Reprinted from *Proceedings of the 35th Annual ACM Southeast Conference* (April 1997), pp. 124–131.

The Case for Java as a First Language

K. N. King
Department of Mathematics and Computer Science
Georgia State University
Atlanta, GA 30303
knking@gsu.edu

Abstract – Java could well be the answer to the problem of choosing an appropriate language for the first programming course. This paper looks at the pros and cons of teaching Java, concluding that Java appears to have outstanding prospects for computer science education in general and the first programming course in particular. In particular, the paper argues that the properties that make Java a suitable Internet language also make it excellent for classroom use.

1. Introduction

Not too long ago, most computer science departments used Pascal as their introductory language. In recent years, however, we've seen a wide divergence, as many departments switched to C, C++, Ada, Scheme, and other languages.

Richard J. Reid of Michigan State University maintains a list of languages used in CS1 courses [11]. This list is updated semiannually; the latest version confirms that there is no longer a single dominant language. Instead, six languages each have a share of 10% to 30% of the market (Table 1). At least a dozen other languages are used as well, but none has more than a 2% share. Although there's no guarantee that the schools in the survey are representative of the thousands of colleges that teach introductory programming, this survey provides the best information currently available.

The lack of consensus concerning the introductory language has caused a variety of problems [3]. One set of problems affects computer science departments. Failing to agree on which language to switch to can cause paralysis and stagnation. A switch that's not widely supported can result in friction between colleagues, in extreme cases even causing highly qualified faculty to depart rather than teach a language that they personally abhor. A profusion of languages also makes it harder for departments to find qualified instructors and teaching assistants.

A plethora of introductory languages can also hurt students by making it difficult for them to transfer courses from one college to another or to get advanced placement credit. Imagine the difficulties faced by a student with a Scheme background, say, transferring to a school where C++ is the language of choice.

Table 1: Language Usage in CS1

Language	*Number of Colleges*	*Percentage of Total*
Pascal, Object Pascal	153	30.1
C++	87	17.1
Ada	74	14.6
C	51	10.0
Scheme	51	10.0
Modula-2, Modula-3	49	9.6
Others	43	8.5

Source: Richard J. Reid, CS1 Language List, 15th edition (October 15, 1996)

Another problem is fragmentation of the textbook market. When multiple languages are used in the introductory course, books designed to teach one language are often hurriedly translated to other languages, with mixed results. Some of the best books are available only for a single language, making them unusable by instructors teaching other languages.

Java, the new language from Sun Microsystems, could well be the solution to these problems. Java is a general-purpose, object-oriented language that's been in the news quite a bit over the past couple of years. Most of Java's press coverage emphasizes its client/server role, as a language for writing "applets" that are downloaded from a server and executed locally. But Java isn't restricted to writing applets; it works just as well for writing traditional single-computer applications.

This paper examines the advantages and disadvantages of Java as a teaching language, with the focus on using Java as the first language that students encounter at the college level. It pays particular attention to how Java stacks up against C++, the language whose use in academia is growing the fastest. In Reid's Spring 1995 survey, only 27 schools reported using C++ in CS1; in the latest survey, the total stands at 87. (The growth of C++ may have peaked, however; the number of schools reported to be using C++ increased by just five between April 1996 and October 1996.)

The perils of teaching C++ are well-known. There is widespread agreement among faculty that C++ needs to be prominent in the curriculum because of its widespread use in industry, but there is considerable doubt that C++ is a suitable introductory language. As Kölling and Rosenberg [7] put it, "We agree that a graduate must be a competent programmer in C++ or a similar widely used language. It is our firm belief, however, that experience with one year of a good teaching language and one year of C++ produces better C++ programmers than two years of C++." That paper argued for the creation of a new language for teaching object-oriented programming. Another paper [3] proposed several ways to attack the problem of choosing a first language. One proposed solution was to design a new language "with a C-like syntax, but without the problematic features of C." This paper will argue that no new language is needed—Java fills the bill.

