Why Java Sucks

Contents

Introduction

Java sucks, a lot. Here are specific and highly relevant reasons why.

Disclaimer

A lot of people read this article, and think I am on crack or that I am just a grouch. After all, Java really isn't that bad of a language!

The bottom line is this. I've learned more languages than I can recall. I started at 8 years old with C-64 basic, then I learned C-64 assembler, then I learned Borland C anc C++. I did Win32 and MFC programming, even making a few DLLs and whatever the heck COM was back in the day. Then I learned perl, and along with that, BASH and the entire Linux ecosystem. Then I learned Python back in 2001 or so, and never looked back. After that, I poked at Lisp and Scheme and other functional languages, which improved my ability with Python. Over the course of my career, I've programming in pretty much any language you can think of, either as a side project or to fix bugs in a variety of systems. Today, I get paid to program in Python, Ruby, Javascript, Java, PHP, and whatever else comes along.

To me, languages are a box of language features combined with their own weird way of expressing them. I have in my head a variety of tools I'd like to use to accomplish a task, but I am limited by which language I need to program it in.

I should also say that I picked up Java without any formal training in it. I just read the manual and went at it. I was finding features that people who had been programming Java for years at the time weren't even aware existed. Because of my background, I quickly found many of the limiting features of Java.

If you're one of those types who only know 1 or 2 languages, and Java is one of them, you probably won't understand a lot of what I'm talking about. This is because you haven't been exposed to as many programming concepts. You may certainly know all or most of the programming concepts in the languages you know, but you don't know what you don't know because you haven't heard of it and you can't even conceive of it in the context of the languages you already know. Accept that fact, embrace it, and determine to find out what you don't know. I guarantee it will help you in your own career, even if you choose to keep coding in Java.

GC Sucks So Badly, That You Should Avoid Using Memory

An in-memory cache is a very logical, sane thing to do, especially when you understand how much memory you have available and how memory is managed. In C or C++, it is trivial to implement with really good results. In perl, python, and other languages, it is not so bad as long as you understand that memory may not be freed once it is no longer referenced.

In Java, however, in-memory caches should be avoided unless they are very small or very stable. See, garbage collection (GC) has the nasty habit of FREEZING the entire Java process. This was a feature that was added to make Java more "enterprise".

In the end, it just makes it useless.

Imagine a language where you are encouraged to use as little memory as possible and to avoid constructing and destroying objects. *That language is Java.*

I've had a kid Java programmer tell me he never had a problem with Java's memory management. This after spending literally months day-in day-out fighting Java with the top Java programmers at the large company I worked with trying to figure out why it randomly hangs. In the end, we were randomly changing things hoping to uncover the mystery behavior's source cause. And this kid Java programmer says he has never seen Java do that. Upon pressing, he said he *had* seen it do it, but he was able to figure it out. Well, how did you solve it? He mumbled something about configuring some esoteric file I already knew about. What parameters? He couldn't recall. In short, he had no idea what he did to fix it, but he felt like an expert.

Gah! Don't be that kid programmer!

UPDATE: I worked heavily on an android app for a startup. It suffered from glitches, especially randomly pausing for seconds at a time. It took a while to understand why that was. It turns out it is the GC causing the problems.

Java should not be used on devices with limited memory for this reason alone.

Worst of Exceptions and Value Checking

Java's design decision to use both null and exceptions for errors means you have to do all the work in C to detect errors, plus all the work in C++ to properly declare your exceptions.

Python and Perl have it right.

- An exception can be thrown anywhere, at any time.
- Never, ever use return values for error checking. (Perl does a lot worse at this than Python.)
- If you don't like it, catch the exception and do something with it. If you don't know what to do with it, it's not your responsibility.

I would say about 30% of the Java code I write is error checking code and code to handle exceptions because Java really sucks at this.

A workaround that is actually semi-decent is to use RuntimeExceptions, or derivatives thereof, for all your exceptions. Of course, there are libraries that may throw something that isn't a RuntimeException, so you'll have to wrap every call to those libraries to convert their exceptions to a RuntimeException. But at least you won't have to write "throws Exception" everywhere.

One particular example of why, eventually, you'll end up using RuntimeExceptions everywhere is that there are very basic interfaces that do not allow exceptions to be thrown. For instance, iterators and comparators. If you want to throw an exception from there, you have one of three options:

- Change the exception to a RuntimeException and just live your life with RuntimeExceptions.
- Log the exception somehow, and check for it afterwards.
- Change the exception to a RuntimeException, detect it outside of the call, and then change it back to its original form to be rethrown.

