The Dark Side of Java

Conrad Weisert, Information Disciplines, Inc.
1205 Madison Park, Chicago, Illinois 60615
29August, 1997

I like Java. I use it for real-world applications. I've chosen it as the vehicle for an introductory course in object-oriented programming (OOP). I can do things with Java that I can't do with other tools.

But Java is far from the perfect tool that almost every article and presentation has led us to expect. Some of its flaws and shortcomings are minor irritations, but others impose serious obstacles not only upon software development and maintenance but also upon learning.

The "Java community" mentality

Java is new. We're willing to put up with shortcomings in a *new* tool, expecting them to get fixed eventually. But with Java that's less certain because of attitudes held by some of its zealous promoters, who:

1. present Java's major omissions as virtues,
2. justify its most annoying restrictions as protections against misuse,
3. explain a host of inconsistencies as contributions to simplicity,
4. assert that anything Java prevents you from doing can't be worth doing[1].

Anyone who believes such things is hardly likely to make proposals or initiate actions to fix the problems.

I recall no other language whose inner circle concerned itself so much with constraining my programming style. Such constraints might be tolerable if they were reasonable, but the Java community[2] is simply *wrong* about much of this. Many of their prescriptions are at odds with established principles of programming practice[3], which OOP has most certainly not repealed.

Say "No" to primitive data types

When we define an *elementary item* class in C++ we try to make it look as much as possible like one of C's built-in data types. Thus, we might declare:

```
double unitPrice;
```

and later, upon finding or building a good **`Money`** class, change it to:

```
Money  unitPrice;
```

confident that references to **`unitPrice`** in assignments, comparisons, arithmetic operations, or function parameters that worked before will still work. In Java we have the opposite extreme: Almost *none* of those references will still work, and we'll have to change almost *every* reference to **`unitPrice`**!

The problem is that Java contains two nearly separate *expression languages*: one for *primitive* (built-in) data types, the other for *reference* (objects of programmer-defined or library classes, as well as arrays) data types.

Various Java enthusiasts suggest three drastic remedies:

1. Just avoid primitive types for application-domain data. Java provides library classes that wrap the representation of primitive data, e.g. integers, in the syntax and semantics of reference data. That's a high price, since reference data leaves much to be desired in convenience, intuitiveness, readability, and for the beginner, ease of learning.
2. Use *only* primitive data types for elementary items, i.e. forgo the benefits of object-orientation for things like amounts of money, dates, distances, and weights. The benefits they ask us to forgo include not only type safety, but also the localization of the choice of data representation, one of the major advantages of OOP. (Haven't they heard of the "Year-2000 crisis"?)
3. Continually convert back and forth between objects and some publicly known external representation.[4] This clumsy technique manages to combine disadvantages of OOP with disadvantages of non-OOP.

I explore this issue further in a companion paper, "Conventions for Arithmetic Operations in Java". which puts a positive face on a bad situation by proposing a standard way of making the best of it.

More non-localization of type information

Worse yet, even among the built-in primitive types, Java violates a basic principle of language design by requiring the programmer to repeat type dependencies throughout the program. Consider these four declarations from a **`Calendar`** package:

---

[1] We encountered strikingly parallel attitudes in the Cobol community of the 1970's. ("But I've never *needed* to pass a file name to a subroutine.")

[2] There's no such body, of course, as a "Java community", but it's a convenient term. I don't mean open-minded professional colleagues who use and like Java, but only those who aggressively exhibit the attitudes described above.

[3] A 20-year-old classic, Glenford Myers: *Composite Structured Design*, Van Nostrand Reinhold, 1978, ISBN 0-442-89584-5, still offers one of the best summaries of those principles.

[4] A technique also common among beginning *Smalltalk* programmers. Accessor functions called **`value`** or **`getValue`** usually indicate the use of this technique.

```
short daysPrYr    = 365;
short daysPr4Yrs  = daysPrYr   *  4 + 1;
int   daysPrCent  = daysPr4yrs * 25 - 1;
int   daysPr4Cents= daysPrCent *  4 + 1;
```

One of those four lines is illegal in Java[5]; which one? Why? How can the programmer fix it?

Surprisingly, the problem is on the second line. For reasons having to do with attributes of literal constants and the result of arithmetic operations, the programmer must, to get the code to compile, code an explicit *cast*:[6]

```
short daysPr4Yrs  =
        (short)(daysPrYr * 4 + 1);
```

That's no minor notational irritation; it's a major impediment to maintaining large programs. We preach the virtues of *localizing* knowledge in a program, a key part of *modularity*. For more than three decades every mainstream programming language has supported the separation of data declaration from data manipulation. Now, for no good reason, we're told that the program must repeat the type whenever it operates on a data item.

"We can live with that," says the C programmer. "Just use **`typedef`** to localize the type. We'll still have to code the casts, but at least we'll only have to alter one line of code if it ever changes." Alas, not only is there no **`typedef`** in Java, but Java gurus assure us that this omission is a positive feature!

"Well, OK, but still no problem. Just make it a preprocessor (**`#define`**) name." Sorry, no luck there either. Java wants to protect us from all those ugly misuses of **`#define`** from misguided C programmers.

Should we just outlaw the **`short`** (16-bit) integers and any other primitive types that cause this problem?

Pointers and references

Java avoids C++'s most cumbersome and error-prone feature: the need to manage memory through explicit pointer manipulation. This alone ought to render Java more attractive as a teaching language than C++.