2. Java as a Teaching Language

A good place to start is with the *The Java Language: An Overview* [14] (previously titled *The Java Language: A White Paper*), a well-known document from Sun Microsystems that lists eleven properties of Java. Although this document describes the significance of these properties from the standpoint of commercial programming, it might as well have been addressing the academic world, where they are just as important, if not more so. Let's review the properties, considering the relevance of each one to the introductory programming course.

2.1 Simple

Although Java resembles C++, it omits many of C++'s more confusing features, including the ones most likely to cause problems for beginners: pointers, operator overloading, multiple inheritance, and templates. Moreover, Java lacks many of the automatic type conversions that C++ performs.

At the same time, Java adds an important feature that simplifies programming: automatic garbage collection. Having the language handle storage management gives Java a big edge in introductory classes over languages such as C and C++, where releasing memory that's no longer needed requires programmer intervention. Garbage collection not only makes programming easier but also avoids the bugs caused by dangling pointers. In C++ programming, too much effort is spent on problems of memory allocation and deallocation. As Bergin [1] notes, "some reports from industry are that on large [C++] projects, half of the programming effort is spent in getting memory management right."

Overall, Java is a small language, closer in size to Pascal or C than Ada or C++. Java's relatively small size is a powerful argument in its favor as a teaching language. As one author puts it, "Possibly the most attractive feature [of Java] is the relative smallness of the language." [5]

2.2 Object-Oriented

The importance of introducing the object-oriented paradigm early in a student's program of study is increasingly being recognized. However, there's widespread disagreement over which object-oriented language to use. C++ is the most popular object-oriented language used to teach introductory programming, but it has plenty of critics. Some even argue that no existing object-oriented language (prior to Java, at least) is really suitable for beginners [6]. (This paper describes desirable characteristics for a beginner's object-oriented language. Although the paper predates the release of Java, the authors might just as well have been describing Java.)

Java also supports object-oriented programming, but with significant advantages over C++:

- Students *must* use objects. Only the primitive types are not objects. There are no stand-alone functions; all functions must belong to a class.
- Objects are always allocated dynamically and manipulated through references, thereby simplifying their semantics.
- Storage management is handled automatically, significantly reducing the difficulty of writing many classes.
- Fancy features, like operator overloading and multiple inheritance, are missing. Students have less to learn before writing useful classes, and they can concentrate on learning the object paradigm rather than mastering a host of esoteric details.

In its support for object-oriented programming, Java is closer to Smalltalk than C++. Smalltalk is even more object-oriented than Java, and it makes a good introductory language as well [17]. However, Smalltalk has a syntax that's difficult for beginners to grasp, among other drawbacks [6].

2.3 Distributed

With the growing importance of networking in general and the Internet in particular, students need experience in writing software that's network-aware. Java is unique among major languages in its support for networking, which includes classes for working with URLs and sockets.

Although most of Java's networking capabilities wouldn't be used in an introductory course, some of the simpler ones could make excellent examples. For example, Java makes it easy for programs to access specific URLs on the Web, allowing students to gain a better understanding of how the Web works as well as being able to write some rather interesting programs. Ambitious instructors could use the more advanced networking features to illustrate how programs cooperate over a network.

If Java is used in the second programming course as well as the first one, its networking support would be more likely to come into play. At one college, students use Java in the first two programming courses. When asked what was the most surprising thing about using Java, the instructor replied, “For me, the way in which CS2 students so readily adapt to the notion of building reactive, distributed programs on the internet, and how this, as much or more so than the object-oriented aspects of Java programming, so fundamentally governs their attitudes on what programming is all about.” [8]

2.4 Robust

A number of Java’s properties are the result of making the language safe for transmitting executable content over the Internet. These properties, as it turns out, are often the same properties instructors look for in an introductory language. As [14] puts it, “Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error prone.” This should be music to the ears of Pascal and Ada instructors who have resisted switching to C or C++ because of their relatively weak abilities to detect errors.