Won't cast an int to a long

Try this:

```
public void myFunc(long foo) { ... }
...
myFunc(0);
```

What do you get? An error.

Some people really don't get why this is a bug. As programmers, we have to remember all sorts of things, but one of the things the computer can take care of us is handling the difference between ints and longs.

In Python, for instance, there is no programmer-visible difference between int and long types. If you do an int operation that takes you beyond the limit of ints, Python silently converts it into a long. You can always use longs or ints interchangeably or together.

UPDATE 2015-08: Siegmar Alber informs me that this should no longer be a bug in Java. https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.2

name clash: X and Y have the same erasure

Try this:

```
import java.util.List;

class foo {
    public void func(List<Long> listOfLongs) { }
    public void func(List<String> listOfStrings) { }
}
```

What do you get?

```
name clash: func(java.util.List<java.lang.Long>) and func(java.util.List<java.lang.String>) have the same erasu
```

What does this imply? If you intend to use generics (comparable to C++ templates), *forget it*. You'll need different function names.

incompatible types

```
   [javac] XYZ.java:141: incompatible types
   [javac] found   : java.util.LinkedList<java.util.LinkedList<ABC>>
   [javac] required: java.util.List<java.util.List<ABC>>
   [javac]         return DEF;
   [javac]                ^
```

Apparently, interfaces are only good 1 level deep. Thus, any reasonably complicated data type makes interfaces useless.

static methods and members in a class

The reason why the singleton design pattern is so common in Java is because Java cannot handle static class members properly. It's simply absurd that you cannot override a static method or member in a derived class.

The take-away is that static functions are useless, so don't use them. Use Singletons instead.

Of course, singletons are simply a hack to get global variables. Other languages simply have global variables. Entire frameworks are written just to provide decent global variables in Java. That should be a hint that something is terribly wrong with the language.

The "class" type is an afterthought

If the "class" type behaved like any other object in Java, as it does in most OO systems (except C++), then we wouldn't have to use so many design patterns to work around Java's built-in limitations.

No control on member access

Java simply has no control over public members, unlike most other OO languages (except C++). That's why you have to write the following a billion times:

```
public Long getMemberName() { return memberName; }
```

Because, if, down the road, you decide you want to do something even remotely interesting with 'memberName', you're going to have to call a method to do it. So might as well write it now.

RSI

My wrists hurt when I write Java, even though I use ViM and type as little as possible.

"USE AN IDE!" you shout at me.

"Why are we writing programming languages that are not intended for human consumption?" I respond. Haven't we ALREADY PROVED that the CPU can do all the work in making things easy for us, the user, A LONG TIME AGO?!?

But apparently, we're supposed to live with one of the worst UI's in the world: the Java programming language.

Seriously, computers can take care of the work, so let them do it. Use a modern programming language that doesn't require an IDE.

Camel Case

One of my pet peeves with camel case is the following. Try to change "varName" to "memberName".

```
private Long varName;
Long getVarName() { return varName; }
void setVarName(Long newVarName) { varName = newVarName; }
```

You have to do two substitutions: once for varName -> memberName, and a second time for VarName -> MemberName.

If you used underscore case, then you'd have the following:

```
private Long var_name;
Long get_var_name() { return var_name; }
void set_var_name(Long new_var_name) { var_name = new_var_name; }
```

Now, it's a simple matter of a single substitution: var_name -> member_name.

Of course, I wouldn't have to do so much search-and-replace if there wasn't so much code duplication everywhere in Java. Conciseness is nice.

Class-centric

Whenever someone designs something in Java, particularly people that grew up on Java and C++, they have to think of everything as a class. They do not understand that functions are "things", too.

The worst example of this are classes that have exactly one method as part of its interface. This is no different than a function.

The justifications people make for this design pattern is absurd.

The cost in development time is atrocious.

Will people please recognize the lowly function as a first class citizen in the world of programming?

Iterators SUCK!

In Python, iterators have only one method: next(). The beauty of this solution is that you never have to implement a "hasNext()" method. Simply call "next()" and see if it throws an exception saying there are no more.