But Java hasn't actually done away with pointers, only with explicit pointer manipulation. Pointers are still very much present in the guise of *reference* data, and the programmer must be constantly aware of them. They affect the syntax and semantics of:

1. assignment
2. comparison
3. parameter passing
4. arithmetic operations

Library classes

In characterizing software I reserve the term "abomination" for truly extreme examples. To my dismay one of the first Java library classes I looked at turned out to qualify: the **`Date`** class. It's bad in so many ways, including blatant violation of OOP principles, that it takes a full-page course handout to describe its serious and amazing flaws.[7]

I cite this example not because I'm worried about difficulty in manipulating dates; after all, we can always develop or find some other **`Date`** class. We have to worry, however, about the competence and the attitudes of the people we're depending on to define a vital tool. It's alarming enough that a mainstream software developer would propose such an atrocious component, but it's truly appalling that so many in the Java community, including textbook authors, would then embrace it and present it unapologetically.

Obstacles to packaging reusable code

That example illustrates another Java shortcoming. On discovering that the library **`Date`** class was useless, I developed my own, modeled after a similar C++ class. In the C++ version I had captured, using a combination of macros and source file inclusion, a reusable pattern[8] of arithmetic operations common to many numeric types. So far I've found no way of doing that[9] in Java; I have no choice but to repeat the code pattern every time I need it. If I ever want to make a correction or an improvement, I'll have to hunt through a daunting collection of source code files.

Java zealots advise me to use an *interface* specification, here, but that's only an enforcement mechanism, not a way of packaging actual code for reuse.

---

[5] To conserve space we've omitted the usual qualifiers **`final`**, **`static`**, and possibly **`public`**.

[6] In this case, with a constant value that's never going to change, it was more expedient just to code:

```
short daysPr4Yrs = 1461
```

but we still have the problem wherever any program uses this item.

[7] These comments refer to the **`Date`** class in JDK 1.0, now completely redone (but not a lot better).

[8] See my article: "Point-Extent Pattern for Dimensioned Numeric Classes", ACM *SIGPLAN Notices*, November, 1997.

[9] As named functions, of course, not overloaded operators. (Java's **`Date`** class doesn't support date arithmetic at all!)

Java is actually quite good at reuse of *entire classes* -- that's inherent in the Java environment. When it comes to reusing smaller building blocks, however, it's weaker than even Cobol.

Pseudo classes and pure object orientation

Java (like Smalltalk) enthusiasts criticize C++ for being a hybrid language that permits independent functions. "In Java, *everything* is in a class," they proclaim, "so Java is a *pure* object-oriented language that enforces the object paradigm.

Nonsense. Calling a construct a *class* doesn't make it one. Yes, every Java function has to be defined inside a "class" definition, but what's the point of a class for which we neither instantiate objects nor derive other classes? It's simply a packaging artifice, one that can actually complicate *procedural* programming.

Java's standard **`Math`** and **`System`** classes illustrate such *pseudo classes*. There are no **`Math`** or **`System`** objects. We never derive subclasses from **`Math`** or **`System`**. **`Math`** and **`System`** never enter into polymorphism, or any other aspect of OOP. Yes, **`Math`** and **`system`** inherit from **`Object`**, but what useful effect does that have?

Java as a teaching language

Such pseudo-classes are another source of confusion for students. After we teach them about *abstract data types* (ADT) and show them that a class is Java's way of implementing an ADT, we have to explain that a lot of everyday "classes" have nothing at all to do with ADT's.

The separate and unintuitive *reference* data expressions undo a lot of the benefit of concealing pointers. After introducing our students to C's operator-rich expressions, we then have to tell them to forget all about most of it when dealing with objects, and to learn a separate clumsy and unnatural syntax. Experienced professionals cope with this duality, even if they don't like it, but beginners find it extremely hard to digest and a source of continuing confusion throughout a course.

As an instructor I've felt not only frustrated by these difficulties but also embarrassed in front of my students, many of whom are undecided about pursuing a career in our profession. I want them to respect the leaders of our field. When I keep having to explain to them how to circumvent poor tool design that comes from industry leaders, I can't help undermining that respect.

So which language, Java or C++, is better suited to teaching the object-oriented paradigm?[10] For programmers of limited experience, I still favor Java, mainly to avoid spending hours on the details of C++ memory management, a topic irrelevant to OOP. However, Java's peculiarities make the choice a closer one than I originally hoped, and the choice may depend more on the length of the course than on the background of the students. A 10-week quarter doesn't allow enough time to go into C++ pointer manipulation, and Java is the clear choice. In a 16-week semester, however, we have the time to develop fluency in pointer manipulation and memory management, and the choice is much closer.

For students who are already fluent in C, we can cover *both* C++ and Java in 16 weeks. I've developed such a course, in which we use C++ to explore the object paradigm thoroughly, and then spend the last 4-5 weeks on Java, emphasizing how it differs from C++.

Summary:

Java's strengths are well known: a platform-independent object-oriented language suitable for server, client, and web based programs. Furthermore, it's *slightly* easier to teach to inexperienced programmers than C++.

Java's serious shortcomings, rarely mentioned so far in the press, are:[11]

1. A large program developed in Java will tend to be costlier to maintain than a program developed in a more structured and internally consistent language.
2. Except for complete classes, component reuse in Java ranges from awkward to impossible.
3. Teaching Java to inexperienced programmers demands confronting a number of clumsy and unnatural constructs.

Java will surely play a growing role, both in major software development and in teaching object-oriented programming. However, before making any commitment to Java, we must be aware of the trade-offs and make a rational choice whether to accept them.

---

[10] Note that *no* object-oriented language is appropriate for a first course in *programming*. See my paper "Learning to Program: It Starts with Procedural".

[11] The one Java shortcoming that does get mentioned often, overhead of interpretive execution, is rarely a serious concern. Machines are getting faster and compilers and interpreters are getting smarter.