Cover Feature

The Feel of Java

Java is a blue collar language. It's not PhD thesis material but a language for a job. Java feels very familiar to many different programmers because we preferred tried-and-tested things.

James Gosling
Sun Microsystems Inc.

Java evolved out of a Sun research project started six years ago to look into distributed control of consumer electronics devices. It was not an academic research project studying programming languages: Doing language research was actively an antigoal. For the first three years, I worked on the language and the runtime, and everybody else in the group worked on a variety of different prototype applications, the things that were really the heart of the project. So the drive for changes came from the people who were actually using it and saying "do this, do this, do this."

Probably the most important thing I learned in talking to the folks building TVs and VCRs was that their priorities were quite different from ours in the computer industry. Whereas five years ago our mantra was *compatibility*, the consumer electronics industry considered secure networking, portability, and cost far more important. And when compatibility did become an issue, they limited notions of compatibility to well-defined interfaces—unlike the computer industry where the most ubiquitous interface around, namely DOS, is full of secret back doors that make life extremely difficult.

I've listed the differing priorities for the commercial software and consumer electronics industries in Table 1. One interesting phenomenon that has occurred over the past five years is that consumer electronics concerns have become mainstream software concerns as the market for software in the home has grown.

The buzzwords that have been applied to Java derive directly from this context. In the consumer electronics world, you connect your VCR to a television, your telephone to a network. And the consumer electronics industry wants to make these kinds of networked appliances even more pervasive.

Architecture neutrality is another issue. In the consumer electronics business, there are dozens of different CPU types and good reasons for all of them in their individual contexts. But developing software for a dozen different platforms just doesn't scale, and it was this desire for architecture neutrality that broke the C++ mold—not so much C++ the language, but the standard way people built C++ compilers.

BLUE COLLAR LANGUAGE

Java is a blue collar language. It's not PhD thesis material but a language for a job. Java feels very familiar to many different programmers because I had a very strong tendency to prefer things that had been used a lot over things that just sounded like a good idea. And so Java ended up as this fusion of four different kinds of programming.

1. It has an object-oriented flavor that derives from a number of languages—Simula, C/C++, Objective C, Cedar/Mesa, Modula, and Smalltalk.
2. Another one of my favorite areas is numeric programming. One of the things that's different about Java is that we say what 2 + 2 means. When C came out there were so many different ways of computing 2 + 2 that you couldn't lay down any kind of rule. But today, the IEEE 754 standard for floating-point arithmetic has won, and the world owes William Kahan and the other folks who worked on it a real debt, because it removes much of the complexity and clutter in numerical programming.
3. Java also has a systems programming flavor inherited from C that has proven useful over the years.
4. But the one way in which Java is unique is its distributed nature—it feels like there aren't boundaries between machines. People can have pieces of behavior squirt back and forth across the network, picked up here, landed over there. And they just don't care. The network, by and

This article is based on remarks made at OOPSLA 96.

0018-9162/97/$10.00 © 1997 IEEE

Table 1. Differing priorities of the commercial software and consumer electronics industries five years ago.

Commercial software	Consumer electronics
Compatibility	Security
Performance	Networking
Portability	Portability
Reliability	Reliability
Networking	Performance
Multithreading	Multithreading
Security	Compatibility

large, starts to behave like a sea of computation on which you can go rafting.

Distributed objects on the Web

In this particular environment, one of the key design requirements was to create quanta of behavior that could be shipped from place to place. Oddly enough, classes provide a nice encapsulation boundary for defining what one of these quantum particles is.

We also wanted to keep these quanta of behavior separate from data, which at the time was a real departure. General Magic was doing a very similar thing, except it was putting the code and the data together. In some cases this has a real advantage, but what if you're shipping around JPEG images? You end up in an untenable situation if every JPEG image has to have its own JPEG decompressor: You load a page with 20 images and end up with 20 JPEG decompressors at 100 Kbytes each.