If you add the "hasNext()" method, then you have two entry points to iteration: hasNext() or next(). It turns out that they both need to do the same thing, so you abstract out the generation of the next item into a "generateNext()" method.

```
public class MyIterator extends Iterator<T> {
    private T theNext;

    private void generateNext() throws NoMoreElementsException {
        // generate the next value
        // store it in 'theNext'
        // throw an exception if none ...
    }

    public boolean hasNext() {
        if (null != theNext) return true;
        try { generateNext(); return true;}
        catch (NoMoreElementsException e) {
            return false;
        }
   }

   public T next() throws NoMoreElementsException {
       if (null == theNext) {
           generateNext();
       }
       T next = theNext;
       theNext = null;
       return next;
   }
}
```

(The above isn't perfect. The problem with the above is that you can't iterate across 'null' value, so you have to have a additional attributes to determine whether there are any more values, whether there is a next value, in addition to what it is.)

Notice, reader, that next() and hasNext() are exactly the same for any iterator in Java. Only generateNext() will vary in the slightest.

Notice, also, that hasNext() is really pointless. All it does is catch the NoMoreElementsException and tells you that it did. You can, and really should, do that for yourself.

Notice, also, that if you wanted to throw an exception from generateNext(), that the iterator interface in Java is completely incapable of handling such a thing.

Python totally kicks Java's butt in this area, and it's not even funny.

foreach doesn't take iterators

Enough said.

Function Pointers -- Missing

One of these days, I am going to write a class that looks like this:

```
public class Function {
    public Function(name, parameters, body) { ... }
    public ReturnValue execute(values...) { ... }
}
```

ReturnValue is either going to have the result or an exception.

Then I am going to implement my entire program on top of this.

Those of you who know Lisp see the joke for what it is. Except it's not a joke. If you want to have a reference to a pointer, you have to create a class or an instance of a class and pass that around.

Constructors can't call each other

So you have something like this:

```
class Foo {
    private final int bar;
    public Foo(int theBar) {
        bar = theBar;
    }
    public Foo() {
        bar = 1;
    }
}
```

Now, this is a pretty trivial example. I want you to imagine that the "bar =" line above is actually some complicated calculation. The second constructor is really a different way to initialize bar using the same calculation but with some slightly different parameters.

The bottom line is you *can't do this*, at least not in any sane way. You have to copy the code from the first constructor to the second constructor, turning Java into a language where certain bits of code cannot be abstracted away.

Two possible solutions:

First, we might imagine that we can dispatch one constructor to another. This is what Python allows, mostly because it is so laissez-faire about everything. This would be the ideal solution. You have one "master" constructor that does all the smart stuff, and other constructors that would dispatch to it. Of course, this is impossible.

Second, you might imagine a third method, a private method, that would do the actual constructor work. The real constructors would just dispatch to this. Great idea, unless you have "final" attributes as in the example above. You simple can't assign to final attributes unless you are in the constructor itself.

Note: See this thread (http://stackoverflow.com/questions/285177/how-do-i-call-one-constructor-from-another-in-java) for how you can (literally) call one constructor for another. Seems simple, right? Just use "this()". Well, it's not, and it's not clear why. I mean, the whole reason you would want to have a different constructor is because you do something non-trivial before you pass control onto the simpler and more basic and common constructor.

Methods With the Same Name as Constructors

What's wrong with this?

```
public class Foo {
    public void Foo() { }
}
```

What, you can't see it? Why, nothing's wrong, of course. The compiler accepts this just fine, creating a new class Foo with a single method of the same name.

What? You thought that was the constructor? Ha ha, joke's on you. It's not the constructor.

Run-time Dispatch: Fantasy

Let's say you have a class defined as:

```
public class Foo {
    void bar(Baz b) { }
    void bar(Boo b) { }
}
```

This has two methods of the same name but different behavior depending on the type of parameter passed. This is, of course, fairly common.

But what if Boo is a sub-class of Baz? Which bar does the following code call?

```
...
Foo f = new Foo();
Baz b = new Boo();
f.bar(b)
...
```

Before you give your answer, think:

1. What is the most reasonable answer?
2. What features of Java are reasonable and what features are unreasonable?
3. Does this make Java a typically reasonable or unreasonable language?

Now, points 1 and 3 about should tell you that the answer is that it calls "void bar(Baz b)". Which is, of course, the most unreasonable answer since Java is unreasonable. Or, if you prefer, the reasonable answer since Java is a reasonable language in your universe.

