2000

B. A Java Dialectic

Douglas Lyon

Frances Grodzinsky
Sacred Heart University

B. A Java Dialectic

barf [ba:rf] 2. "He suggested using FORTRAN, and everybody barfed."
- The Shogakukan
DICTIONARY OF NEW ENGLISH

Speak the truth,
then leave quickly.
-Serbian Proverb

By Douglas Lyon and Frances S. Grodzinsky, Ph.D[1]

ABSTRACT

Many books often gush over Java. People want to know, before they learn a language, what the drawbacks are for a language. They want to know what the strenghs are. After all, a computer language is just a tool. We shape our tools, and there after, our tools shape us!

This chapter contains a dialog between three Java instructors. The participants are, Professors Good, Bad and Ugly. Professor Good is a strong proponent of Java. Professor Bad is a strong opponent of Java. Professor Ugly has been leaning toward the asthetic analysis of the language, particularly with an eye toward teaching.

---

[1]Computer Science Dept., Sacred Heart University, Fairfield, CT 06432, (203)371-7799
grodzinskyf@sacredheart.edu

Our Gang of Three evaluate aspects of the language, the Java environment and Java as a teaching tool.

B.1. Introduction

Arguments about preferences in programming languages, especially as a teaching tool, often become akin to religious ones. Although we may be motivated to quantify the criterion of optimality for measuring the 'goodness' of a language, this is an on-going debate where even the criteria become hotly contested. Instead we intend to focus on how choices made by the Java language designers impact, either directly or indirectly, the use of the language as a teaching tool.

Many computer science departments have opted to teach an object-oriented language in CS1. Java is becoming more and more the language of choice. How does this choice affect the students as we extend the use of Java beyond CS1 and 2?

What are the subtleties of the language of which we should be aware, when we take the Java paradigm into our more advanced courses like Image Processing, Graphics and Real-Time Applications?

This paper poses a debate about Java among three professors: Professor Good (PG), Professor Bad (PB) and Doc Ugly (DU) who explore the nuances of the Java language and the Java environment.

B.2. The Java Language

In discussing Java, we have to separate issues of the Java language from issues of the Java environment.

B.2.1. Java is Object-Oriented

PG: Java is strongly-typed and object-oriented. It offers an object-oriented model with multi-threading and platform independence. It has a ‘simpler’ object model than C++ and its standard API library helps students get results quickly [Tyma].

PB: But Java has primitive data types and these are not really object-oriented as compared to the atoms of Smalltalk or ZetaLisp.

DU: Even worse, Java has static methods. These look a lot like functions that reside in a class name space.

PG: Well, OK, but every instance can be treated the same, to a point. For example, I can always use:

```
public void printName (Class c){
  System.out.println(c.getName())
};
```

to print the name of the instance of any class. This is a kind of polymorphism.

DU: Right, but the printName method will never work within a static method. That’s because you can’t get the name of a class before it is instanced!

PG: Well then, just make an instance of the class.

PB: You can’t always do that without knowing the class name. For example, suppose you had a frame that needed to be named for the class name, you might write:

```
public class OpenFrame extends Frame{
      public static void main(String args[]) {
        OpenFrame f = new OpenFrame (“OpenFrame”);
        f.show();
```

```
        }
}
```

Notice how OpenFrame requires a string for the title of the frame and how this string must be embedded in the program. Since the instance of OpenFrame cannot exist before the string is given, the string must be embedded in the program. That is bad!

PG: OK, I acknowledge that Java is a hybrid language. It has procedural, functional and object-oriented elements. It allows the programmer to choose which style to use and when.

PB: That is clearly a problem. Beginners in Java all but ignore the object-oriented features of the language. Often, they are not used to thinking about objects and try to fit Java into a procedural mode in which they feel more comfortable.

DU: I have students that write large programs in a single main method!

PB: And if you are teaching Java after students have been using a functional language, watch out! Often students think that:

```
public class hi {
      public static void main (String args []) {
          hi h = new hi()
      }
}
```

this an example of recursion. They ask “how can this compile?” DU: To make it worse:

```
public static void main (String args[] )
```

assumes a command line interface that some computers do not have (e.g., MacOS). In fact, it is command-oriented and this is inappropriate for applications on a Mac.

