Java at 30

Gilad Bracha
F5

March 15, 2025

Don't trust anyone over 30.
Jack Weinberg (born 1940)

1 Java and I

I was privileged to work on the Java Language Specification [GJSB05] and the Java Virtual Machine Specification [LYBB13] for almost ten years, between early 1997 and late 2006. During that time, I was probably more familiar with the details of these documents than anyone. What follows are fragments of technical memoir and commentary based on my experience at that time. I have not closely followed Java's subsequent development; after that long decade, I felt the need to move on to lesser things.

My introduction to Java was through a couple of brilliant whitepapers [Gos95, GM95] . The vision they laid out was compelling. At the time, I was at LongView, better known by its code name, Animorphic, a startup working on a high performance Smalltalk [GR83] system.

Smalltalk has colored my view of computing since I first encountered it. It gave me invaluable insight in my work with Java. It showed me what object-oriented programming could and should be, the role of interfaces, of reflection and debugging, of serialization, of tooling, and so on. At the same time it set up Smallltalk as the standard for comparison for other programming systems - a very high bar. Caveat Emptor.

Once we learned of Java at LongView, it was clear it could benefit from the high performance virtual machine (VM) technology we were developing. We decided to temporarily put our Smalltalk work on hold and focus on Java. It seemed hard to believe that a new language could come, seemingly out of nowhere, and take over the world; it had never happened before, nor has it happened since. Yet we realized that we had to be prepared for that eventuality.

Our pivot toward Java paid off. LongView was acquired by Sun, and some of its technology manifested itself as the Hotspot JVM. The credit for that goes to my colleagues Lars Bak, Steffen Grarup, Robert Griesemer, Urs Hölzle and Srdjan Mitrovic.

That is how I found myself working on Java. My initial impression was that the language was done, and that I should focus on building an advanced IDE for it. That was not to be. At the time, Sun had a "Chinese Wall" between itself and tools developers; we would focus on the heart of the system - specifications, VMs, compilers, core libraries - and leave it to an external ecosystem to take care of tooling. In time, the folly of that position became evident, and Sun belatedly jumped into that arena with tools such as NetBeans.

Instead of dreaming about Smalltalk-like environments for Java, my manager, David Bowen, suggested that there were a few bugs related to the language and VM specifications, and maybe I could look into those. It was the best move of my career. Those few bugs turned out to be the start of an unending stream. It catapulted me into a highly visible role to which I will immodestly submit that I was very well suited.

It was inevitable that Java could not live up to all the expectations it engendered. Like a successful politician, the thrill of victory is bound to be succeeded by some measure of disappointment.

What were these disappointments? The confines of this essay are too small to fully account for them, but I must dwell on some, lest I be accused of a thought crime such as positive thinking.

2 A Critique of Pure Java

> There are two kinds of languages - those that everyone complains about and those that aren't used.
> *Bjarne Stroustrup*

> I invented the term 'Object-Oriented', and I can tell you I did not have C++ in mind.
> *Alan Kay*

> I believe that the trade of critic, in literature, music, and the drama, is the most degraded of all trades, and that it has no real value.
> *Mark Twain*

Of course, the point of the quote above is not that there should not be criticism; it is rather to warn us against the professional critics - the theater critic who never wrote a play, the literary critic who never wrote a novel, etc. The best form of criticism is producing your own work and demonstrating a better way. Having done that, I am also happy to criticize. That said, this essay is not the place for a detailed technical critique of the Java platform. Readers who are curious about such details might look to [Bra10a]. I've reserved a special section for self criticism (see section 3).

The Java whitepapers laid out a vision of computing that was clearly influenced by Smalltalk; it is explicitly mentioned several times. In many ways, Java failed to realize that vision, and that accounts for some of the disappointment.

Java lies somewhere between C++ and Smalltalk in the design space. It would have been a much better language if it were closer to Smalltalk. Alas, it would probably have been a much less successful one as well. Programming languages are cultural artifacts, and programming language design is, like architecture, as much art as engineering. A programming language needs to function well, but also to be attuned to the habits and tastes of its users. The two are not always in alignment.

One of the most exciting elements in the whitepapers was the notion of executing code in the web browser. This was not just talk; a project, initially called WebRunner [Mic94] and later renamed HotJava, created a web browser that ran Java code inside of web pages. These were called applets, and were key to the huge wave of interest in Java in the spring of 1995. As discussed above, this was what got my colleagues at LongView to start work on a Java implementation.

