JOEL ON SOFTWARE

I'm Joel Spolsky, a software developer in New York City. More about me.

DECEMBER 29, 2005 *by* JOEL SPOLSKY

The Perils of JavaSchools

TOP 10, NEW DEVELOPER, ARTICLES

Lazy kids.

Whatever happened to hard work?

A sure sign of my descent into senility is bitchin' and moanin' about "kids these days," and how they won't or can't do anything hard any more.

When *I* was a kid, I learned to program on punched cards. If you made a mistake, you didn't have any of these modern features like a backspace key to correct it. You threw away the card and started over.

When I started interviewing programmers in 1991, I would generally let them use any language they wanted to solve the coding problems I gave them. 99% of the time, they chose C.

Nowadays, they tend to choose Java.

> "You were lucky. We lived for three months in a brown paper bag in a septic tank. We had to get up at six in the morning, clean the bag, eat a crust of stale bread, go to work down the mill, fourteen hours a day, week-in week-out, and when we got home

Now, don't get me wrong: there's nothing wrong

Now, don't get me wrong: there's nothing wrong with Java as an implementation language.

> our Dad would thrash us to sleep with his belt." — Monty Python's Flying Circus, *Four Yorkshiremen*

Wait a minute, I want to modify that statement. I'm not claiming, *in this particular article,* that there's anything wrong with Java as an implementation language. There are lots of things wrong with it but those will have to wait for a different article.

Instead what I'd like to claim is that Java is not, generally, a hard enough programming language that it can be used to discriminate between great programmers and mediocre programmers. It may be a fine language to work in, but that's not today's topic. I would even go so far as to say that the fact that Java is not hard enough is a feature, not a bug, but it does have this one problem.

If I may be so brash, it has been my humble experience that there are two things traditionally taught in universities as a part of a computer science curriculum which many people just never really fully comprehend: pointers and recursion.

You used to start out in college with a course in data structures, with linked lists and hash tables and whatnot, with extensive use of pointers. Those courses were often used as weedout courses: they were so hard that anyone that couldn't handle the mental challenge of a CS degree would give up, which was a good thing, because if you thought pointers are hard, wait until you try to prove things about fixed point theory.

All the kids who did great in high school writing pong games in BASIC for their Apple II would get to college, take CompSci 101, a data structures course, and when they hit the pointers business their brains would just totally explode, and the next thing you knew, they were majoring in Political Science because law school seemed like a better idea. I've seen all kinds of figures for drop-out rates in CS and they're usually between 40% and 70%. The universities tend to see this as a waste; I think it's just a necessary culling of the people who aren't going to be happy or successful in programming careers.

The other hard course for many young CS students was the course where you learned functional programming, including recursive programming. MIT set the bar very high for these courses, creating a required course (6.001) and a

textbook (Abelson & Sussman's Structure and Interpretation of Computer Programs) which were used at dozens or even hundreds of top CS schools as the de facto introduction to computer science. (You can, and should, watch an older version of the lectures online.)

The difficulty of these courses is astonishing. In the first lecture you've learned pretty much all of Scheme, and you're already being introduced to a fixed-point function that takes another function as its input. When I struggled through such a course, CSE121 at Penn, I watched as many if not most of the students just didn't make it. The material was too hard. I wrote a long sob email to the professor saying It Just Wasn't Fair. Somebody at Penn must have listened to me (or one of the other complainers), because that course is now taught in Java.

I wish they hadn't listened.

Therein lies the debate. Years of whinging by lazy CS undergrads like me, combined with complaints from industry about how few CS majors are graduating from American universities, have taken a toll, and in the last decade a large number of otherwise perfectly good schools have gone 100% Java. It's hip, the recruiters who use "grep" to evaluate resumes seem to like it, and, best of all, there's nothing hard enough about Java to really weed out the programmers without the part of the brain that does pointers or recursion, so the drop-out rates are lower, and the computer science departments have more students, and bigger budgets, and all is well.