So we worked hard to make sure that data and implementation were separate, but that the data could have tags that say, "I'm a bag of bytes that's understood by this type." And if the client doesn't understand the component's data type, it would be able to turn around and say, "Gee Mr. Server, do you have the implementation for this particular type?" and reach out across the Internet, grab the implementation, and do some checking to make sure that it won't turn the disk drive into a puddle of ash.

Thin clients

You can think of this as the client learning something. It now understands a new data type that it didn't understand before, and it obtained that knowledge from some remote repository. You can start building systems that are much more lean, that feel as though there's this core that understands the basic business of the application.

A Web browser is a good example. It's a simple loop—a set of interfaces to networking standards, document format standards, image format standards, and so on. And other components can plug into this browser until you have this huge brick of code around which you wrap a big steel band. That's your application, and it does everything. But what's lost in this pile of support code is the essence of a Web browser.

Similarly, the support code itself tends to lose its boundaries because people start getting sloppy. They start saying, "Well gee, there's this global variable over there that HTML was using, but I could use that creatively with my HTTP driver." It always bites you in the end, even though short term it feels good. With Java, we tended to do things that promoted up-front pain and long-term health, one of those funny religious principles.

Architecture neutral

Much of Java was driven by the Internet, and there's a series of deductive steps that follow from that starting point. The Internet has a diverse population, some companies' aspirations to the contrary. If you need to avoid doing different versions for different platforms, then you need some way of distributing software that is architecturally neutral. C, by and large, has been very portable, apart from a few gotchas like what does `int` mean. So we pushed for a uniform feeling and a deterministic semantics, so that you know what 2 + 2 means and what kind of evaluation order you have.

JAVA VIRTUAL MACHINE

At the same time, I made the mistake of going to school too long and actually getting a PhD, so I couldn't avoid doing a little bit of theoretical stuff. And besides, when you have people like Bill Joy (Sun cofounder and VP for Research) and Guy Steele (Sun Microsystems Distinguished Engineer) peering over your shoulder and wagging their fingers at you, things become a lot cleaner than the initial hacks one is tempted to commit in the spirit of expediency. And the theoretical work that went into Java really did add a lot of cohesiveness and cleanliness to it. Most of those things are under the covers in the way the virtual machine works. Things like the verifier, which is this minidataflow program prover that determines whether or not programs follow the game rules. But by and large, this kind of innovation was relatively rare in Java.

We use a very old technique where the compiler generates some bytecoded instructions for this abstract virtual machine that's based largely on work from Smalltalk and Pascal-P machines. I put a lot of effort into making it very easy to interpret and verify bytecode before it was compiled into machine code, using both an interpreter and a machine code generator to make sure that generating machine code was pretty straightforward.

Compile-time checking

The Java compiler does a lot of compile-time check-

ing that people aren't used to, and some have complained about the compiler's attitude, that it essentially has no warnings. For example, "used-before-set" is a fatal compilation error rather than just a warning. These may feel like restrictions, but it's rare that the compiler gives an error message without a very good reason. In all cases, we would try something and see how many bugs came out of the woodwork.

One of the interesting cases was name hiding. It's pretty traditional in languages to allow nested scopes to have names that are the same as names in the outer scope, and that's certainly the way it was in Java early on. But people had nasty experiences where they forgot they named a variable *i* in an outer scope, then declared an *i* in an inner scope thinking they were referring to the outer scope, and it would get caught by the inner scope. So we disallowed name hiding, and it was amazing how many errors were eliminated. Granted, it is sometimes an aggravation, but statistically speaking, people get burned a lot by doing that, and we're trying to avoid people getting burned.

Garbage collection

Another thing that's essential for reliability, oddly enough, is garbage collection. Garbage collection has a long and honorable history, starting out in the Lisp community, but it acquired a bad reputation because it tended to take more time than was necessary. Garbage collection gained the reputation of being used by lazy programmers who didn't want to call `malloc` and `free`. But in actual fact, there are a lot of other ways to justify it. And to my mind, one of the ways that works well when you're talking to some hard-nosed engineer is that it helps make systems more reliable. You don't have memory leaks or dangling pointers, and you cut your software maintenance budget in half by not having to chase them. With Java, you never need to worry about pointers off into hyperspace, pointers to one element beyond the end of your array.

