Java Magazine

Coding, Tools

Java is criminally underhyped

Jackson Roberts

September 8, 2021

[Jackson Roberts published the original version of this blog post in April 2021; he has since updated the post to consider reader comments. —*Ed.*]

It's likely that you read the title of this post and thought, "What is this guy smoking? Java is everywhere!" You're correct. Java still dominates enterprises and runs some of the world's largest mission-critical applications.

But Java's adoption isn't what I'm talking about. I'm talking about its *hype*. I spend a lot of time around inexperienced programmers. What do inexperienced programmers love doing? Getting excited and opinionated about tools such as programming languages. None of the computer science undergrads I meet are hyped about Java, but they should be.

Young and naive developers (myself included) often fall into the trap of fetishizing new languages and tools at the expense of productivity and sanity. Prior to working at Halp (now owned by Atlassian), I had a nearly romantic relationship with back-end TypeScript. I thought the Node.js ecosystem was the coolest thing ever: I loved the idea of transpiled code, live debugging, the massive package library, and even the weird and fragmented build systems. However, when I used it in production and spoke to more-experienced engineers, the magic quickly faded away.

I had an irrational affinity towards the JavaScript ecosystem because it was the hot thing; it had hype. Reality did not live up to my expectations. Today, the wonderful things I expected from JavaScript I am currently enjoying as I gain experience in Java. I feel betrayed that hype did not lead me to Java sooner. Java is fun to write and productive, and it gets an unfair reputation among new developers as a dinosaur.

Ergonomics are what make Java great

This cannot be understated: Java simply feels good to write. A lot of this is due to the craftsmanship JetBrains puts into IntelliJ IDEA. Everything is autocompleted, jump-to-definition is fast, find-usage works well, and refactoring is easy. However, where Java truly shines is the developer experience with third-party libraries.

Dependency-heavy workloads and industry trends. My experience is limited, but I feel the winds have shifted towards liberal usage of external dependencies. The "not invented here" syndrome is out; "not invented there" is in.

JavaScript developers in particular are extremely likely to include third-party libraries, even for trivial operations such as left-padding a number. I don't think the current affinity for third-party dependencies is particularly harmful, but upstream API changes can wreak havoc on untyped JavaScript and Python codebases.

When you consume third-party libraries in Java, you always know *exactly* what types need to be passed to a method. Most importantly, incorrect use of a function will result in red squiggles in your code editor. Given that heavy library usage is in, more people should be excited about Java.

Nominal typing. There are several disadvantages with dynamic/duck/weak/"whatever" typing. When a dependency changes an API method, and your application fails at runtime rather than at build time, that's a problem. When a developer must refer to the implementation of a method to figure out which types to pass it, that's a waste of time. TypeScript and Python type hints solve this problem a bit, but they lack the ability to validate passed types at runtime without extra code.

Type guards are my least favorite TypeScript feature. They're essentially duck typing that you must implement yourself, and you must trust that they're implemented correctly. In my opinion, this is the worst of both worlds. Consider the code in **Figure 1**. There's something about declaring a type *and* having to write validation logic for that type that really rubs me the wrong way. The code in **Figure 1** smells like someone using the wrong tool.

```
interface Dog {
    bark: () => void;
}

/* The developer has to manually implement
a heuristic check for interface adherence!
When they update the interface, they have
to update the type guards too! */
function isDog(pet: object): pet is Dog {
  return (pet as Dog).bark !== undefined;
}
const dog: any = {bark: () => console.log('woof')};

if (isDog(dog)) {
    // TS now knows that objects within this if statement are always type Dog
    // This is because the type guard isDog narrowed down the type to Dog
    dog.bark();
}
```

Figure 1. Code smell

Unlike TypeScript definitions, Java's nominal type systems take a load off the programmer's brain by crystallizing type definitions and guaranteeing type guards by default.

Removal of responsibility for optimization. Java developers can confidently trust the JVM to do what's best. Whether they're implementing a multithreaded application or storing a large amount of data on the heap, they can be confident they won't shoot themselves in the foot with memory management or data races. This advantage is largely missing in C++, which contains a multitude of "footguns."

This ability to trust the JVM is part of Java's ergonomic experience. When developers can worry less about technicalities, they can focus more on the problem at hand.

A holy grail of productivity. How many languages can you think of that meet the following conditions?

- Quality package manager and build system (I love Maven)
- Nominal typing
- A large community
- Hands-off optimization

Java is the only qualifying tool I've used, but let me know if there are others!