The lucky kids of JavaSchools are never going to get weird segfaults trying to implement pointer-based hash tables. They're never going to go stark, raving mad trying to pack things into bits. They'll never have to get their head around how, in a purely functional program, the value of a variable never changes, and yet, it changes all the time! A paradox!

They don't need that part of the brain to get a 4.0 in major.

Am I just one of those old-fashioned curmudgeons, like the Four Yorkshiremen, bragging about how tough I was to survive all that hard stuff?

Heck, in 1900, Latin and Greek were required subjects in college, not because they served any purpose, but because they were sort of considered an obvious

they served any purpose, but because they were sort of considered an obvious requirement for educated people. In some sense my argument is no different that the argument made by the pro-Latin people (all four of them). "[Latin] trains your mind. Trains your memory. Unraveling a Latin sentence is an excellent exercise in thought, a real intellectual puzzle, and a good introduction to logical thinking," writes Scott Barker. But I can't find a single university that requires Latin any more. Are pointers and recursion the Latin and Greek of Computer Science?

Now, I freely admit that programming with pointers is not needed in 90% of the code written today, and in fact, it's downright dangerous in production code. OK. That's fine. And functional programming is just not used much in practice. Agreed.

But it's still important for some of the most exciting programming jobs. Without pointers, for example, you'd never be able to work on the Linux kernel. You can't understand a line of code in Linux, or, indeed, any operating system, without really understanding pointers.

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable. The very fact that Google invented MapReduce, and Microsoft didn't, says something about why Microsoft is still playing catch up trying to get basic search features to work, while Google has moved on to the next problem: building Skynet^H^H^H^H^H^H the world's largest massively parallel supercomputer. I don't think Microsoft completely understands just how far behind they are on that wave.

But beyond the prima-facie importance of pointers and recursion, their real value is that building big systems requires the kind of mental flexibility you get from learning about them, and the mental aptitude you need to avoid being weeded out of the courses in which they are taught. Pointers and recursion require a certain ability to reason, to think in abstractions, and, most importantly, to view a problem at several levels of abstraction simultaneously. And thus, the ability to understand pointers and recursion is directly correlated

with the ability to be a great programmer.

Nothing about an all-Java CS degree really weeds out the students who lack the mental agility to deal with these concepts. As an employer, I've seen that the 100% Java schools have started churning out quite a few CS graduates who are simply not smart enough to work as programmers on anything more sophisticated than Yet Another Java Accounting Application, although they did manage to squeak through the newly-dumbed-down coursework. These students would never survive 6.001 at MIT, or CS 323 at Yale, and frankly, that is one reason why, as an employer, a CS degree from MIT or Yale carries more weight than a CS degree from Duke, which recently went All-Java, or U. Penn, which replaced Scheme and ML with Java in trying to teach the class that nearly killed me and my friends, CSE121. Not that I don't want to hire smart kids from Duke and Penn — I do — it's just a lot harder for me to figure out who they are. I used to be able to tell the smart kids because they could rip through a recursive algorithm in seconds, or implement linked-list manipulation functions using pointers as fast as they could write on the whiteboard. But with a JavaSchool Grad, I can't tell if they're struggling with these problems because they are undereducated or if they're struggling with these problems because they don't actually have that special part of the brain that they're going to need to do great programming work. Paul Graham calls them Blub Programmers.

It's bad enough that JavaSchools fail to weed out the kids who are never going to be great programmers, which the schools could justifiably say is not their problem. Industry, or, at least, the recruiters-who-use-grep, are surely clamoring for Java to be taught.

But JavaSchools also fail to train the brains of kids to be adept, agile, and flexible enough to do good software design (and I don't mean OO "design", where you spend countless hours rewriting your code to rejiggle your object hierarchy, or you fret about faux "problems" like has-a vs. is-a). You need training to think of things at multiple levels of abstraction simultaneously, and that kind of thinking is exactly what you need to design great software architecture.