Pointer restrictions

This also relates to restrictions on pointers and pointer arithmetic, which can lead to interface integrity problems. We in the engineering world have become accustomed to taking back doors into an object's private space to solve problems in the short term. In C, there's a standard cliché I've used frequently—`((int *) p) [n]`—where you take some pointer, cast it to a pointer or to an integer, subscript it, and are then able to get anything as anything. The world is your oyster.

But long term, this practice always bites you. It creates a tremendous versioning problem, and systems become incredibly fragile. Having one little private variable can make the whole system fall apart. If you look at what often happens in commercial systems, you'll find they end up not using object-oriented programming because of these back doors. They end up doing it in a way that hides all the stuff, so that it's much more obscure.

Exception handling

The exception model that we picked up pretty much straight out of Modula 3 has been, I think, a real success. Initially, I was somewhat anxious about it, because the whole notion of having a rigorous proof that an exception will get tossed can be something of a burden. But in the end, that is a good burden to have. When you aren't testing for exceptions, the code is going to break at some time in any real environment where surprising things always happen. Ariane 5 provides a vivid lesson on how important exception handling is.

Although exception handling makes Java feel somewhat clumsy because it forces you to think about something you'd rather ignore, your applications are ultimately much more solid and reliable.

OBJECT-ORIENTED EXTENSIBILITY

One of the things about Java that I pushed on pretty hard was allowing for future change. Much of that comes from Java's Lisp-like late binding, where methods are looked up on the fly at the very end. But there's a lot of optimization that gets done to rewrite the instructions so that method calls are fast and the various code generators turn them into the obvious three-instruction sequence.

Thus, you can add methods almost fearlessly and can add and remove private variables with total impunity. You have to make sure of a few things—for example, that you don't remove methods that aren't being used, or if you want to remove or change them, you at least leave another method in there whose type signature is the same. As long as you practice this relatively simple discipline, you can change classes pretty readily without having to worry about how this breaks all of your subclasses and the applications based on them.

What I found most interesting in watching people use Java was that they used it in a way similar to rapid prototyping languages. They just whacked something together. I was initially surprised by that, because Java is a very strongly typed system, and dynamic typing is often considered one of the real requirements of a rapid prototyping environment. But after watching people for a while and doing it myself and thinking "Why does this feel this way to me," I decided that probably the most important thing was that in a typical rapid prototyping language like Smalltalk, you find out about it fast when

What I found most interesting in watching people use Java was that they used it in a way similar to rapid prototyping languages. They just whacked something together.

```
System.getProperty("os.name")
abstract class Spam {…}
class SpamSolaris extends Spam
{…}
class SpamWin32 extends Spam{…}
Class c = Class.forName("Spam"
 +System.getProperty("os.name"));
```

Figure 1. Sample adapter providing appropriate subclasses for different operating systems, thus achieving complete portablility.

something goes wrong. There aren't these mysterious memory smashes. Java does a pretty good job of avoiding situations where mysterious alpha particles come in from hyperspace and blow up your system, where you spend four days to discover that you had a for-loop clearing an array that went one element too far, and where that fact isn't discovered until thousands of instructions later when some other memory block is being accessed.

Dynamic linking

Another important aspect of Java is that it's dynamic. Dynamic linking—where classes come in and have their links snapped very late—lets you adapt to change. Change not only in the versioning problem from one generation of software to the next, but also in the sense of being able to load handlers for new data types. It lets the system defer a lot of decisions—principally object layout—to the runtime.

We had a longstanding debate, particularly with people from the Objective C crowd, about factories versus constructors. A factory is a static method on a type—that is, you would say `type.new` rather than `new Type`. I was not totally persuaded by the factory argument because there was always the problem of who, in the end, creates the object. So Java stayed with the C++ way of saying `new Type`.

But factories are used as a style in places where you don't know exactly what you want or if you need a new object. If you want a font, for example, you don't necessarily want to create it and might prefer to look it up. Java's dynamic behavior is often fed by this style of using static factory methods to allocate objects rather than call the constructor directly.

