Now Chief Java Architect at Google, Bloch previously was a Distinguished Engineer at Sun Microsystems, where he led the design and implementation of the Java Collections Framework introduced in Java 2 and was involved in the design of several language additions in the Java 5 release. He has a BS from Columbia University and a PhD from Carnegie-Mellon University, where he worked on the Camelot distributed transaction processing system, which later became Encina, a product of Transarc, where he was a Senior Systems Designer. He wrote the 2001 Jolt Award–winning book Effective Java and coauthored Java Puzzlers and Java Concurrency in Practice.

As you might expect from someone whose job is to encourage the use of Java at Google, Bloch is a strong advocate of the language. Despite the recent flurry of interest in approaches to concurrency such as Software Transactional Memory or Erlang’s message passing, Bloch thinks Java has “the best approach of any language out there” to concurrency and predicts a resurgence of interest in Java as more and more programmers are forced to deal with programming for machines with multicore CPUs.
Bloch is also a strong advocate of treating programming as API design, and we talked about how that affects his own design process, as well as whether Java has gotten too complex and why picking a programming language is like picking a bar.

Seibel: How did you get into programming?

Bloch: I'm tempted to say it's in the blood. My dad was a chemist at Brookhaven National Lab. When I was in fourth grade, he took a programming course. Back then, of course, machines were mainframes behind glass windows and you handed your deck of cards to the operator. It wasn't hands-on, but I was just thrilled by the idea of these electronic computing machines that would do stuff for you. So I learned a little bit of Fortran from him while he was taking that course.

Seibel: This would have been what year?

Bloch: I think it was 1971. The bug didn't really bite me until a couple years later. And what did it, of course, was timesharing. Long Island had a DECSYSTEM-10, which was shared among all of the schools in Suffolk County. There was another one for Nassau County. It's amazing how many well-known people got their start on one of those two DECSYSTEM-10s.

Once you have interactivity, the bug bites you. I was programming in BASIC, like everybody else back then, from about 1973 through 1976. That's when I got seriously into it. The amazing thing is, I still have programs from back then on Teletype paper—that's the medium that survived—and I look at them and I can sort of see that bits and pieces of my style haven't changed since then.

Seibel: So what was the first interesting program that you remember writing?

Bloch: Well, I remember on July 4th, 1977 writing a version of the classic Twenty Questions game called "animals." The program had a binary tree with yes-or-no questions at the interior nodes and animals at the leaves. When it first encountered a new animal, it "learned" the animal by asking the user for a yes-or-no question to distinguish the new animal from the one it had incorrectly guessed. The binary tree was stored on disk so the program kept getting "smarter" over time.

I remember thinking, "My gosh, this is cool: the program actually learns." That was one sort of aha! moment for me. Another thing I remember was in high school—10th grade, I think—on that DECSYSTEM-10. We weren't allowed to write what would now be called instant-messaging programs—they were thought to be too big a drain on system resources.

Seibel: As they are, in fact, now.

Bloch: Don't get me started. IM ruins my life. No, email ruins my life—IM is just a distraction. Anyway, being the bratty kid that I was, I entered a project into the Long Island Math Fair on what I called "inter-job communication programs." I actually won a prize for it.

Seibel: And you actually wrote the programs?

Bloch: Yes. I wrote the programs, except for one that was contributed by a friend named Thomas De Bellis. The unique thing about Tom's program was that it was written entirely in BASIC. It was line-oriented, and used files to communicate. It wasn't fast or efficient, but it worked! I wrote two, one line-oriented and one character-oriented. I wrote them in MACRO-10, the PDP-10 assembly language. They used a kind of shared memory called the "high segment" for the communication.

I didn't know anything about concurrent programming back then. I remember not really understanding mutexes. But there were communication buffers, and independent agents trying to communicate with each other concurrently. So there were race conditions, and occasionally the program lost a character or two. I wasn't able to figure that out myself as a high-school student.

Seibel: You say that you saw aspects of your current style in your earliest programs. What are the bits that have stayed the same?

Bloch: My attempts to make my programs readable. As Knuth would say, a program is essentially a work of literature. For whatever reason, I realized even back then that a program has to be readable. And that hasn't changed.
Seibel: And what has changed?

Bloch: Well, it's hard to make your programs readable when you're restricted to single-character variable names. So I worry more about variable naming now. Obviously, as you use languages with new features, many things change. And things that you vaguely understood over the years really get slammed home.

For example, don't repeat yourself. I was freer with the copy-and-paste back then than I am now. Now I really try not to do it at all. That's a little bit of an overstatement, but only a little bit. Generally speaking, if I find myself copying and pasting, I think, "What's wrong with this design? How can I fix it?" So that's something that took a little while to get right. Basically I've become harder on myself over the years—that's what it takes to write good programs. You really can't accept bad habits from yourself.

Seibel: If you were going to go back in time and do it all over again, is there anything you wish you had really done differently? The BASIC didn't brain-damage you or anything?

Bloch: No, actually that's a funny thing. I think Dijkstra, God rest his soul, was entirely wrong about that. I know so many really good programmers who got their start programming BASIC because that's what was available to them.

I do think it's good to use lots of languages, though. By the time I was in college, I was programming a whole bunch of them. Each course you would do in a different language. In a numerics course or a science course, you'd use Fortran. If you were taking a programming course back then, it was Pascal or SAIL or Simula or something like that. In an AI course, it was Lisp.

But maybe I should have learned more languages. It's funny—I didn't really get into the object-oriented thing until late in the game. Java was the first object-oriented language I used with any seriousness, in part because I couldn't exactly bring myself to use C++.

Seibel: When was that?

Bloch: Starting in '96 when I joined Sun. I think it would have been good to learn those concepts a little earlier than I did. That said, I don't think all those concepts are good. O0 is a funny thing. It means two things. It means modularity. And modularity is great. But I don't think the O0 people can claim the right to that. You can look at older literature—for example Parnas's information hiding—and see that the notion of a kind of class as an abstraction predates object-oriented programming. And the other thing is inheritance and I consider inheritance a mixed blessing, as many people do by now.

Also I should have exposed myself to more areas, inside and outside of computer science. The more things you learn and the younger you learn them, the better off you are. One thing I've never really done much of is GUI programming and I should have forced myself to do that at some point. But for whatever reason, libraries [have] appealed the most to me over the years, writing the building blocks for other people to use. So I've been doing data structures and algorithms and so forth for decades.

Seibel: Are there any books that every programmer should read?

Bloch: An obvious one, which I have slightly mixed feelings about but I still think everyone should read, is Design Patterns. It gives us a common vocabulary. There are a lot of good ideas in there. On the other hand, there's sort of a mish-mash of styles and languages, and it's beginning to show its age. But I think it's absolutely worth reading.

Another is Elements of Style, which isn't even a programming book. You should read it for two reasons: The first is that a large part of every software engineer's job is writing prose. If you can't write precise, coherent, readable specs, nobody is going to be able to use your stuff. So anything that improves your prose style is good. The second reason is that most of the ideas in that book are also applicable to programs.

My desert-island list is a little bit odd. For example, a book that's terribly important to me is Hacker's Delight, by Hank Warren.

Seibel: That's the bit-twiddling book?
Bloch: Yes. I love bit twiddling and it's relevant to what I do. If you write libraries, compilers, low-level graphics, or crypto, this book is indispensable. Warren has taken what used to be an oral tradition, put it all in one place, and given it the rigorous mathematical treatment that it deserves. I was thrilled when that book was published.

