a string, and on top of that adding a few instructions to interpret the invocation of an imaging operation, that's relatively low cost. But there are other ways to do it. For example, one technique that people use to get really good performance, rather than writing their own interpreters, is leveraging off one that exists already.

A common one to use is the Java Virtual Machine. There are things like Jython, which is a Python compiler that instead of compiling for the Python virtual machine actually generates Java bytecodes. That way they get a more dynamic-feeling language, but it sits on top of the Java Virtual Machine, and so they get to leverage off all the work that gets done in that optimization.

**EA** Do you know if any other languages have done the same thing?

**JG** There are several hundred of them. People have done things like ML compilers that compile down to Java bytecodes. You can even find Fortran and Lisp compilers that do the same thing.

**EA** I'm curious about your opinion of some of the other languages that in some sense overlap Java. Notably, C#.

**JG** They did such a job of copying Java, it's hard to criticize them. Imitation is the sincerest form of flattery.

**EA** They did have a different philosophy. Java's was "write-once, run anywhere." And the whole initiative of C# was "write in anything run on Windows."

**JG** Yes, and no. You can actually build all kinds of compilers for the JVM, and people have done hundreds of them. We just never mounted a major marketing campaign about that—which is maybe a mistake on our part.

One thing I think was pretty significantly stupid on Microsoft's part is that it built its virtual machine environment so it would support languages like C and C++, and we thought long and hard about doing that. As soon as you support C and C++, you blow away most of the security story.

So C# added these unsafe regions, and the CLR (Common Language Runtime) allows you to do unrestricted pointer operations; and as near as I can tell, the way that the standard Microsoft APIs are built, you have to drop into these unrestricted pointer environments a lot.

As a part of its "support all possible languages," Microsoft basically gave up on security. Microsoft is getting hammered over and over and over again about this, and has been for years, and the company says a lot of good words, but it doesn't actually seem to do anything really significant. It issues a lot of patches. It doesn't actually think about things from the ground up.

Security is one of these things that you don't add by painting it on afterward. Like our metaphor earlier about buildings, if you're going to build an earthquake-safe building, you have to do it from the foundation up. You can't just build any old sloppy building and then apply earthquake safety paint to it.

**EA** I'm curious about a couple of other languages. My favorite language to hate is Perl. It seems like no real thought was given to the language. It kind of grew over the years. So it's just really deeply, deeply ugly.

**JG** Well, yes, but that's part of its charm. It's sort of the resurrection of TECO. Everything in it is about text, so if the data you care about is text, it's pretty nice. I actually like Perl. I can't say I'm real good at it, and I certainly wouldn't want to do any big projects in it. It's sort of a maintenance nightmare. It has very little in the way of structuring and abstraction and all the other things that one would need to do truly industrial-strength software.

But as a language to do narrowly defined mind puzzles, it's pretty entertaining.

**EA** Let me ask about a language that may be a little closer to home: Tcl/Tk?

**JG** That one is pretty nice. It's the child of its environment. It's really very much scripting for C where the data type is all strings, and it works really nicely for that.

**EA** Is there any problem where you would say it's really inappropriate to use some sort of dynamic language, other than perhaps building the operating system? Maybe what we need is the kernel recoded in Java.

**JG** Actually, people have written complete operating systems from the ground up in Java and it has worked pretty well. The question is not so much the language but how that ends up realized. The way Java works, there's a pipeline of bytecodes, validation, translation, whatever, so one can certainly use it in places like operating systems. It's pretty hard to say that from a language level, this is inappropriate, because so much depends on the underlying mechanism.

If you look at the just-in-time compilers and things like the Java HotShot Virtual Machine, they are very much oriented toward maximizing throughput, and maximizing throughput doesn't mean that you instantaneously get the fastest response, but it means that averaged over time, you get the fastest response.

The one area where just-in-time compilers have difficulty is in realtime responsiveness. If you absolutely have to respond to this thing within 10 milliseconds, you can't afford to take a paging fault or a translation fault or a garbage-collection pause.

People have used a lot of these tools in some of these realtime environments. There are a lot of complicated trade-offs. In languages like C, while they meet the performance goals and mostly meet the timeliness goals—although it's actually kind of amazing how ill-understood some of the timing is in C—things like malloc and free can actually be awful.

**EA** People don't understand it. They say, "Let's malloc anything anytime I need it."

**JG** Right, and malloc and free can both be very expensive. In fact, in HotSpot, we do benchmarks comparing our garbage collector against malloc and free, and we're at least an order of magnitude faster—except every now and

then, there's a garbage-collection pause. But if you amortize over a long period of time and include the garbage collection, it will be much faster than malloc and free.

**EA** Do you do incremental garbage collection?

**JG** Yes. We do parallel garbage collections. We actually have multiple garbage collectors. On the high-end systems that run on multiprocessors, what we'll tend to do is run these concurrent garbage collectors that allow the garbage collector to run in parallel with active code, and we'll dedicate some number of processors to garbage collection.

