Why Java? Moving Beyond Procedural Programming

"Not knowing OO in 2004 is quite simply career suicide" - Hal Helms

October 4, 2004

Summary

.NET languages such as C# and VB .NET are OO, and CFML as of MX is kind of OO, so why do CF programmers need to Java? First, argues Matt Woodward, Java is one of the purest implementations of OO that exists, and to truly make this shift from procedural to OO programming, a deep understanding of OO concepts is crucial. Java is the best way to get this deep understanding. Second, as Hal Helms has stated: "In 2004, not being deeply involved in OO is career suicide."

Hal Helms' CFUN-04 presentation reinforced my thoughts on the topic of OOP.

Relatively early on in his session he made the statement that "Not knowing OO in 2004 is quite simply career suicide," and as many of you already know, I completely agree. If for no other reason you need to become very familiar with OO and start using it because the rest of the world expects this sort of development now.

That is far, far from the only reason to start getting into OO development, however. So why should we use OO? Trying to solve real-world problems with software oftentimes becomes quite complex, and OO is the best way in software development to manage complexity. I know many of you don't agree with this statement, but it's been around for a very long time at this point, and the giants in software development all agree that OO is the best way to manage complexity in our software systems. The more functionality you need in your applications, the harder it gets to manage this complexity, and objects manage this far better than any other software programming methodology.

I'll probably reiterate this point again later, but the bottom line in my mind is that it's far easier to develop small, cohesive objects that do one thing and do it well, and develop communication between these objects, than it is to build top-down procedural code. I fully admit that I have a pretty strong Java background so this notion may be so ingrained in me that it's hard for me not to think this way, but I still think that most of us haven't dived into OO because we think it's too complicated. In actuality it will *simplify* your development once you get the hang of it. You'll build things more easily, your applications will be far more maintainable and flexible, and quite honestly I believe you'll have more fun building your applications. That isn't to say there isn't a learning curve to get past if you're used to procedural programming, but once you grasp the concepts I guarantee you'll have an "A-ha!" moment and you'll never want to do things any other way.

I had a really interesting conversation along these lines with a Fusebox developer when he found out I was pretty heavily into Mach-II. He was using Fusebox 4 and made the statement that since Fusebox 4 also dealt with announcing and processing messages, Mach-II and Fusebox 4 were essentially the same thing. I had to bite my tongue not to say, "Well, you obviously don't understand Mach-II very well then."

Of course when you boil two things down to their most basic elements they will have a lot of similarities. In music theory there is a method of analysis called "Schenkerian," and the basic idea is that you draw a diagram that charts the overall process of a piece even as large as a symphony. The fascinating thing to me about this is that when you diagram a symphony by Mozart and one by Beethoven in this way, you could very well end up with diagrams that are virtually identical. At that point I say, "So what?? What does boiling things down to this level really get you" It's the details that we listen to in music, and it's the details we work with as developers. Sure the end user doesn't care what language or framework we're using, but we as developers have to work at that level.

Under the hood, of course, Fusebox 4 and Mach-II are quite radically different. When I pointed this out to the Fusebox advocate, he said something along the lines of, "Yes, but I bet you have an OO background. For beginning developers I think that's pretty hard to grasp, so I think the frameworks really appeal to two different types of developers." I completely agree. The thing I *don't* agree with is the underlying sentiment of this statement: "Mach-II and OO is too hard, so I'll just keep doing procedural coding and use Fusebox." It's TOO HARD? This stuff isn't rocket science, and it just annoys me to no end when people look at something, see a challenge, and walk away. If you don't know this stuff, learn it! Developers should WANT to learn new things and NEED to be learning new things all the time. If it's too hard at this point, start learning it gradually and using it, and before you know you'll have the hang of it, but for crying out loud don't just ignore it because it presents a challenge.

OK, preaching over, and on to what Hal had to say about Java specifically. Why would we as CFers want to learn Java? We have CFCs, aren't those objects? Well yes, CFCs are objects (and I think I probably believe they're more pure objects than some others do), but there are absolutely some fundamental concepts involved with true OOP that don't exist in CFCs. That's not to take away from their value; I use CFCs everywhere I possibly can and will

continue to do so. When we're talking about learning OOP, however, Java probably presents a more pure way of learning the concepts than CF does.

Among major languages, Java is by far the most pure implementation of OO that exists. (As an aside, Bruce Eckel, author of *Thinking in Java*, believes that Python may even be more pure.) Java provides excellent tools for writing complex applications simply, but this does *not* mean that Java itself is complex, which Hal feels is a common misconception. We all want to write robust, maintainable applications, but not all systems (and CF is guilty of this) *enforce* this kind of development. You can get away with things in CF applications that Java wouldn't even allow you to compile. What this does essentially is traps potential bugs or functional side effects at every step of the way and keeps your applications more robust and safe.