Here are some of the measures that Java uses to achieve robustness:

- ***No pointers.*** Although Java uses pointers internally, no pointer operations are made available to programmers. There are no pointer variables, arrays can’t be manipulated via pointers, and integers can’t be converted into pointers.
- ***Garbage collection.*** Thanks to automatic garbage collection, there’s no chance of a program corrupting memory via a dangling pointer.
- ***Strict type checking.*** Java’s type checking is much stricter than that in C or C++. In particular, casts are checked at both compile time and run time. As a bonus, type checking is repeated at link time to detect version errors.
- ***Run-time error checking.*** Java performs a number of checks at run time, including checking that array subscripts are within bounds.

2.5 Secure

In addition to being *robust* (resistant to programmer error), Java programs are designed to be *secure* (safe against malicious attack). Java’s run-time system performs checks to make sure that programs transmitted over a network have not been tampered with. The code produced by the Java compiler is checked for validity, and the program is prevented from performing unauthorized actions. For example, an applet that’s been downloaded from a Web page can’t access files on the local computer. Moreover, the nature of Java makes it hard to write viruses and other kinds of malicious programs. A program that can’t access memory locations via pointers will find it hard to do much damage. Instructors who have been stung by viruses in student programs will appreciate the security provided by Java.

2.6 Architecture-Neutral

The Java language is completely architecture-neutral. As a result, programs written in Java will run on any platform that supports the Java run-time system.

The significance of a multiplatform language like Java cannot be overstated. Sun’s Java Development Kit is available for a variety of platforms, including Windows 95 and NT, Macintosh, and Sun Solaris, all of which are widely used in education. Colleges can offer Java without having to worry about whether their labs contain enough computers of the same type. Students can easily transport programs from campus to home and vice-versa, even though their home computers may be different from the ones on campus.

2.7 Portable

Java programs are not only architecture-neutral but portable as well. One way in which Java achieves portability is by completely defining all aspects of the language, leaving no decisions to the compiler writer. Consider the issue of types. Most programming languages don’t define the exact ranges of types, allowing for variations based on the computer’s architecture. Java, on the other hand, completely defines the ranges and properties of all types. Values of the `int` type are always signed 32-bit integers; `float` values are stored in 32 bits using the IEEE 754 representation. That’s a plus for instructors, who don’t have to worry about trying to explain to beginners why a program may not work if compiled with a different compiler.

Other aspects of Java are portable as well. Java’s libraries are designed for complete portability. The Java system itself is portable. Sun’s Java compiler is written in Java itself; the run-time system is written in Standard C.

2.8 Interpreted

Java is usually an interpreted language. A Java compiler translates a program into bytecodes, which can then be executed by an interpreter. Linking is done at run time, with code loaded dynamically by the run-time system as needed. For students, this means that building a program is simple: there’s no linking step to perform. When one part of a program is changed, only that part needs to be recompiled, and there’s no linking step to redo.

From the standpoint of the instructor, the fact that Java is interpreted has two primary implications. One is that Java programs won't run at the same speed as programs written in a compiled language such as Pascal, Ada, C, or C++. For most student programs, however, the speed of execution is irrelevant.

A more important implication of interpretation is that students will be able to get excellent feedback when a program fails during execution. A C or C++ program that fails at run time generally doesn't provide any clue as to the problem; students are forced to crank up the debugger. A Java program that fails can print the call stack and describe the exception that caused the program to fail. That information alone is often enough to pinpoint the cause of the error, without the need to use a debugger. This behavior is possible thanks to information about the source program that's embedded into the bytecodes during compilation.

2.9 High-Performance

Programs written in interpreted, garbage-collected languages often don't execute at high speed. In Java, however, the performance penalty isn't as bad as in some languages. One reason for Java's superior performance is that garbage collection is done by a separate low-priority thread. That way, garbage collection takes place primarily when the program has nothing else useful to do—while it's waiting for user input, say.

Greater speed can be achieved by translating Java's bytecodes into native machine instructions. This can be done by translating the entire program to native code prior to execution, or it can be done on the fly by a "just-in-time" (JIT) compiler. JIT compilation is becoming a standard feature of commercial Java environments; both Borland C++ 5.0 (which supports Java) and Microsoft J++ 1.0 provide JIT compilers. When translated into native code, Java's performance "is almost indistinguishable from native C or C++" [14]. Java's support for translation to machine code is one of the features that gives it an edge over Smalltalk, another interpreted, garbage-collected language.