B.2.2. Java has Simula-like classes

PG: Java does have classes. These allow the creation of reference types that work just like the built-in classes in the Java API.

PB: But this does not apply at all to the built-in primitive types. Java does not allow user-defined types to be treated the same as the built-in types. Java will not allow you, for example, to create an unsigned byte. This is really bad news. It means that if you want three arrays of red, green and blue, they can't be bytes that range from 0-255, or combinations like:

```
r[i] = b[i] + g[i];
```

They will have to be rewritten to avoid magnitudes that are above 127:

```
r[i] = Math.abs(b[i])+Math.abs(g[i]);
```

This basically makes Java unsuitable for digital signal processing.

PG: All you have to do is define the byte arrays to be arrays of *short*, this will give you the dynamic range you need, without having to resort to arrays of byte.

PB: But this is going to double the storage requirements. This is making you pay for a feature that nobody ever wanted. After all, what is so wrong with allowing users to define their own primitive data types?

PU: It makes the compiler harder to write!

B.2.3. Overloaded operators

PG: It is very good that Java has no overloaded operators because bugs due to overloading are hard to track down. You see, it is very hard to know exactly which overloaded operator method is being invoked.

PB: Overloaded operator methods are useful. Even the designers of Java couldn't escape them. What about the "+" sign? After all:

```
int x=2;
int y=3;
String z = "4";
System.out.println(x+y+z);
// outputs 54
System.out.println(x+z+y);
//outputs 243.
```

DU: Java has to have the overloaded "+" because the alternative is too ugly to consider! How would you handle the un-overloaded "+" when concatenating strings?

```
String s = Integer.toString(x+y);
s= s.cat(z);
System.out.println(s);
```

What a mess! So, we have to have overloaded string operators.

PB: So is Sun saying that they can be trusted to do overloading, but the computer science community can't be trusted? Just think, if Java really had overloaded operators we could write:

```
Complex c = a*b
```

rather than

```
Complex c =a.multiply(b);
```

PG: Look, the basic design objective for Java was to make the language more reliable. By excluding operator overloading you take away a tool that has been

abused in the past. We have to get people to do what is good for them. This means taking away choice!

PB: So now it looks as if Java was designed by idealists. People have been using operator overloading since Algol 68 [Ralston and Reilly]. Operator overloading allows a program's notation to approach that of the mathematics. Mathematics can be an elegant and simple statement for a problem. No overloaded operators means no conventional notation. So why did they leave out this feature?

DU: Many of the features of Java may have been left out to make the compiler easier to write!

PB: Getting the compiler to market quickly is a bad criterion of optimality for the design of a language.

B.3. Multiple inheritance

PG: Java designers made a wise choice in opting for no multiple inheritance. When there is a method or class variable name conflict, multiple inheritance requires topological sorting to determine the order in which things are overridden. This is exceedingly hard to debug.

PB: But inheritance is supposed to be an is-a knowledge representation. If a person is-a human and is-a graphic object, then the person instance inherits the traits of a human and can be drawn on the screen. What is wrong with that? Why doesn't Java do that?

DU: Even worse, the relationship of an is-a can only be simulated with an interface that lies as a super prototype for all drawable objects. When this interface is changed, all the subclasses may have to be recompiled. This is called the fragile base-class problem.

PG: You have to admit that the lack of multiple inheritance removes some of the problems that students were having with C++ and leads to fewer programming errors. Although a class can be declared to implement multiple interfaces, it can only inherit from one implementation class [Singhal and Nguyen].

PB: It may be OK for beginning Java students, but it is a disadvantage for advanced Java programmers!

PG: Not really, if you want to share implementations, you should use the *delegation* pattern, not inheritance. Thus, the relationship is changed so that a class will invoke an instance of another class that supports the implementation of the method. This is better for two reasons. The first is that multiple inheritance should only happen on a *pure interface definition* not an implementation. This is consistent with CORBA IDL and Microsoft's OLE. It is by the use of interfaces that a translator (like *javah*) is able to generate headers in other languages. The second reason is that the delegation pattern is an extreme example of *composition*. Composition permits dynamic invocation. In other words, the inheritance structures and implementations are defined at *compile time* with Java. However, if you use the delegation pattern, you may alter the implementation at run-time.

```
public class Face implements Drawable {
      private Drawable faceElements;

      public void setDrawableElements(Drawable fe) {
          faceElements = fe;
      }

      public void draw() {
        faceElements.draw();
      }
}
```

In this example, we see how the class *Face* delegates the implementation of the *draw* method to *faceElements*. The only inheritance is in the interface. This must be what the designers of Java were thinking when they selected not to include multiple inheritance [Gamma et Al.].