Of course there's Knuth's *The Art of Computer Programming*. In truth, I haven't read the whole series or anything close to it. When I'm working on a particular algorithm, though, I go there to see what he has to say about it. And often it's exactly what I need—it's all in there.

But I simply don't have the capacity and speed to read through all of it, so I'd be lying if I told you I had. An old book that I think is great is *The Elements of Programming Style*, by Kernighan and Plauger. All the examples are in Fortran IV and PL/I, so it's a bit out-of-date. But it's amazing, given the age of the book, the ideas are all still current.

Another old one is Frederick Brooks's *The Mythical Man Month*. It's 40 years old and still as true today as when it was written. And it's just a joy to read. Everyone should read that. The main message of the book is "adding people to a late software project makes it later," and it's still true. But there are a lot of other important things in it. Some of the details are beginning to age, but everyone should read it anyway.

These days, everybody has to learn about concurrency. So *Java Concurrency in Practice* is another good bet. Although it has Java in the title, a lot of the material transcends any particular programming language.

Seibelt: Other than naming your variables better, and cutting and pasting less, is there anything else about how you approach programming that has changed as you gained experience?

Bloch: The older I get, the more I realize it isn't just about making it work; it's about producing an artifact that is readable, maintainable, and efficient. Generally speaking, I find that, contrary to popular belief, the cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast. As they say, it's easier to optimize correct code than to correct optimized code.

Some of the changes in my approach are specific to languages. Every language presents you with a toolkit. You want to use the right tool for the job, and what would be the right tool in one language may not be the right one in another. A trivial example: if you're writing in Java 5, using enums instead of int constants or Booleans can greatly simplify your program and make it safer and more robust.

Seibelt: Given that, can you say anything about how to speed up the process of getting to fluency in a new language?

Bloch: I think it's a lot like spoken languages. One way is by knowing a lot of languages—if you already know Italian and Spanish and you want to learn Portuguese, you're not going to have a very hard time doing it. The more you know, the more you have to draw on.

When you're learning a new language, come in with all that you've learned, but remain open-minded. I know people who have sort of decided, "This is the way that all programs should be written." I won't mention any languages, but some languages, for whatever reason, cause people to get this way. Whenever they go to a new language, they criticize it to the extent it isn't like God's true language, whatever that happens to be. And when they use the new language, they try to program in God's true language to the extent that you can in the new language. Often you're missing what makes a language special if you do that.
It’s like if the only tool you have is a hammer and someone gives you a screwdriver, and you say, “Well, this isn’t a very good hammer but I guess I can hold the blade in my hand and whack with the handle.” You have a crappy hammer when in fact you could have used it as a fine screwdriver. So, a combination of open-mindedness and a willingness to apply everything you already do know. And of course, code, code, code! The more you use the language, the faster you’ll learn it.

Seibel: Why do people get so religious about their computer languages?

Bloch: I don’t know. But when you choose a language, you’re choosing more than a set of technical trade-offs—you’re choosing a community. It’s like choosing a bar. Yes, you want to go to a bar that serves good drinks, but that’s not the most important thing. It’s who hangs out there and what they talk about. And that’s the way you choose computer languages. Over time the community builds up around the language—not only the people, but the software artifacts: tools, libraries, and so forth. That’s one of the reasons that sometimes languages that are, on paper, better than other languages don’t win—because they just haven’t built the right communities around themselves.

Seibel: Java strikes me as interesting in that regard because it has two communities. There’s the implementers and systems programmers—people who worked at JavaSoft or Weblogic or places like that. Then there’s all the people who use Java and app servers and prebuilt frameworks to build business applications. Those are very different bars.

Bloch: There are multiple communities associated with Java and with other programming languages too. When there aren’t, it’s usually a sign that the language is either a niche language or an immature language. As a language grows and prospers, it naturally appeals to a more diverse community. And furthermore, as the amount of investment in a language grows, the value of it grows.

It’s like Metcalfe’s law: the value of a network is proportional to the square of the number of users. The same is true of languages—you get all these people using a language and all of a sudden you’ve got Eclipse, you’ve got FindBugs, you’ve got Guice. Even if Java isn’t the perfect language for you, there are all these incidental benefits to using it, so you form your own community that figures out how to do numeric programming in Java, or whatever kind of programming you want to do.

Seibel: Do you enjoy programming as much as you did when you were a kid?

Bloch: I do, although not necessarily in the same way. Like many kids, I think, to some degree programming was a refuge from aspects of life that I couldn’t handle. And the other thing is, when you’re young you have boundless energy and you can hack for hours and hours on end.

As you get older and have a family and kids and all that, you have other responsibilities, other important things in your life. And yet, there’s still this undeniable high that comes from writing a program, watching the pieces fall into place and coming up with several beautiful lines of code that are readable, fast, and do what you want.

Seibel: Do you ever find that because of your greater awareness that it’s not just enough to get it to work, that there are all these other issues, that it’s almost more daunting?

Bloch: Absolutely. Books too, by the way. I definitely go into avoidance behaviors when starting things. Starting is the hardest part, whether it’s a program or a book or anything else. On the other hand, sometimes you remind yourself, “Come on Josh; you’ve been doing this for three decades now, you know how to do it as well as most other people, so just go for it.” And you just sort of remind yourself that, “Look, pretty much every other time you’ve tried to do this the results have been good, so they’re probably going to be good this time too.”

Seibel: So you just talked about how as your life experience broadens, it can be a distraction, but are there any things, experiences outside of programming, that you feel have made you a better programmer?

Bloch: Oh, absolutely. I think almost everything you do, if you do it well. Ideas transfer from all over the place. One example that comes to mind is, when I wrote my thesis, I did an analysis of a distributed data structure, the replicated sparse memory. And the basic idea that enabled me to do the analysis came from a chemistry course I had taken. It was the notion of a
rate-balance equation: when you have a dynamic equilibrium in a system, you can write equations that say, “Things are entering a certain state at the same rate that they’re leaving it.” I got three simultaneous equations in three variables, solved them, and came up with results that precisely matched the observed behavior of this complicated distributed data structure. This was an idea I stole straight from chemistry and retargeted at computer science.

Many things that you see in life, whether in architecture—the way buildings are constructed, in language—the way that communication occurs, many of these ideas can be retargeted. And, of course, there’s math. Math and programming are pretty darn similar. So keeping your eyes open and being willing to reuse ideas is a good thing.

Seibel: Do you know programmers who are great programmers but who aren’t mathematical or well-educated in math? Is it actually important to have learned calculus and discrete math and all this stuff in order to be a programmer? Or is just more a kind of thinking that you could have even if you hadn’t had that training?

Bloch: I think it’s a kind of thinking that you could have if you hadn’t had that training. But it sure helps. I worked with a guy by the name of madbot, Mike McCloskey. He’s very mathematically inclined but hadn’t taken number theory. He rewrote BigInteger. It used to be a veneer over a C package, and he rewrote it in Java with marching orders to make it run as fast as the C-based version. He actually pulled it off. In doing so he had to learn a heck of a lot of number theory. He couldn’t have done it if he weren’t mathematically inclined, but he wouldn’t have had to learn it if he already knew it.

Seibel: But that was an inherently mathematical problem.

Bloch: You’re right; it’s a terrible example. But I believe that even for problems that aren’t inherently mathematical, the kind of thinking that you learn in math is essential to programming. For instance, inductive proofs are so tied to recursive programming that you can’t really understand one without understanding the other. You may not know the terms base case and induction hypothesis, but you have to understand these concepts if you’re going to write correct recursive programs. So even if the domain is unrelated to math, a programmer who isn’t comfortable with these concepts is going to have a harder time.