2.10 Multithreaded

Unlike most major programming languages (with the notable exception of Ada), Java has built-in support for multitasking. A Java program may create any number of threads, which appear to execute in parallel.

For instructors, Java's support for threads provides a golden opportunity to introduce students to the concept of concurrency, a topic that's already important and will only become more so in the future, with multiprocessor PCs soon to be commonplace. It is imperative that students become comfortable with concurrency early in their studies. Java's model of concurrency is simple enough that even beginners can use concurrency effectively.

Instructors teaching concurrency often use Ada, one of the few mainstream languages to provide built-in support for concurrency. However, Ada provides no support for GUI interfaces, without which writing simple concurrent programs is difficult. Ideally, a concurrent program should support multiple input sources and multiple output destinations, to avoid problems of mutual exclusion. Doing this in Ada is not trivial. Java's support for GUI interfaces makes it a snap to write programs that illustrate concurrency. Entering keyboard input into a concurrent program is tricky, because only one task can read from the keyboard. With separate windows for tasks, however, input is easy. Similarly, writing output to the screen becomes an exercise in mutual exclusion in Ada; a Java program simply writes to different windows.

Knowledge of threading isn't required to write Java programs, so instructors who wish to skip it in a first class may easily do so.

2.11 Dynamic

Java is designed to accommodate the fast-paced, modern world of software development, in which components of a system may change on a regular basis. Java's run-time linking guarantees that a program always loads the most recent version of its library modules. (That's good for students, who often forget to relink and end up running older versions of their programs.) It also reduces recompilation by making it possible to add methods and instance variables to a library without having to recompile its clients.

3. Java's Support for GUI Programming

With graphical user interfaces now standard among everyday software applications, it seems odd to continue to teach students to write only programs that perform character I/O. Yet that is the norm in many introductory computer science courses. One reason for this situation is that graphical user interfaces are traditionally tied to a particular platform or even a particular software vendor. Another is that writing programs that use these interfaces is an arduous task.

Various solutions have been proposed to the problem of introducing GUI programming into the computer science curriculum. One solution is to let students write GUI programs in a proprietary language like Microsoft's Visual Basic or Borland's Delphi. Another is to develop simplified libraries that shield students from the complexity of a commercial GUI application interface [10, 16]. The alternative—asking students to write in C++ and master the Microsoft Foundation Classes (MFC) or Borland's Object Windows Library (OWL)—is difficult, and students won't progress with any speed [15].

Java changes all that. Java comes with a platform-independent windowing API, known as the Abstract Windowing Toolkit (AWT). Java's AWT is remarkably easy to use. A simple component, such as a button, can be created and added to the screen in one statement. Handling an event (such as that triggered when the button is pressed) isn't much harder. Using AWT, students can write GUI programs with ease, and their programs will run on a variety of platforms. Once they're familiar with the event-driven nature of GUI programming and the basic components from which GUI programs are built, they'll be better prepared to tackle the complexities of MFC or OWL in a subsequent course.

AWT supports not just GUI components, but graphics and sound as well. The advantages of using graphics in introductory classes are well documented [12]. With Java, it's easy for beginners to incorporate graphics, animation, and sound in their programs.

4. Other Advantages of Java

Besides those mentioned in Sections 2 and 3, Java has some other advantages as a teaching language:

- ***Low cost.*** The tools needed to build and test Java programs are available without charge. Sun makes the Java Development Kit (JDK) available over the Internet (at *www.javasoft.com*), where faculty and students alike can download it. The JDK—which includes the Java compiler and interpreter, among other tools—is admittedly spartan, but students should find it adequate for most programming assignments. Those willing to spend a little money will find nicer program development environments (such as Symantec Café and Microsoft J++) available at moderate prices.
- ***Easy to test.*** Students can put their programs—written as applets—on their Web pages for instructors to test and critique. Instructors can monitor a student's progress at any stage by simply visiting the student's Web page.
- ***Student enthusiasm.*** Java has gotten so much publicity that students are bound to be excited about learning it. By harnessing that enthusiasm, instructors can use Java as a vehicle to teach students a tremendous amount about modern-day computing. Students will be motivated by Java's growing importance in the "real world." Moreover, students will be thrilled by the ease with which they can build sophisticated GUI programs.
- ***Suitable for advanced courses.*** After students gain familiarity with the basic features of Java in CS1, they can use its advanced features in later courses. For example, a course on operating systems can take advantage of Java's support for threads. The network classes that come with Java make it ideal for a networking course.
- ***Easy transition to C++ and other languages.*** Java's syntactic similarity to C and C++ should ease the transition to those languages. We'll return to this point in Section 6.
- ***International appeal.*** The Unicode character set is an integral part of Java, allowing students to learn about the issues of developing software for the international market.

5. Disadvantages of Java

Like any programming language, Java is not without drawbacks. Section 2.8 mentioned one of them: Java is an interpreted language, so programs written in Java won't be speed demons. Still, for most programs that students write, speed is secondary. With ever-faster computers available at bargain prices, Java should be fast enough for all but the most time-intensive programs.

Other problems stem from Java's youth. For one thing, Java isn't available on all platforms. Although Java will likely spread to more as time goes on, it's unlikely to be implemented on older platforms. That will cause problems for institutions that lack funds for hardware and software upgrades. Still, that barrier should be only a temporary one.

Java is relatively immature. Although the language itself is unlikely to change dramatically, we can expect significant changes in the Java API and associated technology. (The latest version of the Java Development Kit, JDK 1.1, is currently in beta. It features a host of changes to the API.) Still, C++ has been in a state of almost continuous change since it was created—the official standard isn't expected for another year or two—but that hasn't stopped industry and academia from adopting it.

Java is also hampered by a shortage of genuine textbooks. This problem should also take care of itself within a short time. The tremendous sales of Java books in the trade market will undoubtedly spur publishers to make Java texts available as well. *Java How to Program* [4] is the first introductory textbook; *Java Software Solutions* [9] is due out in 1997.

The syntax of Java can be criticized. Making Java resemble C and C++ was a shrewd move on the part of Java's designers: the language looks familiar to most software developers, making it easy to switch to. On the other hand, Java has inherited some of the quirks and traps of C and C++. For example, the placement of semicolons can easily trip up the beginner. The "dangling `if`" problem is present in Java; most modern languages have eliminated it. C's `/*` … `*/` comment style is supported, allowing students to accidentally "comment out" parts of their programs.

Another criticism of Java is that it lacks certain features that are desirable for teaching introductory courses. Hosch [5] provides a list of such features:

- ***No separation of specification from implementation.*** Java classes aren't divided into specification and implementation parts. Hosch feels that this is an important point for beginners and would like “this distinction between specification and implementation to be supported by the syntactic structure of the language.” Lacking such a separation, he would at least like to be able to write prototypes for methods, which Java doesn’t allow except within an abstract class or interface.

- ***No preconditions and postconditions.*** Hosch’s introductory course emphasizes preconditions and postconditions, for which he would like language support. Java, like most languages, has no such support. (Among major languages, only Eiffel does.)
- ***Visibility rules are “baroque.”*** Hosch decries the many types of visibility in Java. He also laments that Java’s syntactic support for hierarchical packages doesn’t carry any semantic significance.
- ***No support for genericity.*** Hosch notes Java’s lack of support for writing generic data structures and methods. As a substitute, he would accept “type by association,” such as Eiffel’s anchored types.
- ***No enumeration types.*** Java lacks enumeration types entirely, although they can be simulated by creating a series of named constants.
- ***No local constants.*** In Java, variables that belong to a class can be made constant by declaring them to be `final`. Variables that are local to a method cannot be declared `final`, however.
- ***Exceptions not caught within a method must be declared as thrown by that method.*** Hosch finds this requirement to be onerous: “for introductory students, it’s a sequence of ugly, unintelligible syntactic marks.”