No Globals means Frameworks

Java lacks globals. And because it lacks globals, every useful Java project requires a framework that gives you globals. So, in order to write the simplest app, you have to bolt-on thousands and thousands of lines of code and massive complexity just to get globals.

import is Useless

In Java, you have to name all of your classes uniquely, despite their namespace. Why? Because if you don't, you can't import them or else they will clash with another namespace's classes.

In Python, you have:

```
from foo import bar as baz
```

In Java, you only have:

```
from foo import bar
```

Which means if something.something.something.bar and something.something.something.else.bar need to be used in the same package, you're out of luck. You have to write "something.something.something.bar" ad infinitum.

No List Literals

This gets me again and again.

I want a list of integers:

```
[1, 1, 2, 3, 5, 8, 13]
```

But in order to do that, I have to write:

```
LinkedList<Integer> l = new LinkedList();
l.add(1);
l.add(1);
l.add(2);
l.add(3);
l.add(5);
l.add(8);
l.add(13);
```

The obvious way doesn't compile:

```
// Syntax error
LinkedList<Integer> l = new LinkedList({1,1,2,3,5,8,13})
```

Update 1: You can use doubly curlies to do literals. See http://www.c2.com/cgi/wiki?DoubleBraceInitialization HT: Josh Goldberg

Update 2: No, you really shouldn't ever use double curlies for anything. See http://blog.jooq.org/2014/12/08/dont-be-clever-the-double-curly-braces-anti-pattern/ HT:Anonymous source. Commentary: Name ANY other language that introduces a memory leak when you just want to use list literals

Inner classes Don't Work

Inner classes sound like a good idea. Until you try to use them.

Java is So Hard People Prefer to Write Code in XML, Jython, Scala, and Clojure

One of the things that continually amazes me is how many people write code in XML to avoid Java. A lot of the frameworks is really a way to allow you to write code of one form or another in Java.

In other languages, such as Python, it is rare to see people write code in anything but Python. For instance, in the Pylons framework, there is a config file, but this is the exception and not the rule. Even then, the config file is there because it makes the job of configuring a Pylons app *easier*, not *possible*. That is, you want to set the database connection string or change where files are logged. Even then, the developers freely admit that they are trying to open up the app to people who refuse to learn Python but who don't mind writing Windows-style config files.

This is also the reason why Jython, Scala, and Clojure are rapidly becoming popular. Just wait until someone tells them that the JVM gives you NOTHING.

Speaking of the JVM...

(Written about 2015)

I'm going to share a funny story. Well, it's not funny. It's sad.

This year (2015), someone came to visit our company to hawk their framework of the day. One of our developers was super-excited because he loves Java and wants to use it for everything. Except he really doesn't love Java, he likes this particular framework which REPLACES THE ENTIRE LANGUAGE. Now, I don't want to complain about the language, but it's interesting that you can write an entire language in Java and somehow keep all of the warts Java has. That takes some talent. Normally, when you write a programming language, you do so to get rid of the warts, but I'm getting distracted.

So the sales people ask the team what questions they have. We all go around the room, and finally I get my turn.

"Why JVM?"

Their faces turned red. They also turned white.

You see, in today's world, the JVM gives you absolutely nothing. It is much slower compared to Javascript running on v8. I mean, node.js is eating JVM's lunch. And we have things like PyPy and LLVM and much more nowadays making the entire idea of a cross-platform VM less and less appealing. And even Python is moving, as a community, towards a compiled, JIT language, bit by bit, and in some cases, Python beats Java (executed in the right way, of course.)

Our engineer made up some excuses about how many libraries there are and such, but really, there is no answer. They knew it. Java's time is up. It really is.

I remember back in the day when the perl community was convinced they would stay the #1 programming language for the web because they had so many libraries. Their hubris was their downfall. Any language that is any good will have extensive libraries.

So, "Why JVM?" What does it *really* give you? Performance? Even JS can beat Java now. And "enterprise" game developers are still writing C++ code. Memory management? Don't make me laugh. Cross-platform? Not an issue nowadays, especially nowadays where LLVM and PyPy and so much more exists. Scaleability? If you're not fast, and you can't manage memory well, how can you scale better than languages that are fast and can manage memory well?

UPDATE 2016-12-20: I realize that people don't really understand what I'm talking about above, so let me try to make it clearer. TL;DR: We're not in the 90's anymore. Computing has fundamentally changed.