You mentioned calculus—I think it’s less important. A funny thing has happened over the years. It used to be just assumed that if you were an educated person who had gone to college you had to know calculus. And there are a lot of beautiful ideas there—it’s nice to be able to get your mind around infinity in that way.

But there’s a discrete and a continuous way to get your mind around infinity. I think that for a programmer it’s more important to have mastered the discrete way. For example, I just mentioned induction proofs. You can prove something true for all integers. It’s kind of magical. You prove it for one integer and you prove that one implies the next and then you’ve proved it for all of them. And I think that is more important for a programmer than, let’s say, understanding the notion of limits.

Luckily we don’t have to make a choice. I think that there’s plenty of room in the curriculum for both. So even if you’re not going to use the calculus as much as you use the discrete mathematics, I think it should still get taught. But I think that the importance of the discrete stuff is greater than that of the continuous.

Seibel: You talked before about how writing prose has many similar characteristics to programming. While mathematics has always been closely associated with computers and programming, I wonder if once you’re talking about developing things like web frameworks or a web application on top of a framework, if it requires skills more related to writing.

Bloch: Yes—earlier you mentioned that there were two distinct communities of Java programmers. The need for math is much greater in the community that writes libraries, compilers, and frameworks. If you write web applications on top of frameworks, you have to understand communication, both verbal and visual. I get infuriated at web sites when they drive me to do the wrong thing. It’s clear that someone just hasn’t thought about how someone approaching this thing will deal with it. So yes, the truth of the matter is that programming is at the confluence of a whole bunch of disciplines. And depending on which ones you excel at, you will be better at writing different applications. But even libraries, compilers, and
frameworks have to be readable and maintainable. I contend that you’ll have a hard time achieving that goal if you aren’t a competent writer.

Seibel: What is your process for designing software? Do you fire up Emacs and start writing code and then move it around until it looks right? Or do you sit down on your couch with a pad of paper?

Bloch: I gave a talk called “How to Design a Good API and Why It Matters” at OOPSLA a couple years ago, and several versions of it are floating around the Web. It does a pretty good job explaining how I go about it.

The most important thing is to know what you’re trying to build: what problem you’re trying to solve. The importance of requirements analysis can’t be overstated. There are people who think, “Oh, yeah, requirements analysis; you go to your customer, you say, ‘What do you need?’ He tells you, and you’re done.”

Nothing could be further from the truth. Not only is it a negotiation but it’s a process of understanding. Many customers won’t tell you a problem; they’ll tell you a solution. A customer might say, for instance, “I need you to add support for the following 17 attributes to this system. Then you have to ask, ‘Why? What are you going to do with the system? How do you expect it to evolve?’” And so on. You go back and forth until you figure out what all the customer really needs the software to do. These are the use cases. Coming up with a good set of use cases is the most important thing you can do at this stage. Once you have that, you have a benchmark against which you can measure any possible solution. It’s OK if you spend a lot of time getting it reasonably close to right, because if you get it wrong, you’re already dead. The rest of the process will be an exercise in futility.

The worst thing that you can do—and I’ve seen this happen—is you get a bunch of smart guys into a room to work for six months and write a 247-page system specification before they really understand what it is they’re trying to build. Because after six months, they’ll have a very precisely specified system that may well be useless. And often they say, “We’ve invested so much in the spec that we have to build it.” So they build the useless system and it never gets used. And that’s horrible. If you don’t have use cases, you build the thing and then you try to do something very simple and you realize that, “Oh my gosh, doing something very simple like taking an XML document and printing it requires pages upon pages of boilerplate code.” And that’s a horrible thing.

So get those use cases and then write a skeletal API. It should be really, really short. The whole thing should, usually, fit on a page. It doesn’t have to be terribly precise. You want declarations for the packages, classes, and methods and, if it’s not clear what they should do, then maybe a one-sentence description for each. But this is not documentation of the quality that you will end up distributing.

The whole idea is to stay agile at this stage, to flesh the API out just enough that you can take the use cases and code them up with this nascent API to see if it’s up to the task. It’s just amazing, there are so many things that are obvious in hindsight but when you’re designing the API, even with the use cases in mind, you get them wrong. Then when you try to code up the use cases you say, “Oh, yeah, this is fundamentally wrong; I have too many classes here; these should be combined, these need to be broken out,” whatever it is. Luckily, your API doc is only a page long, so it’s easy to fix it.

As your confidence in the API increases, then you flesh it out. But the fundamental rule is, write the code that uses the API before you write the code that implements it. Because otherwise you may be wasting your time writing implementation code that won’t get used. In fact, write the code that uses the API before you even flesh out the spec, because otherwise you may be wasting your time writing detailed specs for something that’s fundamentally broken. That’s how I go about designing stuff.

Seibel: And how specific is this to designing things like the Java collections, which are a particular kind of self-contained API?

Bloch: I claim it’s less specific than you might think. Programming of any complexity requires API design because big programs have to be modular, and you have to design the intermodular interfaces.

Good programmers think in terms of pieces that make sense in isolation, for several reasons. One is that you, perhaps inadvertently, end up producing useful, reusable modules. If you write a monolithic system and,
when it gets too big, you tear it into pieces, there will likely be no clear boundaries, and you'll end up with unmaintainable sewage. So I claim that it's simply the best way to program, whether you consider yourself an API designer or not.

That said, the world of programming is very large. If programming for you is writing HTML, it's probably not the best way to program. But I think that for many kinds of programming, it is.

Seibel: So you want a system that's made up of modules that are cohesive and loosely coupled. These days there's at least two views on how you can get to that point. One is to sit down and design these intermodule APIs in advance, the process that you're talking about. And the other is this "simplest thing that could possibly work, refactor mercilessly" approach.

Bloch: I don't think the two are mutually exclusive. In a sense, what I'm talking about is test-first programming and refactoring applied to APIs. How do you test an API? You write use cases to it before you've implemented it. Although I can't run them, I am doing test-first programming: I'm testing the quality of the API, when I code up the use cases to see whether the API is up to the task.

Seibel: So you write the client code to use the API and then look at it and ask, "Is this code I would want to write?"

Bloch: Absolutely. Sometimes you don't even get to the stage where you can look at the client code. You try to write it and you say either, "I cannot do this at all because I forgot this piece of functionality in the API," or, "I can do this but it's going to be so tedious that this was not the right approach."

It doesn't matter how good you are; you can't get an API right until you've tried to code to it. You design something; try to use it; and say, "Oh, this is so wrong." And if you do this before you've wasted time writing all of the layers underneath it, that's a huge win. So what I'm talking about is test-first programming and refactoring the APIs, rather than refactoring the implementation code underneath the APIs.

As far as doing the simplest thing that will work, I'm all for it. The fundamental theorem of API design is, *when in doubt, leave it out.* It should be the simplest thing that is big enough to handle all the use cases that you care about. That doesn't mean "just throw some sloppy code together." There are oodles of aphorisms to this effect. My favorite is one that's commonly misattributed to Thelonious Monk: "Simple ain't easy."

Nobody likes sloppy software. People who say, "Write the simplest thing that could possibly work and refactor mercilessly" aren't saying, "Write sloppy code," and they aren't saying, "Don't do upfront design work." I've talked to Martin Fowler about this. He's a huge believer in thinking about what you're going to do so your system has a reasonable shape and a reasonable structure. What he's saying is, "Don't write 247-page specs before writing a line of code," and I agree.