[Author's note: As Reddit user Jwosty pointed out, Microsoft's competitor to Java—C#—has all these characteristics. I have never used C# outside the Unity game engine, so I cannot fairly judge it. However, even if you include Mono, C#'s portability seems to be lacking when compared to Java.]

Surprising absence from university curriculum

I attended the University of Colorado Boulder (CU); it's a great school but it's certainly not known for computer science. However, most of its upper-division computer science curriculum is shamelessly stolen from Carnegie Mellon or Stanford, assignments and all. During my time at CU, I used the following programming languages. Java did not make a single appearance.

- **C++.** This language was chosen for all the required core courses, for example, computer systems, operating systems, data structures, and so on. This language is a reasonable choice because it enables direct memory management and the creation of kernel modules, and it presents many challenges and learning opportunities.
- **Python and Julia.** As you might expect, these languages were the darlings of numerical computation and discrete math professors.
- **Scala.** This language was used in principles of programming languages instruction, primarily for its functional programming and pattern matching features. While Scala uses the JVM and interoperates with Java, it provides a very different developer experience than Java.
- **Web languages: HTML, Cascading Style Sheets (CSS), and JavaScript.** These were used only in a single course called "Software Development Methods and Tools," which focused on industry trends.

The delightful scrutability of JVM bytecode

Debugging external libraries, particularly if they exist as a compiled binary, can be exceedingly challenging. This difficulty is compounded further if the library vendor declines to include debugging symbols or a source map. The ability for developers to easily include and inspect library code is one of the reasons for JavaScript's and Python's popularity.

Let's consider the worst-case scenario for working with an external library: The library is behaving in unexpected or undocumented ways and the code is obfuscated.

One way this can happen is due to code minification, which is extremely common in the JavaScript ecosystem. If you use the jump-to-definition capability and land in a minified JavaScript file, you're overwhelmed by a completely incomprehensible blob of syntax.

A similarly difficult situation comes from attempting to debug static libraries. While a few products (such as Hopper) can produce pseudocode from binaries, it's still extremely difficult to interpret.

JetBrains' bytecode decompiler makes this nightmare slightly less bad. While still not ideal compared to source code, it provides the easiest experience for investigating libraries that you don't have the source code to. This is possible because JVM bytecode is organized into classes rather than as a single stack of instructions (like you'd see from optimized `gcc` output).

Reddit user nonbirithm left an insightful comment about how Java propelled Minecraft's popularity by making extensive modding possible (even against the game developer's intentions).

> The amount of vitriol Java receives in the game development space is rather strong. However, whenever people dismiss Java as being a bad language, I can't help but think of Minecraft.
>
> Yes, it's one of the few mainstream titles where Java has been successfully applied, but I believe Minecraft's success was specifically because it was written in Java.
>
> With no JVM, there would have been no Minecraft mods at the level of flexibility that Forge provides, and the game would probably have stagnated in comparison with the sheer amount of community content from the past ten years that's available. Being able to use libraries like ASM that allow you to transform the compiled bytecode in radical ways were possible only because Minecraft targeted a virtual machine. The incredible thing was that this was in spite of the obfuscation Mojang applied to the compiled source. If it was possible to create a flexible mod system at all thanks to the JVM, people were just too motivated for any deterrence to stop them.
>
> For the people who say that Minecraft should have been written in C++ or something from the start, because Java is a mediocre programming language, there's Bedrock Edition. Nobody I know cares about it enough to play it. For all its bloat and performance issues, the benefits of the JVM were simply too convincing.

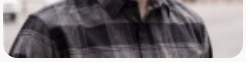
Out of all the machine code formats, the JVM's is easily the most human-understandable. While it may not be relevant often, it's still an interesting boon for the JVM ecosystem.

Conclusion

There is no "one true way" to build applications, but Java doesn't get enough attention particularly among startups and the newbie programming community. Untyped languages are useful tools, but I don't think they should be the default choice for building large applications. If you're a full-stack developer and have never extensively used Java, you'll be pleasantly surprised if you try.

Jackson Roberts

Jackson Roberts (@jacksondotsh) graduated in May 2021 from the University of Colorado



Jackson Roberts (@jacksondotsh) graduated in May 2021 from the University of Colorado Boulder. He currently works at Paperstreet as a software engineer.

Previous Post

The art of long-term support and what LTS means for the Java ecosystem

Donald Smith | 4 min read

Next Post

Java 17 is here: 14 JEPs with exciting new language and JVM features

Alan Zeichick | 8 min read