# Java at 20: How it changed programming forever

Java synthesized sound ideas, repackaging them in a practical format that turned on a generation of coders

By Elliotte Rusty Harold
InfoWorld |
MAY 21, 2015

Remembering what the programming world was like in 1995 is no easy task. Object-oriented programming, for one, was an accepted but seldom practiced paradigm, with much of what passed as so-called object-oriented programs being little more than rebranded C code that used `>>` instead of `printf` and `class` instead of `struct`. The programs we wrote those days routinely dumped core due to pointer arithmetic errors or ran out of memory due to leaks. Source code could barely be ported between different versions of Unix. Running the same binary on different processors and operating systems was crazy talk.

Java changed all that. While platform-dependent, manually allocated, procedural C code will continue to be with us for the next 20 years at least, Java proved this was a choice, not a requirement. For the first time, we began writing real production code in a cross-platform, garbage-collected, object-oriented language; and we liked it ... millions of us. Languages that have come after Java, most notably C#, have had to clear the new higher bar for developer productivity that Java established.

**[ See InfoWorld's full coverage of Java at 20: The programming juggernaut rolls on | The JVM, Java's other big legacy | Its successes, failures, and future | Keep up with hot topics in app dev with InfoWorld's Application Development newsletter. ]**

James Gosling, Mike Sheridan, Patrick Naughton, and the other programmers on Sun's Green Project did not invent most of the important technologies that Java brought into widespread use. Most of the key features they included in what was then known as Oak found its origins elsewhere:

- A base Object class from which all classes descend? Smalltalk.
- [illegible]king at compile time? Ada.
- Multiple interface, single implementation inheritance? Objective-C.
- Inline documentation? CWeb.
- Cross-platform virtual machine and byte code with just-in-time compilation? Smalltalk again, especially Sun's Self dialect.
- Garbage collection? Lisp.
- Primitive types and control structures? C.
- Dual type system with non-object primitive types for performance? C++.

Java did, however, pioneer new territory. Nothing like checked exceptions is present in any other language before or since. Java was also the first language to use Unicode in the native string type and the source code itself.

But Java's core strength was that it was built to be a practical tool for getting work done. It popularized good ideas from earlier languages by repackaging them in a format that was familiar to the average C coder, though (unlike C++ and Objective-C) Java was not a strict superset of C. Indeed it was precisely this willingness to not only add but also remove features that made Java so much simpler and easier to learn than other object-oriented C descendants.

Java did not (and still does not) have `structs`, `unions`, `typedefs`, and `header` files. An object-oriented language not shackled by a requirement to run legacy code didn't need them. Similarly Java wisely omitted ideas that had been tried and found wanting in other languages: multiple implementation inheritance, pointer arithmetic, and operator overloading most noticeably. This good taste at the beginning means that even 20 years later, Java is still relatively free of the "here be dragons" warnings that litter the style guides for its predecessors.

But the rest of the programming world has not stood still. Thousands of programming languages have risen since we first started programming Java, but most never achieved more than a minuscule fraction of collective attention before eventually disappearing. What sold us on Java were applets, small programs running inside of Web pages that

ould interact with the user and do more than display static text, pictures, and forms. e much, but remember -- in 1995, JavaScript and the DOM didn't exist, and an HTML form that talked to a server-side CGI script written in Perl was state of the art.

The irony is that applets never worked very well. They were completely isolated from the content on the page, unable to read or write HTML as JavaScript eventually could. Security constraints prevented applets from interacting with the local file system and third-party network servers. These restrictions made applets suitable for little more than simple games and animations. Even these trivial proofs of concept were hampered by the poor performance of early browser virtual machines. And by the time applets' deficiencies were corrected, browsers and front-end developers had long since passed Java by. Flash, JavaScript, and most recently HTML5 caught our eyes as far more effective platforms for delivering the dynamic Web content Java had promised us but failed to deliver.



Still, applets were what inspired us to work with Java, and what we discovered was a clean language that smoothed out many of the rough edges and pain points we'd been struggling with in alternatives such as C++. Automatic garbage collection alone was worth the price of admission. Applets may have been overhyped and underdelivered, but that didn't mean Java wasn't a damn good language for other problems.

Originally intended as a cross-platform client library, Java found real success in the server space. Servlets, Java Server Pages, and an array of enterprise-focused libraries that were periodically bundled together and rebranded in one confusing acronym or another solved real problems for us and for business. Marketing failures aside, Java achieved near-standard status in IT departments around the world. (Quick: What's the difference between Java 2 Enterprise Edition and Java Platform Enterprise Edition? If you guessed that J2EE is the successor of JEE, you got it exactly backward.) Some of these

enterprise-focused products were on the heavyweight side and inspired open source projects such as Spring, Hibernate, and Tomcat, but these all built on top of the foundation Sun set.

Arguably the single most important contribution of open source to Java and the wider craft of programming is JUnit. Test-driven development (TDD) had been tried earlier with Smalltalk. However, like many other innovations of that language, TDD did not achieve widespread notice and adoption until it became available in Java. When Kent Beck and Erich Gamma released JUnit in 2000, TDD rapidly ascended from an experimental practice of a few programmers to the standard way to develop software in the 21st century. As Martin Fowler has said, "Never in the field of software development was so much owed by so many to so few lines of code," and those few lines of code were written in Java.

Twenty years since its inception, Java is no longer the scrappy upstart. It has become the entrenched incumbent other languages rebel against. Lighter-weight languages like Ruby and Python have made significant inroads into Java's territory, especially in the startup community where speed of development counts for more than robustness and scale -- a trade-off that Java itself took advantage of in the early days when performance of virtual machines severely lagged compiled code.

Java, of course, is not standing still. Oracle continues to incorporate well-proven technologies from other languages such as generics, autoboxing, enumerations, and, most recently, lambda expressions. Many programmers first encountered these ideas in Java. Not every programmer knows Java, but whether they know it or not, every programmer today has been influenced by it.

**Related articles**

- Review: The big 4 Java IDEs compared
- Java forever! 12 keys to Java's enduring dominance
- Java vs. Node.js: An epic battle for developer mind share

*Elliotte Rusty Harold has been programming in Java since version 1.0 alpha 2. His most recent books ail API" and "Java Network Programming (4th edition)," both from*



## YOU MIGHT LIKE