I do disagree with Martin on one point: I don't think tests are even remotely an acceptable substitute for documentation. Once you're trying to write something that other people can code to, you need precise specs, and the tests should test that the code conforms to those specs.

Seibel: Since you mentioned Fowler, who's written a couple of books on UML, do you ever use UML as a design tool?

Bloch: No. I think it's nice to be able to make diagrams that other people can understand. But honestly I can't even remember which components are supposed to be round or square.

Seibel: Have you ever done full-on literate programming a la Knuth?

Bloch: No. I'm not against it in principle. I just haven't had occasion to do it. The other thing is—how can I put this delicately—I tend not to buy into religions, any religions, whole hog. Whether it's object-oriented programming or functional programming or Christianity or Judaism—I mine them for good ideas but I don't practice them in toto. There are a lot of great ideas in literate programming, but it's not the right bar: there aren't
enough other programmers hanging out there. I could see maybe doing it once as an experiment.

What I do instead is I will cheerfully spend literally hours on identifier names: variable names, method names, and so forth, to make my code readable. If you read some expression using these identifiers and it reads like an English sentence, your program is much more likely to be correct, and much easier to maintain. I think that people who say, “Oh, it’s not worth the time; it’s just the name of a variable,” just don’t get it. You’re not going to produce a maintainable program with that attitude.

Seibel: One way that programs differ from most literature—non-experimental literature anyway—is that there is no one order in which to read a program. How do you read a big program that you didn’t write?

Bloch: Good question. The truth is I really want programs to be well-written. I know a few people with the ability to take an arbitrarily large and poorly written system and wrap themselves in the code till they get a total mental picture of the architecture. It’s a really useful skill, but I’ve never been able to do it.

I want to be able to take small modules, read them, and understand them in isolation. If I’m trying to read a system that’s tightly coupled so I have to read the whole thing in order to understand one part, it’s a nightmare. I have to psych myself up even to attempt to read it, and I have to have access to all the code at the same time. I usually print everything out and sit on the floor surrounded by the printout, writing notes on it.

If I’m reading a well-written piece of code, I try to find a view from 10,000 feet: usually someone, somewhere has written a description of the shape of the entire system. If I can find it, I know what the important modules are, and I read them first, occasionally diving down into lower-level modules to aid my understanding.

Also, although code is written linearly down the page, the execution is not at all linear. If I’m lucky enough to have a piece of code that can be read from top to bottom, great. If not, it’s important that I have access to tools that let me quickly locate methods that are being invoked, classes that are being extended, and so on. This lets me understand key execution paths through the code.

Seibel: Do you ever step through code as a way of understanding it?

Bloch: Absolutely! That is still my chosen method of debugging. Especially for concurrent code—there are too many states that the thing can be in for me to possibly enumerate all of them. I just stare at the code; step through it mentally; think of what invariants must hold at what time. For all of the fancy debugging tools at our disposal, there’s nothing that can match the power of simply stepping through a program, in a debugger or by reading it and mentally executing the code. I’ve found many bugs that way and I use it as part of the writing process.

As I write the program, I say to myself, what is that must be true here? And it’s very important to put those assertions into the code, to preserve them for posterity. If your language lets you do it with an assert construct, use it; if not, put assertions in comments. Either way, the information is too valuable to lose. It’s what you need to understand the program six months down the road, and what your colleague needs to understand the program any time at all.

Seibel: Do you feel like people understand invariants and how to use assertions as well as they ought?

Bloch: No. You probably know that assertions were the first construct that I added to the Java programming language and I’m well aware that they never really became part of the culture. Only a small fraction of Java programmers use them. I don’t exactly know why that is. Talking of mathematics—invariants are very much a mathematical idea.

Seibel: But you don’t have to have a lot of math to be able to understand it.

Bloch: You don’t. But let me just play the devil’s advocate. There’s a certain precision of thinking that comes with doing math. I coached a Math Olympiad team for fourth and fifth graders. This is just the age at which some kids are starting to understand, at some level, the notion of a proof—
that a proposition can be demonstrably, unequivocally true rather than just, “I think it’s true because here are a few examples where it seems to work.”

In order to understand the notion of an invariant, you have to understand the notion of a proof. Unfortunately, there are plenty of adults who don’t. And it’s a style of thinking that is typically taught in mathematics classes.

Seibel: You’d almost wonder if maybe the better forum to teach that kind of thinking would be in programming. If you just taught programming as being about invariants—

Bloch: To a certain extent I agree, but you can go too far in that direction. Then we’re back to Dijkstra. I’m sure you’ve read “On the Cruelty of Really Teaching Computing Science”, which I think is as wrong as it could possibly be. Dijkstra says that you shouldn’t let students even touch a computer until they’ve manipulated symbols, stripped of their true meaning, for a semester. That’s crazy! There’s a joy in telling the computer to do something, and watching it do it. I would not deprive students of that joy. And furthermore, I wouldn’t assume that I could—computers are everywhere. Ten-year-olds are programming.

Seibel: As a Java guy at Google, do you think it could be used more? Leaving aside the force of history and historical choices, if somehow you could wave a magic wand and replace all of the C++ with Java, could that work?

Bloch: Up to a point. Large parts of the system could be written that way, and over time, things are moving in that direction. But for the absolute core of the system—the inner loops of the index servers, for instance—very small gains in performance are worth an awful lot. When you have that many machines running the same piece of code, if you can make it even a few percent faster, then you’ve done something that has real benefits, financially and environmentally. So there is some code that you want to write in assembly language, and what is C but glorified assembly language?

I’m not religious. If it works, great. I wrote C code for 20 years. But it’s much more efficient, in terms of programmers’ time, to use a more modern language that provides better safety, convenience, and expressiveness. In most cases, programmer time is much more valuable than computer time.

But that isn’t necessarily so if you’re running the same program on many, many thousands of machines. So there are some programs that we write where probably using less-safe languages to extract every ounce of performance is worth it. I think for most programs these days the performance of all modern languages is a wash and if anyone tells you that their language is ten times more efficient, they’re probably lying to you.

But in terms of efficiency, in terms of use of engineers’ time, it’s far from a wash. More modern languages, first of all, are exempt from large classes of errors. Second of all, they have marvelous sets of tools which make engineers more efficient. To some degree it’s cultural; it’s what languages people learned in schools. But to some degree I think it’s actually fundamental engineering at work. For example, if a language has a macro processor it’s much harder to write good tools for it. Parsing C++ is a much trickier business than parsing Java.

Google is writing a lot more of its code in Java now than it used to. I don’t know what the numbers are, but if the lines haven’t already crossed, they will soon. So there’s a big difference between how many lines of code do we have in each language versus how many cycles are getting executed in each language. And I think it would be a fool’s errand and not particularly meritorious, either, to try and get the inner loops of the indexing servers written in Java. If you were starting a company to do this sort of thing today, you might write things largely in Java or in some other modern, safe language, and then escape it when you needed to. But we have this engineering infrastructure. Libraries and monitoring facilities and all of that stuff that makes it go. And finally Java is, if not an equal partner in this, it’s reasonably usable within these systems, which is good. When I arrived that wasn’t the case yet.

Companies establish their DNA very early on. It can make them tremendously successful, but it can also make it hard for them to escape when what served them well in the early days doesn’t serve them so well any more. I remember being an intern at IBM Research in Yorktown Heights around 1982, seeing the culture still dominated by batch processing. Even when they were doing timesharing, they talked in terms of virtual card readers and virtual card punches. Everything was still 80-column records. With DEC, it was the timesharing mentality that they never escaped. And I
suppose with Microsoft it's an open question whether they'll be able to move beyond the desktop-PC mentality.