As with anything, however, this is a trade-off. Java is an extremely vast language, and does take a significant amount of time to learn well. That isn't to say that it's hard to get started, but when we're talking about saying "I'm a Java developer," that's a significant time investment to get to that level. Ben Edwards who works with Hal on Mach-II also teaches a lot of Java classes with Hal, and Hal had a great quote from Ben. At one point they were talking with some programmers about Java and one of them said, "You know what? I REALLY know Java." Ben?s response was "That's about like saying 'I really know science.'"

Luckily, however, it's quite easy to get started with Java, and Hal had some great examples that were designed to take the fear out of it. When I first started learning Java I didn't have a choice - my employer just sent me to Sun Microsystems classes and doing Java was required of my job. Coming from a CF 4 background prior to these courses, it seemed a bit daunting to me at first as well. The first course I took was "Migrating to OO Programming with Java Technology." We didn't write a single line of code in this class, we just discussed the concepts and did a lot of diagrams of objects and simple applications. I definitely had my "A-ha!" moment in that course and once that happened, I was sold and it was difficult for me to go back to thinking about programming in any other way. Going back to CF 4 and later 4.5 and 5 was pretty painful. Before CFMX came out I had been doing Java programming for a few years, so when I got moved to another position in my company and was going to be doing CF programming again, I was really happy to see that CF now had some sort of OO capabilities, and they're only getting better.

Back to Java. If I'm a CF developer, why would I want to learn Java? Other than the fact that it's probably the best way to become extremely firmly grounded in OO concepts, since CFMX now runs on top of J2EE, we can start doing some amazing things by integrating the two technologies, and the two complement each other amazingly well. CFML is a fantastic presentation and middle-tier technology, but when things start getting really complex on the backend, CFML starts to break down at some level. Java isn't very good at the front-end, but is extremely powerful on the backend and can offer capabilities and robustness at that level that CF can't match.

Also, Java *enforces* best practices, whereas ColdFusion really doesn't. That's not necessarily a slam on CF since that isn't the way it was designed (and the bifurcation some see in the CF community is another topic in and of itself), but Java by its very design is a good teacher. The Java compiler is basically like the piano teacher who slaps your hands with a ruler when you do something wrong; if, for example, you declare x to be an integer and later try to write x = ?some string?, Java won't even compile this and will give you a (usually) very specific error so you know exactly what the problem is. This may seem annoying but it's absolutely essential for building robust software that's free of weird bugs that are horrendously difficult to track down. Believe me, you'll learn to love the assistance and "saving us from ourselves" help that the language offers.

So specifically what does OO do to manage complexity? Hal broke this down into three areas, in order of importance:

1. encapsulation
2. polymorphism
3. inheritance

These three hallmarks of OOP help maintainability and can greatly reduce the complexity of building large or complex systems. I completely agree with Hal putting encapsulation at the top of the list, and this is something we most definitely *can* leverage in CFCs. Basically when you put things in an object and build your objects in a particular way, you isolate the impact that changing one piece of the system has on other parts of the system, and you also help to protect your data. Keeping objects small and highly cohesive greatly simplifies the creation of complex systems.

Polymorphism is also important, largely for extensibility and maintainability. The example Hal used was different types of customers who pay their bills in different ways. From a maintainability standpoint, if you had to add a new method for each payment type every time you added a new type of customer, things get ugly very quickly. If,

however, you have a payBill() method for all types of customers (credit, cash, etc.) and each implementation defines *how* the bill gets paid then the system doesn't care - it just calls payBill() and the rest happens as it should. The example that I remember from one of my Java classes is having a draw() method for shapes. A circle, triangle, and square all are drawn in very different ways, but if each object implements its own draw() method, the system can just call draw() and not worry about the details. Powerful stuff.

Inheritance is probably the thing that's most overused by people when they first get started with OO. Many folks see inheritance everywhere and use it all over the place, when in actuality it's really only used in specific circumstances. One good example that comes to mind is having a base Person class and having an Employee and Customer class that inherits from Person. That way your Employee and Customer classes inherit all the "stuff" from the Person class like name, address, phone number, etc. but can also have their own properties and methods. You certainly wouldn't want to pay a Customer a salary! When used wisely inheritance helps to eliminate a great deal of duplication that would otherwise be necessary. CFCs support inheritance pretty well (although some will gripe about the specifics).