B.4. Arrays can be C-style or Java style

PG: Arrays in Java can be either C-style or Java style. That means you can either write:

```
int a[][];
```

or

```
int [][] a;
```

This can really help when you are returning arrays from a method. This is really good!

DU: Yes, you can also mix the styles on the same line:

```
int [] a[];
```

This really does put the UG in Ugly!

PG: Actually, the mix of styles could be eliminated; all Sun would have to do is deprecate the practice. You can write bad code in any language.

B.5. Platform Independence: Compile Constantly

PG: One of the more highly touted features of Java is its portability. We can have one set of class files that run on any virtual machine.

PB: If you compile under a new API and attempt to run your program on an older JVM, you will get a *class not found exception*.

PG: That is exactly right! It is good that you get that exception. What the exception tells the programmer is: “No, that feature is not here!” So yes, there will be API’s that we invoke, and when they are missing, we must react.

PB: Sun has a nasty habit of *deprecation*. The dictionary defines deprecation as a mild expression of disapproval. However, for Sun there is more to it. Not only will the method be disapproved of, but future support may be withdrawn. Thus your code can break.

DU: In fact, if you want to avoid Sun deprecating your code, you must rewrite for every change in the API, and many of the changes add no real features. For example:

```
public void goAway(FileDialog fd) {
  fd.hide();
}
```

is now:

```
public void goAway(FileDialog fd) {
  fd.setVisibility(false);
}
```

It almost looks like Windows code.

PB: So the new motto should be: Rewrite (for each version of the API) recompile (if you are lucky) and run anywhere (the new API is supported).

PG: But it really is compile-once, run everywhere, once you have debugged the program for all the platforms.

PB: Actually, there is delay in execution as the just-in-time compiler kicks in. Thus a byte-code file is *recompiled* every time it is executed. SO we are really compiling once, and then again for each invocation.

B.6. New APIs support multi-media

PG : Sun is starting to support all the latest multi-media technologies: speech, sounds, image processing, 3D. Why I could teach all my courses in Java!

PB: Very nice, but I still can not write to a serial port.

DU: And, most of the API's that you mentioned are non-core. That means that they will not have been implemented in most browsers. It will be sometime before they are available on many platforms.

PB: Before you can teach these things in a course, you had better have a textbook that covers the topic.

DU: That is why I love Java; every time Sun writes a new API, I get to write another book!

B.7. The Virtual Machine

PG: The Java virtual machine is great. I don't have to program in Java at all. I can just write a compiler for any language I like (for example, SmallTalk) then compile it to run on the virtual machine. Thus, the virtual machine ends up being a means towards writing portable programs in *any language*!!

PB: But you are always at the mercy of the vendors. Who knows if your virtual machine will be implemented correctly? For example, one popular product causes the following error:

```
public class TrivialApplication {
public static void
          main(String args[]) {
      double k[][] ={
          {0,1},
          {2,3}
        };
        System.out.println("k[0][0]="+k[0][0];
System.out.println("k[0][1]="+k[0][1];
System.out.println("k[1][0]="+k[1][0];
System.out.println("k[1][1]="+k[1][1];
      }
}
The output printed is:
      k[0][0]=Infinity
      k[0][1]=1.0
```

```
k[1][0]=2.0
k[1][1]=3.0
```

Please note that k[0][0] was set to 0!

This error was acknowledged and I was told that I should use another JVM, that the JVM I had was not going to be fixed...ever! The compiler vendor said that the computer manufacturer was going to provide JVM's from now on. But the computer manufacturer had a critically flawed product (i.e., full of bugs, or the company is being sued for non-compliance, etc.). Sun does not support distributions of all platforms from its site. They only support JDK for Windows 95/NT and, of course, Sun workstations.