Seibel: And 20 years from now people will be talking about how Google can't get past how to sell ads on the Internet.

Bloch: Absolutely. Anyway, there was this sort of cultural meme at Google that Java is slow and unreliable. And it's obvious where it came from: Blackdown Java on Linux, around 1999, was slow and unreliable. And old ideas die very hard. Although the truth is, Google uses Java for many sorts of business-critical functions, including, by the way, ads.

So at some level they understand that it's neither slow nor unreliable. But the actual search pipeline, which is the most intense in terms of machine cycles, that stuff is all basically C++ and there's an obvious reason having to do with the genesis of the company. And I think that will continue to affect us for quite some time.

Seibel: What are the tools you actually use to program?

Bloch: I knew this was coming; I'm an old fart and I'm not proud of it. The Emacs keystrokes are wired into my brain. And I tend to write smaller programs, libraries and so forth. So I do too much of my coding without modern tools. But I know that modern tools make you a lot more efficient.

I do use IntelliJ for larger stuff, because the rest of my group uses it, but I'm not terribly proficient. It is impressive: I love the static analysis that these tools do for you. I had people from those tools—IntelliJ, Eclipse, NetBeans, and FindBugs—as chapter reviewers on *Java Puzzlers*, so many of the traps and pitfalls in that book are detected automatically by these tools. It think it’s just great.

Seibel: Do you believe you would really be more productive if you took a month to really learn IntelliJ inside out?

Bloch: I do. Modern IDEs are great for large-scale refactorings. Something that Brian Goetz pointed out is that people write much cleaner code now because they do refactorings that they simply wouldn’t have attempted before. They can pretty much count on these tools to propagate changes without changing the behavior of the code.

Seibel: What about other tools?

Bloch: I'm not good with programming tools. I wish I were. The build and source-control tools change more than I would like, and it's hard for me to keep up. So I bother my more tool-savvy colleagues each time I set up a new environment. I say, “How do you do it these days?” They roll their eyes and help me and I use the environment until it doesn't work anymore.

I'm not proud of this. Engineers have things that they're good at and things they're not so good at. There are people who would like to pretend that this isn't so, that engineers are interchangeable, and that everyone can and should be a total generalist. But this ignores the fact that there are people who are stunningly good at certain things and not necessarily so good at other things. If you force them all to do everything, you'll probably make mediocre products.

In particular there are some people who, in Kevin Bourrillion’s words, “lack the empathy gene.” You aren't going to be a good API designer or language designer if you can't put yourself in the shoes of an ordinary programmer trying to use your API or language to get something done. Some people are good API and language designers, though. Then there are people who are stunningly good at the technical aspects of language design where they can say, “Oh, this will make the thing not LALR(1) and you need to tweak it in just such a way.” That's an incredibly useful skill. But it's no substitute for having the empathy gene and knowing you have this awful language that's unusable.

I know other people who are stunningly good at extracting that last percentage of performance. You want to put them in a position where that's what they're doing. They'll be happy and they'll do good stuff for your company. I think you've got to figure out what your engineers are good at and use them for that. So that's my apologia for why I suck at tools. Lame, I know.

Seibel: Let's talk about debugging. What's the worst bug you ever had to track down?
Bloch: One that comes to mind, which was both horrible and amusing, happened when I worked at a company called Transarc, in Pittsburgh, in the early '90s. I committed to do a transactional shared-memory implementation on a very tight schedule. I finished the design and implementation on schedule, and even produced a few reusable components in the process. But I had written a lot of new code in a hurry, which made me nervous.

To test the code, I wrote a monstrous “basher.” It ran lots of transactions, each of which contained nested transactions, recursively up to some maximum nesting depth. Each of the nested transactions would lock and read several elements of a shared array in ascending order and add something to each element, preserving the invariant that the sum of all the elements in the array was zero. Each subtransaction was either committed or aborted—90 percent commits, 10 percent aborts, or whatever. Multiple threads ran these transactions concurrently and beat on the array for a prolonged period. Since it was a shared-memory facility that I was testing, I ran multiple multithreaded bashers concurrently, each in its own process.

At reasonable concurrency levels, the basher passed with flying colors. But when I really cranked up the concurrency, I found that occasionally, just occasionally, the basher would fail its consistency check. I had no idea what was going on. Of course I assumed it was my fault because I had written all of this new code.

I spent a week or so writing painfully thorough unit tests of each component, and all the tests passed. Then I wrote detailed consistency checks for each internal data structure, so I could call the consistency checks after every mutation until a test failed. Finally I caught a low-level consistency check failing—not repeatably, but in a way that allowed me to analyze what was going on. And I came to the inescapable conclusion that my locks weren’t working. I had concurrent read-modify-write sequences taking place in which two transactions locked, read, and wrote the same value and the last write was clobbering the first.

I had written my own lock manager, so of course I suspected it. But the lock manager was passing its unit tests with flying colors. In the end, I determined that what was broken wasn’t the lock manager, but the underlying mutex implementation! This was before the days when operating systems supported threads, so we had to write our own threading package. It turned out that the engineer responsible for the mutex code had accidentally exchanged the labels on the lock and try-lock routines in the assembly code for our Solaris threading implementation. So every time you thought you were calling lock, you were actually calling try-lock, and vice versa. Which means that when there was actual contention—rare in those days—the second thread just sailed into the critical section as if the first thread didn’t have the lock. The funny thing was that that this meant the whole company had been running without mutexes for a couple weeks, and nobody noticed.

There’s a wonderful Knuth quote about testing, quoted by Bentley and McIlroy in their wonderful paper called “Engineering a Sort Function,” about getting yourself in the meanest and nastiest mood that you can. I most certainly did that for this set of tests. But this tickled all of the things that make a bug hard to find. First of all, it had to do with concurrency and it was utterly unreproducible. Second of all, you had some core assumption that turned out to be false. It’s the hallmark of the tyro that they say, “Yeah, well, the language is broken” or, “The system is broken.” But in this case, yes, the bedrock on which I was standing—the mutex—was, in fact, broken.

Seibel: So the bug wasn’t in your code but in the meantime you had written such thorough unit tests for your code that you had no choice but to look outside your code. Do you think there were tests that the author of the mutex code could have, or should have, written that would have found this bug and saved you a week and a half of debugging?

Bloch: I think a good automated unit test of the mutex facility could have saved me from this particular agony, but keep in mind that this was in the early '90s. It never even occurred to me to blame the engineer involved for not writing good enough unit tests. Even today, writing unit tests for concurrency utilities is an art form.

Seibel: We talked a bit before about stepping through code, but what are the actual tools you use for debugging?

Bloch: I'm going to come out sounding a bit Neanderthal, but the most important tools for me are still my eyes and my brain. I print out all the code involved and read it very carefully.
Debuggers are nice and there are times when I would have used a print statement, but instead use a breakpoint. So yes, I use debuggers occasionally, but I don’t feel lost without them, either. So long as I can put print statements in the code, and can read it thoroughly, I can usually find the bugs.

As I said, I use assertions to make sure that complicated invariants are maintained. If invariants are corrupted, I want to know the instant it happens; I want to know what set of actions caused the corruption to take place.

That reminds me of another very difficult-to-find bug. My memory of this one is a bit hazy; either it happened at Transarc or when I was a grad student at CMU, working on the Camelot distributed transaction system. I wasn’t the one who found this one, but it sure made an impression on me.