As alluded to earlier, Java also is incredibly strict when it comes to data typing (this is also known as "strongly typed"). You can't have something be an integer one second and a string another - Java refuses to allow you to do that. This was done by design in the language, and what this does is ensures that the specifications in our system that we provide for ourselves and others on our team are enforced by the language. If one programmer creates a class that assumes variable x is going to be a number, another programmer shouldn't be allowed to use x as a string. CF lets you do that, Java does not. Rather than seeing that as a limitation, I agree with Hal's sentiment that this is done to help you, not hinder you as a programmer. Because Java ensures that the original specification is enforced, everything is more safe, and the amazing thing about Java is that because of this design there are really *very* few runtime exceptions. By the time something will compile, you're nearly home free.

The most important thing to remember is this: OO IS NOT DIFFICULT. Procedural programmers just need to get out of the procedural mindset. This is, as I've probably said several times by now, a bit of a learning curve if you've never examined OO programming concepts before, but the syntax and writing the code is really not the difficult part. If you understand OO very well conceptually writing the code isn't the thing that will trip you up.

At this point Hal got into a brief demo that seemed to get a lot of folks in the room to have a mini "A-ha!" moment. Java classes are basically cookie cutters used at runtime to stamp out real components. The important thing to remember according to Hal is "we don't really care about the cookie cutter - it's the cookie we're after!" Well-designed OO programs are quite simply a series of communications between objects, and if the objects are designed well, are cohesive, and do only what they're supposed to do and do it well, even the communication bit isn't terribly difficult.

As an aside Hal made the point to mention that performance is no longer an issue with Java. It certainly used to be as even the Java camp will admit, but with modern versions of Java the performance is fantastic - at least as good as C++ in most areas, and in certain cases Java has actually been found to outperform C++. If you still think of Java as a lumbering beast, it's time to re-examine things.

To make a comparison between procedural programming and OO programming, Hal then pointed out that in a procedural program, you change data directly: name = "Matt";

In OO, on the other hand, you make a request to a component to set the name. You never access data directly:

```
mattComponent.setName("Matt");
```

This is again for safety and to reduce complexity. In this particular example you might be saying "But look at all that extra code in the OO example!" but I hope you can imagine that in a more complex system with lots of data, things can get hairy quickly in procedural coding, whereas using objects and messaging between these objects reduces the complexity involved with the logic of the program.

OK, so at the end of the day we're still back to the fundamental question: Why Java? .NET languages such as C# and VB .NET are OO, CFML as of MX is kind of OO, so why Java? At this point Hal reiterated some earlier points to drive them home. First, Java is one of the purest implementations of OO that exists, and to truly make this shift from procedural to OO programming, a deep understanding of OO concepts is crucial. Java is the best way to get this deep understanding. Second, he stated "In 2004, not being deeply involved in OO is career suicide." Regardless of all the arguments to the contrary, I agree with this point. This suicide may not happen today or tomorrow, but you won't get a warning either!

Other than the academic reasons for learning Java, there are many practical reasons as well. As mentioned earlier, Java doesn't have good presentation layer capabilities, and CF doesn't have the same level of backend power that Java does, so crafting a strategy for truly integrating the two seems to be a way to gain the best of both worlds. Using Java for the "model" layer of your applications (and yes, this does get into MVC, but again don't be afraid, it's not that tough to grasp!) and using CF for the presentation layer, especially with all the fantastic presentation layer enhancements in Blackstone, you can give your users a great experience and build more solid, robust applications. Hal then made the argument that if you do build everything with CFCs, at a minimum build your app using strict MVC design patterns so that if/when you need to, you can swap your CFCs out with Java objects and minimize the impact on your system.

Hal then gave some very nice specific examples of some simple animal classes in a petting zoo application that I'll be happy to share with folks. His final points were that with a strongly-typed, pure OO language like Java, the tools and language help you more than with something like CF. The language works with you and protects you because it enforces the specifications you define rather than just suggesting them. He also made the caveat that you don't have to learn ALL of Java, but we should all at least be getting started so we can use bits and pieces of Java as needed.

In many senses Hal was preaching to the choir as far as I personally was concerned in this session, but it was great to see such a large number of people there who are starting to understand the importance of OO and how learning Java and beginning to incorporate it into their CF programming can help them. I left being even more convinced than I was when I went in just how important this really is.

About the author

Matt Woodward is a Web application developer for i2 Technologies in Dallas, and also works as a consultant through his company, Sixth Floor Software. He is a Macromedia Certified ColdFusion developer, a member of Team Macromedia, and has been working with ColdFusion since 1996. In addition to his ColdFusion work, Matt also develops in Java and PHP. (*more*)