In 2008, Google released Chrome with the V8 Engine (https://en.wikipedia.org/wiki/V8_%28JavaScript_engine%29). The concept of a JIT wasn't new, but applying it to Javascript meant that Javascript could run at comparable speeds to C/C++ code. The way JIT works is it takes Javascript code and compiles it down to the machine language that your processor uses. Obviously, it is adapted to different architectures and processors. Again, this concept isn't new, people in the Lisp/Scheme world have been doing it for a very long time, it's just that they never got the exposure that Chrome did.

Because Javascript is now fast (all the other browsers have since adopted similar changes to keep up), people can write thousands and thousands of lines of Javascript code and still have a usable site. Before, you could only sprinkle, at best, a few hundred lines of carefully optimized Javascript code in the browser.

The JIT in Javascript is so good that people are working on something called asm.js (https://en.wikipedia.org/wiki/Asm.js). This would allow ``any`` program to run in the browser at native speeds. We have companies investigating moving their entire video game platform into the browser. Remember, Javascript has strict rules about how to manage memory and such, so they're writing ``safe`` code that runs at CPU speeds.

You who are stuck in the world of "The JVM is the BEST!!" have no idea what just happened. When people realized what JIT meant and how it works, they realized that VMs are now useless. In the Python community, there is a virtual machine that interprets Python bytecode, but the writing is already on the wall: LLVM and PyPy are making CPython obsolete as they demonstrate that you can write Python and have code that runs as fast or even faster than if you wrote it in C/C++.

My question, "Why JVM?" rightfully made them squirm. There is literally no reason why anyone would ever choose a VM over a JIT compiler if the JIT compiler exists and is in a production-ready state. And projects like LLVM which intend to write JIT compilers for every language imaginable and every architecture imaginable mean that the JVM is now a useless toy, a relic of the 90s. My friend's response "We have lots of libraries!" is a cop-out. It didn't save Perl. It won't save any language. Libraries can be written, and the better the language the quicker and more extensive and useful libraries will be written.

Some people say "JVM does JIT too!" At which point, I seriously question if they even know what they are talking about. With JIT, the JVM is pointless. All programs that can do JIT condense down to the minimum set of machine instructions. Once you've unlocked JIT, you're competing with the best C compilers out there. So at best, the JVM JIT will be on par with any other language's JIT. You simply can't get better than native-compiled machine code with full optimizations.

Now that the discussion about performance is moot, really, the only factor that can differentiate languages is how long it takes to develop a correct program that can be optimized by the JIT.

Side note: If you want to see the future of programming, learn Lisp and Scheme.

The JIT makes JVM pointless!?

OK, I admit, if you are using Java bytecode as a code obfuscation technique, then maybe the JVM isn't completely pointless, although code obfuscation seems to be pointless.

My point is this: VMs exist because writing code that runs on any instruction set architecture is impossible. You have to code up something and then compile it for each architecture it will run in. This is really hard to do, and it's quite a burden for the programmer, so it's nice to only target a virtual architecture and then have someone else make the various virtual machines to run on the different architectures. In short, it's abstracting out the architecture. Back when Java was new, people actually thought that x86 would disappear and we'd have new architectures that were stack-based. (Don't feel bad if you were one of them. The Lisp people thought a similar thing.)

Addding a JIT to the VM means you don't need the VM to handle the abstraction anymore. Once you have a JIT working for each of the platforms, you're done.

Concurrency Done Wrong

(Added 2016-12-20)

A lot of people are commenting about how awesome Java's concurrency implementation is. Let me tell you why that is not a good feature.

A long time ago, you wrote programs that would take complete control of the computer. Let's call this "serial" programming.

Eventually, people figured out that serial programming wasn't good enough. You needed a way to transfer control from one serial program to another so you can have more than one program running at the same time. But keep in mind, by "at the same time", I don't mean they are actually doing anything at the same time. The CPU will work on one, then it will work on the other, and then it will work on the other, etc... so only one program is running at a time.

This is where things like the Unix environment appeared.

In such an environment, if you want concurrency, you write code that forks new processes that run different bits of code. If you want the bits of code to communicate with each other, you use one of the many IPC techniques available, such as shared memory or pipes or files.