We had a trace package that allowed code to emit debugging information. Each trace event was tagged with the ID of the thread that emitted it. Occasionally we were getting incorrect thread IDs in the logs, and we had no idea why. We just decided that we could live with the bug for a while. It seemed innocuous enough.

It turned out that the bug wasn’t in the trace package at all: it was much more serious. To find the thread ID, the trace package called into the threading package. To get the thread ID, the threading package used a trick that was fairly common at the time: it looked at some high-order bits of the address of a stack variable. In other words, it took a pointer to a stack variable, shifted it to the right by a fixed distance, and that was the thread ID. This trick depends on the fact that each thread has a fixed-size stack whose size is a well-known power of two.

Seems like a reasonable approach, right? Except that people who didn’t know any better were creating objects on the stack that were, by the standards of the day, very big. Perhaps arrays of 100 elements, each 4k in size—so you’ve got 400k slammed onto your thread stack. You jump right over the stack’s red zone and into the next thread’s stack. Now the thread-ID method misidentifies the thread. Worse, when the thread accesses thread-local variables, it gets the next thread’s values, because the thread ID was used as the key to the thread-local variables.

So what we took to be a minor flaw in the tracing system was actually evidence of a really serious bug. When an event was attributed to thread-43 instead of thread-42, it was because thread-42 was now unintentionally impersonating thread-43, with potentially disastrous consequences.

This is an example of why you need safe languages. This is just not something that anyone should ever have to cope with. I was talking to someone recently at a university who asked me what I thought about the fact that his university wanted to teach C and C++ first and then Java, because they thought that programmers should understand the system “all the way down.”

I think the premise is right but the conclusion is wrong. Yes, students should learn low-level languages. In fact, they should learn assembly language, and even chip architecture. Though chips have turned into these unbelievable complicated beasts where even the chips don’t have good performance models anymore because of the fact that they are such complicated state machines. But they’ll be much better high-level language programmers if they understand what’s going on in the lower layers of the system.

So yes, I think it’s important that you learn all this stuff. But do I think you should start with a low-level language like C? No! Students should not have to deal with buffer overruns, manual memory allocation, and the like in their first exposure to programming.

James Gosling once said to me, discussing the birth of Java, “Occasionally you get to hit the reset button. That’s one of the most marvelous things that can happen.” Usually, you have to maintain compatibility with stuff that’s decades old; rarely, you don’t, and it’s great when that happens. But unfortunately, as you can see with Java, it only takes you a decade until you’re the problem.

Seibel: Since you say that, is Java off in the weeds a little bit? Is it getting more complex faster than it’s getting better?

Bloch: That’s a very difficult question. In particular, the Java 5 changes added far more complexity than we ever intended. I had no understanding of just how much complexity generics and, in particular, wildcards were
going to add to the language. I have to give credit where credit is due—
Graham Hamilton did understand this at the time and I didn't.

The funny thing is, he fought against it for years, trying to keep generics out
of the language. But the notion of variance—the idea behind wildcards—
came into fashion during the years when generics were successfully being
kept out of Java. If they had gone in earlier, without variance, we might have
had a simpler, more tractable language today.

That said, there are real benefits to wildcards. There's a fundamental
impedance mismatch between subtyping and generics, and wildcards go a
long way towards rectifying the mismatch. But at a significant cost in terms
of complexity. There are some people who believe that declaration-site, as
opposed to use-site, variance is a better solution, but I'm not so sure.

The jury is basically still out on anything that hasn't been tested by a huge
quantity of programmers under real-world conditions. Often languages only
succeed in some niche and people say, "Oh, they're great and it's such a pity
they didn't become the successful language in the world." But often there
are reasons they didn't. Hopefully some language that uses declaration­
site variance, like Scala or C# 4.0, will answer this question once and for all.

Seibel: So what was the impetus for adding generics?

Bloch: As is always the case for ideas that prove less wonderful than they
seemed, it was believing our own press sheets. My mental model was, "Hey,
collections are almost all homogeneous—a list of strings, a map from string
to integer, or whatever. Yet by default they are heterogeneous: they're all
collections of objects and you have to cast on the way out and that's
nonsense." Wouldn't it be much better if I could tell the system that this is a
map from strings to integers and it would do the casting for me and it
would catch it at compile time when I tried to do something wrong? It could
catch more errors—it would have higher-level-type information and that
sounds like a good thing.

I thought of generics in the same way I thought about many of the other
language features we added in Java 5—we were simply getting the language
to do for us what we had to do manually before. In some cases I was dead
on: the for-each loop is just great. All it does is hide the complexity of the
iterators or the index variables from you. The code is shorter and the
conceptual surface area is no larger. In a sense, it's even smaller because
we've created this false polymorphism between arrays and other collections
so you can iterate over an ArrayList or an array and not know or care
which you're iterating over.

The main reason this thinking didn't apply to generics is that they represent
a major addition to an already complex type system. Type systems are
delicate, and modifying them can have far-reaching and unpredictable effects
throughout the language.

I think the lesson here is, when you are evolving a mature language you have
to be even more conscious than ever of the power-versus-complexity
trade-off. And the thing is, the complexity is at least quadratic in the number
of features in a language. When you add a feature to an old language you're
often adding a hell of a lot of complexity. When a language is already at or
approaching programmers' ability to understand it, you simply can't add any
more complexity to it without breaking it.

And if you do add complexity to it, will the language simply disappear? No, it
won't. I think C++ was pushed well beyond its complexity threshold and yet
there are a lot of people programming it. But what you do is you force
people to subset it. So almost every shop that I know of that uses C++ says,
"Yes, we're using C++ but we're not doing multiple-implementation
inheritance and we're not using operator overloading." There are just a
bunch of features that you're not going to use because the complexity of
the resulting code is too high. And I don't think it's good when you have to
start doing that. You lose this programmer portability where everyone can
read everyone else's code, which I think is such a good thing.

Seibel: Do you feel like Java would be better off today if you had just left
generics out?

Bloch: I don't know. I still like generics. Generics find bugs in my code for
me. Generics let me take things that used to be in comments and put them
into the code where the compiler can enforce them. On the other hand,
when I look at those crazy parameterized-type-related error messages, and
when I look at generic type declarations like the one I wrote for Enum—
class Enum<E extends Enum<E>-1 think it’s clear that the generics design wasn’t quite mature enough to go in.

We’re all optimists in our profession or we’d be forced to shoot ourselves. So we say, “Oh, yeah, of course we can do this. We’ve known about generics since CLU. This is 25-year-old technology.” These days you hear the same argument applied to closures except it’s 50-year-old technology. “Oh, it’s easy; it doesn’t add any complexity to the language at all.”

Hell yes, it does. But I think many of us have learned from our experience with generics. You shouldn’t add something to a language until you really understand what it’s going to do the conceptual surface area—until you can make a convincing argument that working programmers will be able to use the new feature effectively, and that it will make their lives better.

If you look at how the man on the street has been reacting to generics, we certainly should have done something other than what we did. Does that mean we shouldn’t have done generics at all? No, I don’t think so. I think that generics are actually good. The fundamental argument that most collections are homogeneous, not heterogeneous, so it should be easy to deal with homogeneous collections is true. Furthermore casting is generally a bad thing. Casts can fail and casts don’t make your program beautiful. So I think you should be able to say what kind of collection it is and then it should just automatically be enforced for you. But does that mean you have to suffer with all this complexity that we have today? No. I think we just didn’t take the right cut at it.

Seibel: Was there real user pressure for generics? Were people complaining that the lack of generics was stopping them from writing software?