B.8. Javadoc

PG: Javadoc is an excellent part of the JDK. It allows me to generate HTML code automatically. This is really an improvement over C and C++ and is sound software engineering.

PB: Actually, HTML is only recently able to document equations. If you really want to document an algorithm, you still need graphics and equations as well as everything you would expect to see in a formal paper.

DU: In fact, Java is missing the dynamic feedback information that was available from the Symbolics Lisp Machines running ZetaLisp. Typically, we used to get documentation in the ZeMacs editor at a key-stroke. We received method completion and even a bit of human written documentation on the methods' arguments. And, this technology is decades old!

B.9. Performance

PB: Java is slow. Its high level of abstraction pushes it further and further away from machine code. In fact, we can't even know how arrays are stored internally, row-major or column major [Tyma].

PG: Java's Just In Time (JIT) compilers are alleviating the problem by changing Java executable into native code before it executes on your machine.

PB: Java also imposes MALLOC type overhead when using "New".

DU: In fact, benchmarks show that making a new instance of a complex number slows is six times slower than using an existing instance. When taking a Fast Fourier Transform [Lyon and Rao], we found it is always worth while to keep large arrays of primitives for both the real and imaginary parts of a complex number. Thus, the speed considerations alone defeat the object-oriented nature of Java because it takes so long to make an instance of a new object.

PG: Java chips will address all the problems that we are having with speed. We have even seen a RISC technology announcement with a four stage pipe-line and variable length instructions. The chip will be call PicoJava and this is going to be a rocket!

PB: This is old news. Sun was working on this technology two years ago. And if they had it two years ago, it would have been hot. Now it seems a little too late!

DU: Perhaps the only impact will be in embedded systems. But Sun has never set a delivery date for chips.

B.10. Other languages

PG: Have you seen the new Java Native Interface? This lets you integrate Java code with non-Java code.

PB: Yes I have seen it. Linking to native libraries is still not easy. The translation from a C struct, for example, is time-consuming [Lyon and Rao].

DU: Java lacks layout compatibility and this means conversion between the internal data structures of Java and those of other languages is required.

PG: Well, everything should be rewritten in the new language anyhow.

PB: But this makes it hard to create a gradual transition from old code to new code.

B.11. Conclusion

Java was not designed as a teaching tool. Even so, Java has been widely adopted for teaching. This is due, in part, to its object-oriented character, strong typing, portability and availability for Web applications. Java is not a perfect language. Several design decisions, that have not been laid open for public scrutiny, may have been made for the sake of quickly bringing a product to market.

Java technology combines language + virtual machine + API + hardware. Unlike most other languages, Java technology represents a system that is still relatively immature, particularly since the API (over 1,900 classes with over 21,000 members) changes frequently. So far, the Java hardware has not made it to the market place, but that may change. In addition, Java's isolation of the programmer from the hardware is both a source of benefit and hardship. If the objective is to write programs that run fast, Java is probably the wrong tool, at the moment. If the objective is to teach a first language to students, then faculty are cautioned to provide a simplified API for use. By understanding the good, bad and ugly about Java, computer science faculty can judiciously guide their students towards better programming practice and software engineering methodology.

B.12. Acknowledgements

This work was made possible, in part, by a grant from the National Science Foundation, DUE-9451520 and by a grant from the Educational Foundation of America.

B.13. Bibliography

[Gamma et Al.] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns Addison-Wesley, Reading, MA. 1995.

[Gordon] Gordon, R., *Essential JNI: Java Native Interface*, Prentice Hall, Englewood, NJ. 1998.

[Sun] http://java.sun.com/pr/1998/05/spotnews/sn980520.html

[Lyon and Rao] Lyon, D. and Rao, H., *Java Digital Signal Processing*, M&T Books, NY, NY, 1998.

[Ralston and Reilly] Ralston, A., and Reilly, E., *Encyclopedia of Computer Science and Engineering,* Second Edition, Van Nostrant Reinhold Co., NY, NY. 1983.

[Singhal and Nguyen] Singhal,S. and Nguyen, B. "The Java Factor" *Communications of the ACM,* 41(6): 34-37, 1998.

[Tyma] Tyma, P. "Why Are We Using Java Again?" *Communications of the ACM,* 41(6): 38-42, 1998.