Things didn't turn out the way they were supposed to (they seldom do, do they?). Netscape ultimately gave up on using Java in their browser, and opted instead to create a new scripting language.[1] Such was the power of the Java name at that point that they named it JavaScript (and got Sun to agree to this use of the Java brand). The HotJava project was eventually abandoned as well.

This was primarily a technical failure. We were still in the era of 14.4Kbaud modems, and download times were an issue, aggravated by the insistence that applets be downloaded as bytecode, which was more voluminous than source. Latency suffered further from the delays introduced by the JVM's bytecode verifier. The verifier's purpose was to enforce certain safety requirements on the bytecode. It was hailed as an essential security feature for code downloaded over the internet. Today, code is downloaded over the internet all the time (e.g., JavaScript) without anything like verification, so it is safe to say that verification is not essential. Some made this point at the time, but it was a controversial view.

The verifier is one of the real villains of this piece. Both whitepapers have a whole section emphasizing the advantages of Java being interpreted, including making the development process "much more rapid and exploratory." The JVM includes a byte code interpreter, but it does not offer one of the traditional advantages of an interpreted system: instant turnaround for program revisions. It didn't offer even a simple REPL (Read-Eval-Print Loop) to enable an interactive "conversation" with the interpreter, let alone the rich interactive experience of Smalltalk or Lisp machines. The verifier did away with all that, because every change to the program required a slow verification process. The interpreter became mostly inaccessible to the Java programmer, imprisoned in a cell behind the bars of the verifier. As a result, the entire experience of working with Java became batch-oriented, like traditional programming in C, rather than the live experience of Smalltalk, Self [US87, US90] or Lisp. The culture remains so to this day, and the world is poorer for it.

The size and memory footprint of the JVM were also a concern. HotJava

[1] Support for Java in the browser lingered for years, but its use dwindled rapidly.

also suffered from issues of multi-threading and portability. How could this be, when the whitepapers cite these as selling points and tells us that Java could run in 30K of memory?

The Java system drifted away from its client roots. The verifier, already mentioned, was introduced at a late stage, when Java pivoted toward the internet, away from its prior focus on handhelds or set top boxes on a closed video network.

Besides the verifier, there were other contributors to slow startup/high latency. Features like static initializers tended to be abused so that applications did much too much work at startup. Multithreading forced locking of shared state that slowed things down further. All this could be mitigated by techniques like snapshotting, as done in Smalltalk. A relatively slow interpreter like Squeak Smalltalk could bring up applications rapidly using a snapshot, whereas the much faster and more sophisticated JVM could not, because it had to verify and initialize the application from scratch.

In my early days at Sun, a task force focused on improving startup time would be formed prior to almost every release. The idea of using snapshots was brought up, but never taken seriously. There was an interest in addressing the client space, but it was never strong enough to effect real change. This is perhaps unsurprising, given that Sun's business was focused on the enterprise.

Multithreading is another villain in our story. Shared state concurrency is extremely error prone. Cooperative multithreading (aka green threads) is a bit easier to manage than the more general model used in Java. Some forms, like coroutines, are more manageable still. Multithreading introduced a lot of complexity into the implementation as well. Much of the effort of converting Animorphic's prototype into the production Hotspot VM involved the moving away from green threads. The added complexity never goes way, as many aspects of the implementation become much trickier. Examples:

- Class loading [LB98], where shared tables need be accessed "just so".
- The security architecture, [GED03] where a similar issue of shared tables came up.
- Debugging. The original design document for Java core reflection envisaged its use in support of debugging. However, when the work on debugging started in earnest, it became evident that was not feasible, because reflection worked within the process, and tended to interact with multithreading and cause deadlock. So an entirely new reflective system for debugging, JDI, had to be developed.

Complexity often manifests in unforeseen interactions between features or subsystems. You typically need three separate things to go off the rails to get a perfect train wreck. Case in point: serialization and inner classes. In that case, part of the problem was that inner classes were developed on the West coast, while another team on the East coast worked on serialization. Each made assumptions that were valid at the time, while invalidating assumptions the other team was relying on (see [Bra10b] for details).

3 A Generic Mea Culpa

Java generics [BOSW98, THE+04] are quite possibly its least loved feature. As the person most responsible for the design of Java generics, I wish to be clear: they were a mistake. I do not mean that they should have been done differently; I mean they should not have been done at all. Genericity in object-oriented languages is a complex topic, and I am unaware of a completely satisfactory solution, though I believe there are promising possibilities [Bra18].