Bloch: Were real engineers bitching about the lack of generics? I think the unfortunate answer to that question is, no, they weren’t. I think I was guilty of putting in something because it was neat. And because it felt like the right thing to do.

That said, a lot of engineering is from the gut. Had people been telling me to put in foreach? No. They hadn’t been telling me to do that either. But I just knew that it was the right thing to do. And I was right—everybody likes it.

But I think a big sin in our area, in engineering, is doing stuff just because it’s neat, because it’s good engineering, whatever. If you’re not solving real problems for real users—in this case, Java programmers—then you shouldn’t add the feature.

There’s this marvelous talk that James Gosling gave called “The Feel of Java,” in which he said you need three real uses before you put anything in. You don’t put anything in just because it’s neat.

But people just want to put stuff in. What do engineers do? They write code. And if they are writing a library or writing a language, they want to put their stuff in. You need some presence, some guiding voice, to give you something that works well together and has made the right set of trade-offs between what you do and don’t put in. Because there’s simply more stuff that you could put in than you should put in to any given language. Does that mean that any of this stuff is bad? No, it doesn’t. It just means that you make your choices and certain things shouldn’t be mixed.

Seibel: I was reading Java Puzzlers and Effective Java and it struck me that there are a lot of little weird corners for a language that started out so simple.

Bloch: There are weird corners, yes, but that’s just a fact of life; all languages have them. You haven’t seen a book called C Puzzlers. Why not?

Seibel: Because it’s all puzzlers.

Bloch: Yep. It would take up a shelf. In Java, the puzzlers are worth collecting precisely because you think of it as a simple language. Every language has its corner cases and Java has few enough that they’re for the most part fun and interesting.

Seibel: Is there anything that you’ve learned about programming specifically from working on Java and thinking about its design?

Bloch: I’ve learned an awful lot of things. One thing I’ve learned—I wrote about this in the “Nearly All Binary Searches and Mergesorts Are Broken” blog entry—is that even writing small programs correctly is incredibly difficult. We’re just fooling ourselves if we think our programs are, by and
large, free of bugs. They're not. For the most part, we've written programs that are free enough of bugs to approximate the jobs that we want them to do.

I learned that, given how hard it is to write correct programs, we need all the help we can get. So anything that removes potential bugs from our plate is good. That's why I'm very much a believer in static typing and in static analysis—anything that can remove the possibility of a certain class of bugs from our plate is a very good thing. Anything that can make our jobs as programmers easier is a good thing.

My belief in the importance of good API documentation has been reinforced. Javadoc is one of the lesser-appreciated reasons for the success of the platform. Good API documentation has always been a part of Java's culture, and I believe it's because Javadoc has been there from day one.

My natural tendency to believe that simple is good has been reinforced. Over and over I see additions that are more complex proving themselves to be detrimental in the long—or short—run. When I'm designing stuff, I pay close attention to my "complexity meter:" when it starts bumping into the red zone, it's time to redesign stuff.

I've occasionally run into people who just don't believe that, who just say, "Well, you're stupid, Josh, you just don't get it; this is the right way to do it and I'm sorry if you don't understand it." I just don't buy that. I think that if things start getting complicated, there's probably something wrong with them and it's probably time to start looking for an easier way to do it.

There's a brilliant quote by Tony Hoare in his Turing Award speech about how there are two ways to design a system: "One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

The paragraph that follows is equally brilliant, though it isn't as well-known: "The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature. It also requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late."

Seibel: Do you expect that you will change your primary language again in your career or do you think you'll be doing Java until you retire?

Bloch: I don't know. I sort of turned on a dime from C to Java. I programmed in C pretty much exclusively from the time I left grad school until 1996, and then Java exclusively until now. I could certainly see some circumstance under which I would change to another programming language. But I don't know what that language would be. I think it may not exist yet. I think the world is ripe for a new programming language but I also think that the inertia of a platform is so much higher now than it used to be. A modern platform isn't just a language and a few libraries; it's got loads of tools, a virtual machine—it's an enormous thing. The prospect of creating an entire new platform is much more daunting than it ever was before.

I don't know what's coming next. But I'd like to think that if changing my primary language was the correct thing to do, I could still do it. I want to keep my mind open to the possibility. I want to play around more with other languages. I haven't had the time to do that recently, but I want to take the time.

Seibel: What's your short list of ones you want to play with more?

Bloch: I want to try Scala, though I have some doubts as to whether it will be the next big thing. I have great respect for Martin Odersky. I think there a bunch of neat ideas in the language. But I also think it may be too complex and too oriented towards academics to succeed in the world at large. I have no right to say that, though, because I haven't learned it yet.

I should also play with Python. A real old one I want to play with is Scheme. Everybody says it's such a great book. I bought it—that's the first step. But it'll take a while to do, I guess that's my current short list.

Seibel: These days lots of people are worrying about how we're going to write software that takes good advantage of the coming multicore CPUs.
Java is notable as the first mainstream language to provide built-in mechanisms for multithreading; do you feel like Java's approach is viable in a multicore world?

**Bloch:** I'm going to go one step further. I think it is the best approach of any language out there. It's funny because it seems very popular to talk about Java being dead now. I see it as histrionics, basically. But I think that right now the best existing multithreaded building blocks are in Java. I think Java is poised for a little resurgence. I'm not saying it is where we'll be headed for the next 20 years; that it is the best way to take care of these multicores. But I think of what's available today, it's head and shoulders above the competition.

**Seibel:** What do you see as the competition to Java?

**Bloch:** Well, I'm thinking C++ and C#.

**Seibel:** What about things like Erlang or Software Transactional Memory?

**Bloch:** So far as I know, STM doesn't yet exist in a practical form in any mainstream language. If STM proves to be worth its salt, I suspect it will appear in Java at about the same time it appears elsewhere.

Erlang's approach to concurrency is actors, and if they prove to be a big win, they can also be implemented in many languages. As you know, Odersky and company have already implemented them in Scala. I'm not convinced that actors are the best fit for multicore parallelism, but if they are, I suspect that someone will implement them in Java soon enough.

**Seibel:** So Java provides, as you say, building blocks that let you get portable access to threads provided by the OS and then some higher-level constructs with the java.util.concurrent API. But they're still pretty low-level constructs compared to something like Erlang or STM, aren't they?

**Bloch:** I'm not so sure. Some of Java's building blocks are low-level, like AtomicInteger; some are midlevel, like CyclicBarrier; and some are high-level, like ConcurrentHashMap and ThreadPoolExecutor. I believe that STM and actors could both find comfortable homes in Java's "concurrency building blocks" approach when and if people are convinced that they pull their weight.

Some form of transactional memory may become important in the future, perhaps as a building block for use by concurrency library designers. But I don't think STM will succeed as a tool that lets the application programmer stop worrying about locks and live in a beautiful world where threads don't interfere with one another. It's just not going to happen.

There are a bunch of reasons for this. Here's one I learned when I worked in transaction systems. When you try to do automatic locking or optimistic concurrency control based merely on reading and writing at the byte level, you end up with "false contention" between threads: you have physical conflicts that don't correspond to logical conflicts. If you're forced to think about what locks to acquire, you can do your best to ensure that you don't acquire any locks beyond what is required to enforce logical conflicts.

So, for example, if you have two threads, both of which are incrementing a counter, they should be allowed to proceed concurrently. They may be accessing the same piece of memory but they're not conflicting with each other from a logical perspective. If you have one thread that's reading a counter and one that's incrementing it, they're in conflict. But you can have arbitrarily many readers or arbitrarily many incrementers proceeding concurrently. This is the sort of thing that no system that I've seen to date can figure out of its own accord. The counter example may be artificial, but it's not uncommon that physical contention is far more restrictive than logical contention.