So we'll be metering the storage reclamation rate and the storage consumption rate, and if you start chewing up memory faster, then we'll start allocating more CPUs to the garbage collector. That gives you essentially realtime performance, and it works really nicely—at least on the large-scale multiprocessors. On the smaller-scale unit processor—the typical desktop machine—there are other techniques. While they're incremental, they are not totally pause-free.

**EA** The mantra of Java—at least at one point and probably still is—is "write once run anywhere." In the early days, that was generally felt to not really be true. Everytime you moved to another environment, you still had to port your code. What's your take on where it is now?

**JG** In the early days, that problem was a result of two issues—one was that some of the VMs weren't very well tested, and the other was that Microsoft was aggressively trying to make it false, even though it had contractually agreed otherwise. We ended up dragging them into court, and it got ugly, and we won, and they ended up paying us a big pile of money.

But these days it actually works quite well. The issues where it tends to be a little dicey is where you have applications that have detailed dependencies on specifics of an environment. The way that the APIs are built, you're always supposed to ask questions of the environment—like how big is the screen?

The size of the screen is probably one of the biggest issues that we have to deal with. This is really a cellphone thing. A lot of people who build games really want their game to look good on different cellphone screens. It's 128 pixels on a side versus one that's 200 pixels on a side.

They'll have to redo all of the artwork, for instance. We don't actually try to obscure environmental things. We try to make it easy for you to adapt to them. But if you don't write your program in an adaptable way, then it ain't going to work.

But for desktop software, it works like a charm. I do all of my development on a Macintosh, and I never test on Linux, Solaris, or Windows. I get essentially no complaints about something not working on Linux or Windows.

**EA** So the Mac in particular likes to put preference files in different places than, say, Linux. Is that something you can query where the preferences should go? Or do you have to adapt to that?

**JG** In situations like that, you don't actually have to query where the preferences go because there's actually a standard preferences API, and it takes care of that. In fact, in general, that's what we try to do: have standard APIs that just deal with it for you.

**EA** But there's a limited number of environments or problems you can predict.

**JG** There are a number of them, like file systems and networking. The file system API on Windows and the file system API on Linux are different. But we have a standard API for file systems, and we map to both of them.

So, we sort of make the differences between Windows and Linux go away. You don't actually have to ask questions about whether this is the Windows file system or the Linux file system. You just say open a file, and under the sheet it says, "Well, OK, I'll open it the Linux way or I'll open it the Windows way."

**EA** And that's going to be true of things like threading, one of the most notoriously difficult things to make portable today?

**JG** Since Java uses threading pretty heavily, or at least it's got a lot of facilities in it for doing threading, the applications have tended to use threading very, very heavily. In particular, it's really easy to find server applications that use thousands if not tens of thousands of threads, and we've found that in almost every platform we've ported to, the underlying threading mechanism just falls over dead if you try to create 10,000 threads.

We ended up often creating virtual threads on top of the operating systems' heavy-weight threads. If you go back five years ago, the Linux folks were criticizing Solaris for having this two-layered threading model, which Solaris had entirely for performance. It was just two years ago that Linux came out with a two-layered threading model for exactly the same reasons.

**EA** Let me finish up by just saying that you're kind of what I suspect most of our readers would like to become at some point—these are high-level architects, creative people. Any advice for those folks?

**JG** You know, I have absolutely no idea how I got here. In some sense it feels more like dumb luck and stubbornness.

I've never actually been very good at being mainstream about anything. It took me a long time to learn that listening to my inner self was more important than listening to the orthodoxy. That, and just being obsessive about reading up on stuff. I find that some of the most interesting stuff for me is science fiction, which I like mostly for speculation about how the future could unfold. So for me, a lot of security questions turn into, "Do I want to live in the *Bladerunner* universe?" Naw.

There are also some interesting history of science books—*Guns, Germs, and Steel* (Jared Diamond, W.W. Norton, 1997) or *The Botany of Desire* (Michael Pollan, Random House, 2001)—which are classically wonderful books.

**EA** How do you feel about Bill Joy's "Why the Future Doesn't Need Us?" (*Wired* 8, April 2000; http://www.wired.com/wired/archive/8.04/joy.html). It sounds like you're thinking about the same stuff, maybe a little less passionately, or at least openly, about it.

**JG** I think about a lot of the same stuff, and a lot of the times I actually agree with Bill. I tend to be not quite as dark as Bill on some of these things, and I guess I have more inherent faith in human beings: that once we step up to the brink of disaster, we'll go, "Oh! There's a flaming pit of molten lava there. Maybe I shouldn't actually take the next step." Bill is saying that we're 10 paces away from the pit of molten lava, so let's turn around. Historically, people don't work that way. They don't turn around until they're actually at the edge of the pit of molten lava.

**EA** Unfortunately, they often don't turn around until they notice their feet are really hot.

**JG** But at least they can turn around. They may have some extremity burns. But not many people will march all the way into the pit of molten lava. Often, it's not so much walking in as falling. The big danger is that by the time that you actually recognize the danger, it's already too late. That's the place where I think Bill is very much right-on. Q

**LOVE IT, HATE IT? LET US KNOW**

feedback@acmqueue.com or www.acmqueue.com/forums