The design of Java generics was highly controversial, yet I know of no one who argued that we should not add generics to the language at all. The loudest critics argued that generics should be reified - that is, that the type parameters used in generic definitions should be available at run-time. That would have been a monumental disaster because there was no way to make it compatible. More fundamentally, reified generics add massive complexity to the implementation. My experience in other languages such as Dart [Bra15] confirms this view. A less fundamental, yet very real consequence of this complexity would be that it would have derailed the release schedule massively (at least a year, maybe more).

However, the most important argument against reification was one that was in the back of my mind even then, but could not easily be conveyed to most engineers. The time might come when the JVM would have to host other languages, especially dynamically typed languages. I had always hoped that the JVM could be made to host languages like Smalltalk. This was heresy at the time. After all, arguing that I wanted to design major features of Java so that the JVM could be used for other languages, especially my favorite, would be both unethical and unpopular, and would surely not carry the day. And so I was happy to set that aside if we could agree on a good solution that involved reification.

If one did choose to reify type parameters, one should have followed the design of Java arrays. This design is not statically typesafe, yet it is useful and effective. Crucially, it is very easy for programmers to understand. It was also what the designers of the Beta [MMPN93] programming language did. We could do a lot worse than follow their design (most notably, with respect to nested classes, but that is another topic). This solution was precluded, because the consensus (which I agreed to) was that the generics design should endeavour to be statically type safe. It took many years, but eventually research showed this was untrue [AT16]. Thus we have neither static safety nor simplicity.

On the bright side, today the JVM is home to a rich multilingual ecosystem, including versions of Erlang, Ruby, Groovy and others. This would be very difficult if we had chosen reification over erasure. It is ironic that the JVM, very much designed as a monolingual VM has succeeded, where .Net, originally intended as a multilingual system, has failed. Reification forces programmers of dynamic languages to pass type parameters in order to use standard libraries like collections. It is not necessarily trivial to determine what such type parameters might be in such languages, and it is certainly a bad user experience. The fate of IronRuby, IronPython and VisualBasic provides a salutary lesson.

What of wildcards? This too might be seen as a mistake, given its enormous complexity. Experience had shown us that the alternative, using variance, was very problematic as well. Thus, the best thing we could have done was to *just say no to generics*. This realization gradually dawned on me, but it was too late to do anything about it. The company was publicly committed to generics, and a good deal had been invested in the effort. Arguing to cancel them would never work. Discretion being the better part of valor, I kept my peace.

Were I smarter, I would have seen all this in a flash of intuition at the first meeting where generics came up. There was a discussion of what potential new language features should be considered. I argued for closures, but I was told that "our customers aren't asking for it." As if they had asked for Java, or garbage collection. Trapped in the C++ mindset, they were asking for destructors and generics (well, templates). Even if I had been gifted with divine inspiration and fully understood the design space for generics, I doubt if I could have convinced anyone. So here we are, in the best of all possible worlds. We have a multilingual ecosystem on the JVM, alongside a rather damaged Java.

4 Complexity

> If this deep young man expresses himself in terms too deep for me, why, what a singularly deep young man this deep young man must be!
> *Gilbert & Sullivan*

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
> *Antoine de Saint-Exupéry, Airman's Odyssey*

The Java whitepaper [Gos95] lists simplicity as its first goal, and rightly so, but a major theme in the critique above is excess complexity. People will always agree that complexity is bad, but as usual, their revealed preferences differ from their stated ones. People love complexity. They assume that if something is complex, the people who made it are very smart, and so they and their work are admirable. Indeed, the people who designed and built Java are very smart, and on the whole quite admirable, if I say so myself. And yet ...

Consider the phenomenon of Java puzzlers [BG05]. The book is an intriguing collection of confusing anomalies in the Java language and platform. My colleagues not only wrote a book that sold far more copies than the language and VM specifications I had the privilege of co-authoring, but had a successful traveling roadshow based on it. They gave talks to thrilled audiences, showing off what were essentially design failures, like carnival show freaks. In the process they educated their readers and listeners, while giving them joy - which is more than most teachers and speakers achieve.

The truth, at least with respect to programming systems, is that complexity is not a sign of success - it is a sign you have failed. The best ideas are obvious in retrospect, yet very hard to come up with in advance. Simplicity is much

harder to achieve than complexity. Of course, it is very hard for those who have not been through such a design process to appreciate that. It's much easier to grasp that *more is more* than *less is more*. The former is a tautology; the latter looks like an oxymoron, but is really a koan.