Another problem with STM is that there are all manner of operations that can't occur inside a transaction. I/O is the classic example. A third problem is that some STM schemes allow "doomed transactions" to view memory in inconsistent states, with potentially disastrous results. Again, these are problems that we struggled with back when we were building general-purpose distributed transaction systems. They have solutions, but all the solutions I know of add complexity or reduce performance.

Anyway, to the best of my knowledge, STM is still research. I think it's great that people are doing this research. But I simply don't believe in a silver bullet for concurrency, at least for the foreseeable future.
Seibel: OK, different topic: how do you prefer to work with other programmers?

Bloch: I'm actually quite flexible. I love "buddy programming" where you're working with someone else but not at the same keyboard. You're writing different parts of the system—you trade the code back and forth. You don't even have to be in the same hemisphere. Doug Lea and I have worked that way extensively over the years. One of us will write an interface and the other one will say, "Well this is great but this part sucks and I changed it this way."

Eventually we arrive at some interface that we like and I'll implement the nonconcurrent version and he'll implement the concurrent version and as we do that, we'll find out all the things that we did wrong and take another crack at the interface. And we'll read each other's code and he'll say, "Well, you can make this much faster this way," and I'll say, "You're right, Doug, you can." He's very good at making things go fast—he kind of communes with the VM. So that's one style that I like a lot. And that lends itself to remote collaborations.

I do like sitting at the same terminal with someone and working on code, but I haven't written many programs that way from the ground up. Typically it's in the context of a code review where I'll get some code to review and there'll be a lot of changes and I'll say, "Why don't we just sit together at a machine and bash it into shape?" That's good for a whole bunch of reasons. I think it's a great way to teach, to pass knowledge down from one generation of hacker to the next.

I don't like working in total isolation. When I'm writing a program and I come to a tricky design decision, I just have to bounce it off someone else. At every place I've worked, I've had one or more colleagues I could bounce ideas off of. That's critically important for me; I need that feedback.

Seibel: Is that for the feedback you get or just for the chance to talk it through?

Bloch: Both. There's so much craft in what we do—it's often the case that there's no one right solution, or if there is, it's not apparent until you've used it. You have to go from the gut and talking to someone with a different perspective can be very helpful.

I've known people who don't feel this way—who are willing to program in a vacuum. I think it hurts them. You will discover your bugs earlier—you really want to discover problems with a design long before it hits the point of code. So when you're wrestling with different approaches or even different features—should I support this and this or simply that—you just have to bounce it off other people. On the other hand, you can't take what each person says as gospel because you'll get conflicting opinions, and ultimately, you are responsible for your own work.

Seibel: That raises another age-old question—I think Weinberg wrote about this in The Psychology of Computer Programming in the '70s and the XPers talk about it today: should code be "owned" by one person who is the only person who ever touches it or should everyone on a project collectively own all the code so anyone can fiddle with anything?

Bloch: I believe that code ownership can't be denied. In a way, it's like motherhood—you give birth to the code that you write, and especially if it's large, complex, or original, it is yours. If you find yourself working in someone else's code, talk to them before mucking with their code. Especially if you think there's something really wrong with it, because you might be wrong. If you break someone else's code, that's not nice. Of course, it's bad for an organization if a piece of code belongs to exactly one person because if that person leaves the organization, they're high and dry. So it's really important that multiple people learn about each piece of code and are able to work on it. But I think it's unrealistic to expect everyone to own all the code.

This also touches on what we were discussing earlier in terms of areas of expertise. There aren't that many people who can really write bit-twiddling code, so if you find yourself in the bowels of some code that's doing bit twiddling, you should talk to one of the few people at your company who can actually handle that stuff, if you're not one of them. People who do this stuff love it and are willing to spend whole days reducing an instruction sequence by one instruction or proving some identity that speeds up a computation. But it's so easy to break something. And it's so easy to write
something that, let’s say, works well for \(2^{32} - 1\) of the \(2^{32}\) possible inputs. A unit test may or may not test that one value where your new solution doesn’t work. And if it doesn’t and you broke it, you’re the goat.

**Seibel:** Speaking of writing intricate code, I’ve noticed that people who are too smart, in a certain dimension anyway, make the worst code. Because they can actually fit the whole thing in their head they can write these great reams of spaghetti code.

**Bloch:** I agree with you that people who are both smart enough to cope with enormous complexity and lack empathy with the rest of us may fall prey to that. They think, “I can understand this and I can use it, so it has to be good.”

**Seibel:** Is there something intrinsic in programming that’s always going to draw people with that kind of mentality?

**Bloch:** Absolutely. We love brainteasers. But we have to temper this love with the knowledge that we’re solving real problems for real people. And if we don’t do that we are, essentially, whacking off. I think that part of the failure of the first company that I was involved in was due to the fact that we didn’t understand that what we were doing wasn’t pure engineering.

We weren’t really thinking that the most important thing we could do was solve real problems for real customers. The moment you lose sight of that and who your customers are, you’re dead meat. But I do think that it tends to conflict with the sort of people who are attracted to programming, who are the people who love brainteasers. But I think you can have your cake and eat it too. Keep that empathy gene on when you’re designing your APIs, but then, in order to make them run bloody fast, you can freely descend into the puzzle palace.

You’ll have plenty of opportunity to solve brainteasers when designing and optimizing algorithms and data structures, especially concurrent ones. You have to be able to think with mathematical precision about stuff that is quite complex, and you have to be able to come up with creative ways of combining primitives to achieve the desired effect.

But you have to know where you can and should apply that kind of thinking and where it will just produce a system that is unmaintainable or unusable.

**Seibel:** Are the opportunities for doing that kind of programming going away? A lot of this low-level stuff is implemented in the VM that you’re using or the concurrency libraries that you’re using. So for a lot of people, anymore, programming is about gluing stuff together.

**Bloch:** I totally agree. Well, in relative terms it’s diminishing. The percentage of programmers who have to do this is way smaller than it used to be. Back when you bought a machine and it didn’t even have an operating system on it, nevermind a programming language or any ready-written applications, yeah, everybody had to do that.

The world in which most programmers have to do this is vanishing or vanished. But in absolute terms there’s probably as much need as there ever was for that sort of people. We want to have our cake and eat it too—we want to have the advantages of safe languages coupled with the speed of hand-tuned assembly code, so we need people to write these virtual machines and these garbage collectors and design these chips which are themselves basically works of software, albeit realized in hardware.

I think there’s plenty of employment for people who like doing this stuff, but we have to carefully target them. I think if you have people who are pure puzzle solvers you have to couple them with management who can make sure that they are using their skills in the organization’s best interests.

There’s this problem, which is, programming is so much of an intellectual meritocracy and often these people are the smartest people in the organization; therefore they figure they should be allowed to make all the decisions. But merely the fact that they’re the smartest people in the organization doesn’t mean they should be making all the decisions, because intelligence is not a scalar quantity; it’s a vector quantity. And if you lack empathy or emotional intelligence, then you shouldn’t be designing APIs or GUIs or languages.

What we’re doing is an aesthetic pursuit. It involves craftsmanship as well as mathematics and it involves people skills and prose skills—all of these things that we don’t necessarily think of as engineering but without which I don’t
think you'll ever be a really good engineer. So I think it's just something that we have to remind ourselves of. But I think it's one of the most fun jobs on the planet. I think we're really lucky to have grown up at the time that we did when these skills led to these jobs. I don't know what we would have been doing a few generations back.