Now, when you have two processes accessing the same resource, you need some kind of control. For instance, let's say I had a file with all of the bank accounts and their balances. If someone makes a deposit at the exact moment someone is making a withdrawal from the same account, and both processes read the same initial balance, apply their incremental change, and then set the result, you're going to get the wrong answer every time. Instead, you need to make it so that only one process can access the balance at a time. This is why you have locks and semaphores and things like that.

Someone came up with the idea of threading. The concept of the thread was that you automatically share memory and code but you have multiple processes running at the same time. Writing threaded programs is really hard, even harder than writing multi-process shared memory programs, because you accidentally end up sharing variables that you didn't think you were sharing and next thing you know your program is ruined and no one can figure out why nor reproduce the result.

Java took the idea of threading and said they will fix it by eliminating globals (so nothing really gets shared implicitly) and doing some other things like putting locks on everything and such. Java tried really, really hard to get threading right.

At the same time, you had languages like Python that said, "Screw it! One lock for everything, only one bit of python code can run at a time!" and that seemed to make a lot of the problems disappear without making the language more complex.

Now, let me tell you about the evolution of your program as you scale.

At first, you can generally get away with writing a serial program, and it will work, up until you realize you need to do more work at once. At that point, you have two choices:

1. Rewrite the program so multiple copies of it can run in a single process with threads

2. Rewrite the program so multiple copies of it can run in multiple processes.

People often reach for #1, because, threads are cool, right?

However, the correct answer is always to reach for #2. The reason why is when you've maxed out the number of CPUs you can put in a single machine, you can never max out the number of machines you can buy. If you wrote your program for a multi-process architecture, moving it to multiple machines is almost trivial.

In short, threads are always the wrong answer.

But, but, but, you might say, what about optimizing that single process to more efficiently utilize the CPU and other resources?

In that case, you'll want to go with an asynchronous solution using an event loop. Such code is much easier to write and maintain and debug than threaded code. The difference is with real threads, you never know when you might lose control. With asynchronous "threads" the places where one thread loses control is explicit.

So while Java does do a good job of threads, threads are never the answer you are looking for.

Also, locking and unlocking every single variable is really, really slow. It's one of the reason why CPython can beat Java in certain cases.

DNS Client Implementation

The DNS client that comes with Java is horribly broken. The default is to cache DNS responses forever. You can change this configuration to cache responses for a specific amount of time for *all domains*. VeriSign recommends its users who happen to use Java disable caching altogether because it is horrible broken. See http://www.verisigninc.com/assets/stellent/030957.pdf

In a normal programming language, you can fix this. In Java, you can't.

Try Running More than One JVM on a Box

I heard a report from someone that they have a big problem with running more than one JVM on a box at a time. Java appears to be so resource hungry that it doesn't get along with itself.

I recall that Java people would always prefer each process has its own box, though I never really questioned why at the time. My latest project is Python-based, and we started off running ~10 processes for our system on a single box before breaking things out to their own box when they became too busy. Python seems to do a really good job at sharing memory and other things, so I don't have to think really hard before throwing stuff on a box to try it out.

It doesn't appear this works well with Java however.

I'll try to collect more data and anecdotes on this, but I may be wrong and I am open to that too.

Sun Microsystems May Sue You

I should point out something that I thought was obvious but I guess needs to be repeated.

When you use proprietary software, things can go wrong in a really bad way. According to Murphy's Law, if something bad can happen, it will happen, and so you are almost guaranteed that eventually, one of the bad things associated with proprietary software will happen to you. Combine this with the economic incentive to hurt you, you really shouldn't use proprietary software for anything.

Google chose to use Java for its Android platform. It had to make some substantial changes to Java so much so that they simply rewrote the core part of it, the libraries. Google's version was designed to work with the platform they were building. For whatever reason, Sun's version was simply not feasible, and since it was proprietary, Google couldn't use their code and adapt it to their needs, and then merge their changes upstream. Sun sued Google because they basically copied the API, meaning, Google's version of the code library was compatible with Sun's version. It doesn't matter what the result was --- years were spent, fortunes were spent, and careers came and gone as the lawsuit trudged on.

I don't say that Sun Microsystems are evil. As a hardcore conservative capitalist, I know for a fact that not only are all corporations are bad, all people are bad, myself included. That's why I don't put my trust in them anymore than I put my trust in the government. I know full well that if there is something they can do to increase their profits, they will do it, and indeed, *must* do it. The solution isn't to ban corporations or create marketing campaigns about how bad corporations are. The solution is to stop using proprietary software. It's really that simple.