To some instructors these criticisms may be a major concern; to others they are likely to be minor quibbles. None of Hosch’s complaints would dissuade me from using Java in an introductory class. In particular, his complaints about visibility and hierarchical packages seem to be of little consequence in an introductory course.

Hosch’s criticism of exceptions appears to be based on a misunderstanding of this feature. Not all exceptions need to be declared as being thrown by a method. Many of the most common exceptions, including `ArithmeticException`, `IndexOutOfBoundsException`, and `NullPointerException` are exempt from this requirement. Only exceptions that the programmer should have handled but didn’t are required to be declared as thrown by a method. This feature actually has a pedagogical advantage, since it enables the compiler to point out exceptions that students have overlooked.

Bergin [1] complains that Java lacks templates, overloaded operators, and user-defined conversions. The latter two, he points out, are useful for making a user-defined type behave like a built-in type. Like Hosch, he also laments the lack of separation between specification and implementation.

6. After Java, What?

Java won’t be a universal language anytime in the near future; students need exposure to other languages as well. That raises a few questions:

- If Java is the first programming language, where should the transition to other languages take place?
- What languages should follow Java?
- How difficult will it be to move from Java to other languages?

C++ is well-established in commercial software development, so it obviously needs to be taught. C remains the language of choice for low-level, close-to-the-metal programming, so it should be in the curriculum as well. Smalltalk is increasingly popular in the business world, so it’s also a likely follow-up to Java.

C++ is the language most likely to be taught after Java. Learning C++ exposes students to issues that Java hides, including pointers and memory management. Introducing C++ during (or just prior to) a data structures course might be appropriate. C++ and Java are similar enough that students can pick up C++ in a fraction of the time it would take without prior knowledge of Java. Students will view C++ as a more baroque version of Java, spending most of their time learning pointers and memory issues, as well as mastering multiple inheritance, templates, and other advanced C++ features.

A logical place for C would be in a class that also covers UNIX programming, since UNIX system calls are done using C and since many UNIX tools use a C-like syntax.

The transition to Smalltalk should be easy regardless of where it occurs in the curriculum. The main issues will be syntax, a different class library, and the need to learn a different kind of program development environment.

Java has enough in common with C, C++, and Smalltalk that the transition to any of these languages should be relatively painless. In many cases, it won’t be necessary to spend an entire course on the transition; instead, it can be managed at the beginning of another course. (A UNIX course, for example, could spend the first couple of weeks on C.)

7. Experiences So Far

So far, few colleges have tried using Java in an introductory course, and even fewer have reported their experiences. Sun Microsystems maintains a Web page listing academic institutions that teach Java [13]. According to this page, there are 102 colleges known to teach Java as of January 2, 1997, of which 60 are in the U.S. However, only one (Washington University) is identified as "committed to teach Java as a first language." This isn't surprising, given the newness of the language and the lack of teaching materials. Most colleges are still in the experimentation phase, as instructors learn Java and evaluate its potential place in the curriculum.

One of the first colleges to teach Java was the State University of New York (SUNY) at Oswego, where Java was used in the introductory programming course during the fall of 1995. The experiences of the Oswego instructors—summarized on a Web page [8]—were positive, an encouraging sign given the primitive state of Java tools and the paucity of textbooks. In particular, they report less attrition in the course than when C++ was the introductory language.

Other feedback from faculty who have taught Java appears in a recent *JavaWorld* article [2]. According to David Dobkin of Princeton, "Java seems to correct some of [the] flaws [of C++], and we are now coming to believe that Java can be made to work as a first programming language." Roger Whitney of San Diego State says that, teaching Java instead of C++, he has been more successful in getting students to write modular code. Whitney also notes that Java enables him to cover "material, ideas, and concepts that would not be possible with C/C++." To Doug Lea of SUNY, that aspect of Java is crucial. "Concepts such as distributed computing, component-based design, and theoretical issues in concurrency, distribution, and reactive design used to be reserved for advance [*sic*] courses but now need to be introduced in the first few computing courses."