Java's great contribution was that it moved us to a simpler world; a world simpler than C++. Its greatest failing is that it is still too complex.

5 Object Orientation

While we are in the pontification stage of this essay, a word about object-orientation. This too is listed as a feature of Java in both whitepapers. What does object-orientation mean?

Richard Gabriel and Guy Steele gave two different perspectives on this question, when they discussed whether objects have failed. Guy went with the idea that object-oriented programming means what is commonly understood by most developers. This is in line with a school of thought that says that language is defined defacto by its users, and not by a set of rules concocted by grammarians. By this definition, objects have been massively successful. Whether you use C++ or Java or JavaScript or Python you are using objects and object-oriented programming. The opposing viewpoint, articulated by Dick Gabriel, is that objects have failed, because objects and object-orientation, as intended by those who pioneered these concepts (in Beta or Smalltalk or Common Lisp) did not wish for anything like current languages, tools and practice. It should be obvious that I agree with Dick on this point.

Java is very much responsible for both the success and failure of objects. It spread the now common understanding of objects far and wide (while improving greatly upon C++). It also killed the vision of object-oriented programming that came primarily from Smalltalk, by occupying that ecosystem and overshadowing anything else that might have grown in that space.

Object orientation is not about inheritance or even classes. It is primarily about interfaces [Coo09], which are Java's best feature, but Java fails to enforce interfaces uniformly across the board.

6 Benevolence, Dictators and Management

A shared thread among several of the technical problems described above is a lack of overall technical supervision and coordination. The original designers were not directly in charge. Without a single mind, or at least a very small number of like-minded individuals, running the show, the vision described in the whitepapers was lost in a sea of technical changes.

Of course, the scale of the Java effort was huge. An entire platform was being conjured up in a great rush. This necessitated a large organization (though it could have been a lot smaller than it was). Most of those involved did not fully grasp the vision of the whitepapers. Few had ever worked with a live

programming system like Self or Smalltalk. Most had at most a brief encounter with REPLs.

It was perhaps eminently sensible to put this effort in the hands of seasoned product engineers, rather than leave it to the more research-oriented language designers. Yet without their day-to-day involvement and creative control, avoidable mistakes were made, and the platform took off in a very different direction.

A few examples of the benefits of having the language designers more closely involved: Any one of them would have known that the language design had an invariant that all new objects are created via constructors, and that serialization was violating that invariant. They would probably know that if you have anonymous nested constructs, they interact with serialization; at the very least, they'd know what both teams were doing, and that they were invalidating each other's assumptions.

Would it have been better to follow Python's Benevolent Dictator for Life model? There are folks in an alternate timeline who can answer that. Personally, the fact that the language was handed off to Sun's product organization gave me a huge opportunity, and I am forever grateful to Dave Bowen for putting me in that position, and to James, Guy and Bill Joy for going along with that. They have always been extremely supportive and gracious.

> If the truths are sufficiently impalatable, our audience is psychically incapable of accepting them and we will be written off as totally unrealistic, hopelessly idealistic, dangerously revolutionary, foolishly gullible or what have you.
> *Edsger Dijkstra, [Dij75]*

References

[AT16] Nada Amin and Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 838–848. ACM, 2016.

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1998.

[Bra10a] Gilad Bracha. Deconstructing Java, April 2010. Slides for Keynote at IBM Programming Languages and Development Environments Seminar.

[Bra10b] Gilad Bracha. Room 101: Serialization killer:, 2010. Blog post.

[Bra15] Gilad Bracha. *The Dart Programming Language*. Addison-Wesley, Boston, Massachusetts, 2015.

[Bra18] Gilad Bracha. Room 101: Reified generics: The search for the cure, 2018. Blog post.

[Coo09] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 557–572, October 2009.

[Dij75] Edsgar Dijkstra. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*. 1975.

[GED03] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 2003.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.

[GM95] James Gosling and Henry McGilton. The Java language environment: A white paper, 1995.

[Gos95] James Gosling. Java: an overview, February 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE7 Edition*. Addison-Wesley, Upper Saddle River, New Jersey, 2013.

[Mic94] Sun Microsystems. Webrunner: What & why, 1994.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proc. of the ACM Symposium on Applied Computing*, March 2004.

[US87] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.

[US90] David Ungar and Randall Smith. SELF: The power of simplicity, 1990. In The SELF papers, compiled by Urs Hölzle.