Think of it this way: There is no corporation that owns the English language. You are free to use it and adapt it whatever way you like, as much as anyone else, and no one can sue you for changing it or re-implementing key parts of it. Even though all the corporations in the world would like to own English, or whatever language everyone is speaking, they can't. So don't give corporations that kind of power by using the languages and software they created and maintain ownership of.

Java 8!!!

(2016-12-20)

A lot of people are pointing out how wonderful Java 8 is and how it addresses all of the concerns I have.

I'll admit that I am not familiar with Java 8. What little I have seen makes me think it's more of the same, but this time with all the right buzzwords.

Java 8 may be fantastic, but seriously, if Java 1-7 was so wonderful, why Java 8?

It reminds me of a conversation I once had with a communist.

Me: "Communists are traitors to Americal. They want to overthrow the government and enforce their twisted ideas by force at the point of a gun."

Him: "No we don't!"

Me: "Then explain to me why communism has done those things in other countries, and why it has the highest body count? Holodomor? The Great Leap Forward? Pol Pot? Hitler doesn't even compare to what you guys have actually done."

Him: "We're new communists! We don't do those things!"

Me: "Then why call yourself communists?"

Java 8 is written by the same company and the same people that gave us Java 1-7. The fact that those languages were horrible messes makes me think that Java 8 can't be much better.

When someone pays me to write Java 8, I'll list out all the problems with it. Until then, realize that "Java 8" is about as convincing an argument as "new communism".

And also, by saying "Java 8" fixes that, you admit that it was broken in the first place. That makes me right.

Why Use Java at All?

If you intend to use Java, I encourage you to write down all the reasons you'd like to use it. Then check the list of things I made above and reconsider. Once you've done that, I would ask that you consider the following languages seriously:

- Python
- Lisp, Scheme, Haskell or some other functional language.
- Vanilla C

Once you've examined those languages, if you still feel like you should use Java, I would appreciate to hear why. Please email me at jgardner@jonathangardner.net and let me know why it is you feel like Java will make your job easier.

All That Said

I still write Java. I write Java because other people do. I wish I could convince more people to stop using Java and start using Python because I get more done with Python than I do in Java, and I hate being unproductive. Whenever I code in Java, I feel like I am digging a ditch with a spoon, while there is a backhoe right beside me that is already gassed up and ready to go.

Update 2016: At my new job, I don't write Java.

Emails

I've been getting about 1-2 emails a week about this article for the past month. I don't know what's inspired so many people to write, but I imagine I've hit a raw nerve.

I'll basically categorize the responses I get.

You're an idiot

Argumentum ad hominem is a logical fallacy, and those who employ it have already lost the argument.

Translation: Name-calling is a sign that you lost.

You Don't Know Java

The experiences I've had above were not experienced in a vacuum. I was working at a huge corporation with many, many Java experts. I've consulted with them in trying to resolve my issues. Every single one I approached with the attitude, "You can't be serious. I must've understood something wrong."

After having writing 10's of thousands of lines of production-quality Java, I think I qualify as an expert, regardless.

Java is Popular

Argumentum ad populum is another logical fallacy, and shows you don't have any arguments left.

Translation: "Everybody is doing it" does not make it right.

Besides, Java is not popular. It is competing in a tight market of ideas and losing out to old and new languages.

Rock On!

You too. We're in this boat together, and those of us with senior-level experience usually end up cleaning the mess that junior-level developers leave behind. So let's try to get the word out.

You Missed Technical Point X

I've updated the content above to reflect and correct my mistakes.

Something Something Enterprise Something

What does enterprise mean to you? And why do programming languages care whether your code is going to be "enterprise" or not? What does "enterprise" even mean? Code is code is code is code. Either you write code well, or you don't. Whether you are doing something "enterprise" has little bearing on that. Granted, in some languages (like Java) it is incredibly difficult to do the right thing the first time. In other languages (Python) it's hard not to do the right thing the first time. I guess if you count "enterprise" as "requiring hundreds of programmers just to get something that actually works" then Java fits the bill perfectly.

See Also

- Execution in the Kingdom of Nouns (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html) (MUST READ!)
- Law of Demeter
- Hacker News (https://news.ycombinator.com/item?id=13214391): Too bad I can't use the site and comment more fully. Email me if you want more answers, or post your own rebuttals on the internet.