You may be wondering if teaching object oriented programming (OOP) is a good weed-out substitute for pointers and recursion. The quick answer: no. Without debating OOP on the merits, it is just not hard enough to weed out mediocre

debating OOP on the merits, it is just not hard enough to weed out mediocre programmers. OOP in school consists mostly of memorizing a bunch of vocabulary terms like “encapsulation” and “inheritance” and taking multiple-choice quizzicles on the difference between polymorphism and overloading. Not much harder than memorizing famous dates and names in a history class, OOP poses inadequate mental challenges to scare away first-year students. When you struggle with an OOP problem, *your program still works*, it’s just sort of hard to maintain. Allegedly. But when you struggle with pointers, your program produces the line **Segmentation Fault** and you have no idea what’s going on, until you stop and take a deep breath and really try to force your mind to work at two different levels of abstraction simultaneously.

The recruiters-who-use-grep, by the way, are ridiculed here, and for good reason. I have never met anyone who can do Scheme, Haskell, and C pointers who can’t pick up Java in two days, and create better Java code than people with five years of experience in Java, but try explaining that to the average HR drone.

But what about the CS mission of CS departments? They’re not vocational schools! It shouldn’t be their job to train people to work in industry. That’s for community colleges and government retraining programs for displaced workers, they will tell you. They’re supposed to be giving students the fundamental tools to live their lives, not preparing them for their first weeks on the job. Right?

Still. CS is proofs (recursion), algorithms (recursion), languages (lambda calculus), operating systems (pointers), compilers (lambda calculus) — and so the bottom line is that a JavaSchool that won’t teach C and won’t teach Scheme is not really teaching computer science, either. As useless as the concept of function currying may be to the real world, it’s obviously a prereq for CS grad school. I can’t understand why the professors on the curriculum committees at CS schools have allowed their programs to be dumbed down to the point where not only can’t they produce *working programmers*, they can’t even produce CS grad students who might get PhDs and compete for their jobs. Oh wait. Never mind. Maybe I do understand.

Actually if you go back and research the discussion that took place in academia during the Great Java Shift, you'll notice that the biggest concern was whether Java was *simple* enough to use as a teaching language.

My God, I thought, *they're trying to dumb down the curriculum even further!* Why not spoon feed everything to the students? Let's have the TAs take their tests for them, too, then nobody will switch to American Studies. How is anyone supposed to learn anything if the curriculum has been carefully designed to make everything easier than it already is? There seems to be a task force underway (PDF) to figure out a simple subset of Java that can be taught to students, producing simplified documentation that carefully hides all that EJB/J2EE crap from their tender minds, so they don't have to worry their little heads with any classes that you don't need to do the ever-easier CS problem sets.

The most sympathetic interpretation of why CS departments are so enthusiastic to dumb down their classes is that it leaves them more time to teach actual CS concepts, if they don't need to spend two whole lectures unconfusing students about the difference between, say, a Java **int** and an **Integer**. Well, if that's the case, 6.001 has the perfect answer for you: Scheme, a teaching language so simple that the entire language can be taught to bright students in about ten minutes; then you can spend the rest of the semester on fixed points.

Feh.

I'm going back to ones and zeros.

(You had ones? Lucky bastard! All we got were zeros.)

successful software companies, and rubber duckies.

If you want to know when I publish something new, I recommend getting an RSS reader like NewsBlur and subscribing to my RSS feed.

ABOUT THE AUTHOR.

In 2000 I co-founded Fog Creek Software, where we created lots of cool things like the FogBugz bug tracker, Trello, and Glitch. I also worked with Jeff Atwood to create Stack Overflow and served as CEO of Stack Overflow from 2010-2019. Today I serve as the chairman of the board for Stack Overflow, Glitch, and HASH.

← PREVIOUS POST
The Perils of JavaSchools

NEXT POST →
Test Yourself