One way this is used is in this short cliché of doing `new` on a string name, where you start by calling a static method on a class called `forName`, which takes the string as a parameter and gives you a class object that happens to have that name. Where this becomes interesting is when the string parameter to `forName` is something that you compute by concatenating strings together. You use a method on a class object called `makeInstance`, which calls the default constructor:

```
Class c = Class.forName("foo."+x);
Thing b = (Thing) c.makeInstance();
```

These two things together interact with these objects in Java called class loaders, which are responsible for taking a class name and finding an actual class implementation for it. This is the central cliché for achieving this quanta of behavior where we take the MIME type, for instance, do a little bit of string mushing to turn the MIME type into a class name, and say `Class.forName`, which causes all sorts of HTTP searches to happen. And then magically you've got a handler for that type that's been installed dynamically.

Another stylized way we use this is for adapters. Adapters are interfaces that are designed for achieving portability. Say you have an interface to some networking abstraction or file system, and you want to provide both a consistent superclass with a consistent interface and an appropriate subclass that is dependent upon the actual operating system you're using. Adapter objects are often looked up by using a factory rather than a normal constructor. We then use the previous cliché fed by a property inquiry like system.getProperty of os.name, as shown in Figure 1, which returns a string like Solaris or Win32. If you're trying to find an implementation of this abstract class Spam and you have a SpamSolaris, a SpamWin32, and a SpamMac class, you can go through this sequence to obtain the appropriate "spamming" class for your machine. And it ends up being completely portable.

PERFORMANCE

We were working on a prototype just-in-time compiler from the very beginning, as I felt strongly that Java had to feel fast. I wanted something comparable to C in performance. This feeling came from watching what people did with previous scripting languages I had written, where they always pushed them way beyond what I expected.

Much of Java's semantics and that of the virtual machine were driven by a couple of canonical examples. Namely, I wanted to get `a = b + c` to be one instruction, and `p.m` of something to be about three instructions. It currently tends to be four or five, which is still a pretty small number. That goal pretty much dictated that the system be statically typed. Many languages infer a fair amount of this type information or they'll compile it, assume a type, and then do checks, but that adds a huge amount of complexity. Strong typing simplifies the translation process tremendously and picks up a lot of programming errors as well.

Java 1.0 was tuned far more for portability than for performance, but I think it still performed reasonably well. Today, there are a number of JIT compilers on the market that are not quite up there with C but are

getting awfully close. It's pretty interesting to have a language that has a scripting feel without the usual scripting language performance. Furthermore, rewriting the interpreter in assembly would give us at least a factor of three speedup, and actual machine code generators would give us a factor of 10 or 20. There is now enough demand for Java to justify all that platform-dependent work, if only because it makes other people's jobs easier.

Another important thing about Java is its rich class library. Java itself, as a language, is pretty simple, as are most languages. The real action is in the libraries, and we tried hard to have a fairly large class library straight out of the box. That was pretty easy for us because we wrote buckets of code for these prototype consumer electronics applications. This gives the environment a very rich feeling, although it's clear we have a long way to go. This is probably where we're working the hardest right now.

So, how does Java feel? Java feels playful and flexible. You can build things with it that are themselves flexible. Java feels deterministic. You feel like it's going to do what you ask it to do. It feels fairly nonthreatening in that you can just try something and you'll quickly get an error message if it's crazy. It feels pretty rich. We tried hard to have a fairly large class library straight out of the box. By and large, it feels like you can just sit down and write code. ❖

James Gosling is vice president, a Sun fellow, and a Distinguished Engineer at Sun Microsystems. Recently, he served as lead engineer for the Java/HotJava system. He has built satellite data acquisition systems, a multiprocessor version of Unix, and several compilers, mail systems, and window managers. He has also built a WYSIWYG text editor, a constraint-based drawing editor, and a text editor called emacs for Unix systems. He received a BSc in computer science from the University of Calgary, Canada, and a PhD in computer science from Carnegie Mellon University.

Readers can contact Gosling at Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Calif. 94043.