Georgia State's Department of Mathematics and Computer Science has not yet tried Java as an introductory language, although it's being used in other courses. As a result of a recent curriculum redesign, we will soon switch to a breadth-first introductory course (with no programming in a "real" language), followed by a Java programming course. We plan to move students to C++ in the second programming course, so that we can discuss pointers and other issues that don't arise in Java. A later course will give students experience with C in the context of UNIX system-level programming.

8. Conclusion

Java has significant advantages not only as a commercial language but also as a teaching language. It allows students to learn object-oriented programming without exposing them to the complexity of C++. It provides the kind of rigorous compile-time error checking typically associated with Pascal. It allows instructors to introduce students to GUI programming, networking, threads, and other important concepts used in modern-day software.

Java might well be a language that most computer science departments could agree to use as an introductory language. If so, we'll all benefit from once again having a single dominant language in CS1.

References

[1] Bergin, J., "Java as a better C++," *ACM SIGPLAN Notices* **31**, 11 (November 1996), pp. 21–27.

[2] Bowen, B. D., "Educators embrace Java," *JavaWorld* (January 1997), *http://www.javaworld.com/javaworld/jw-01-1997/jw-01-education.html*.

[3] Brilliant, S. S., and T. R. Wiseman, "The first programming paradigm and language dilemma," *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education,* Philadelphia, February, 1996, pp. 338–342.

[4] Deitel, H., and P. Deitel, *Java How to Program,* Prentice-Hall, Englewood Cliffs, N.J., 1997.

[5] Hosch, F., "Java as a first language: an evaluation," *ACM SIGCSE Bulletin* **28**, 3 (September 1996), pp. 45–50.

[6] Kölling, M., B. Koch, and J. Rosenberg, "Requirements for a first year object-oriented teaching language," *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education,* Nashville, March, 1995, pp. 173–177. Published as *ACM SIGCSE Bulletin* **27**, 1 (March 1995).

[7] Kölling, M., and J. Rosenberg, "Blue—a language for teaching object-oriented programming," *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education,* Philadelphia, February, 1996, pp. 190–194.

[8] Lea, D., Some Questions and Answers about Using Java in Computer Science Curricula, *http://g.oswego.edu/dl/html/javaInCS.html.*

[9] Lewis, J., and W. Loftus, *Java Software Solutions: Foundations of Program Design,* Addison-Wesley, Reading, Mass., 1997.

[10] Mutchler, D., and C. Laxer, "Using multimedia and GUI programming in CS 1," *Proceedings of SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education,* Barcelona, Spain, June, 1996, pp. 63–65. Published as a special issue of *ACM SIGCSE Bulletin* **28** (1996).

[11] Reid, R. J., CS1 Language List, *ftp.cps.msu.edu:pub/arch/CS1_Language_List.Z.*

[12] Roberts, E. S., "A C-based graphics library for CS1," *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education,* Nashville, March, 1995, pp. 163–167. Published as *ACM SIGCSE Bulletin* **27**, 1 (March 1995).

[13] Sun Microsystems, Academic Institutions Teaching Java, *http://www.sun.com/edu/hot/java/javaschools.html*.

[14] Sun Microsystems, The Java Language: An Overview, *http://www.javasoft.com/doc/Overviews/java/*.

[15] Szuecs, L., “Creating Windows applications using Borland’s OWL classes,” *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, February, 1996, pp. 145–149.

[16] Wolz, U., S. Weisgarber, D. Domen, and M. McAuliffe, “Teaching introductory programming in the multi-media world,” *Proceedings of SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, Barcelona, Spain, June, 1996, pp. 57–59. Published as a special issue of *ACM SIGCSE Bulletin* **28** (1996).

[17] Woodman, M., and S. Holland, “From software user to software author: an initial pedagogy for introductory object-oriented computing,” *Proceedings of SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, Barcelona, Spain, June, 1996, pp. 60–62. Published as a special issue of *ACM SIGCSE Bulletin* **28** (1996).