Experiences Teaching Data Structures With Java

Mark Allen Weiss
School of Computer Science
Florida International University
Miami, FL 33199
weiss@fiu.edu

Abstract

This paper describes our experiences incorporating Java in a Data Structures course. We describe the features of Java that made for a more interesting course, the difficulties that we encountered, and compare Java to the prior languages used in this course, Ada and C++. All in all, we found Java to be a reasonable, but not overwhelming better, alternative. Our students were particularly happy with the experiment.

Introduction

Among Computer Science educators, hardly any topic inspires more heated debate than the choice of programming language in the introductory sequence. In the late 80s, the uniformly accepted choice was Pascal, but since then, a host of alternatives have come into use. C++ seems to have emerged as the winner, while Pascal, C, Ada, Scheme, and Modula-3 split most of the remaining market.

There appear to be two overriding reasons for C++'s emergence. First, principles such as encapsulation and information hiding, that are important to teach in the CS I/II curriculum, are easily demonstrated in C++. Much of the ugliness associated with C is easily avoided in C++ by the use of a tiny set of classes: About all that is needed is a `String` and `Vector` class. Second, C++ has become an industry standard (even though C++ is itself not yet standardized). Many universities are finding that they must teach C++ at some point, and given limitations on the number of courses that can be offered, they are finding it most convenient to teach it early. C++, however, has its share of problems; some of these problems will be discussed later.

Java is the new alternative to C++. It can be presented as a simpler C++ that fixes many of C++'s bad features and provides a primitive, but useful, GUI toolkit. One argument for teaching Java early is that it is better to use an already-defined language rather than attempt to subset a complex language. While C++ is arguably the most complex language ever to be widely adopted, it appears that Java is easily the most hyped language.

The Data Structures course at our University is a bit unusual. The vast majority of our students are upper-level transfers from community college. As a result, most come to us with much of their non-CS courses complete, and have taken an introductory programming course, typically in Pascal. The first course taken at XXX is formally entitled *Intermediate Programming*. It is taught somewhere between CS-1 and CS-2; we call it CS-1.5. Since 1990, the course has been taught in Ada. *Data Structures* follows Intermediate Programming. However, we also offer a course called *Advanced Programming*; it discusses C, C++, and object-oriented concepts.

Although Advanced Programming is not a prerequisite to Data Structures, many students elect to take it either prior to, or concurrent with, Data Structures. As a result, although Ada is the assumed language for Data Structures, many students want to program in C++ for the course. We have found that the language constructs of Ada and C++ are actually very similar, making this reasonable. Ada and C++ both have enough subtle "tricks" that some class time needs to be devoted to explaining both languages; this has become somewhat of an annoyance. One of the original motivations for using Java was to avoid the problem of dueling languages. Everybody would have to use Java in the course.

Aside from our own unusual sequencing, Java seemed to offer several other benefits. First, since Java comes with a built-in GUI toolkit (the *Abstract Window Toolkit*, or AWT) we could allow the students to experiment with both GUI (rather than line-at-a-time) input and graphical objects, such as circles. Students could draw binary trees on a canvas, thus extending the range of programming assignments to include demos of the basic data structures and algorithms. By having more complex projects, with fancier interfaces, it became reasonable to ask students to work in groups. This allowed us to introduce group work into the curriculum somewhat earlier than usual. Second, it was thought that students would be more motivated if they could work in Java, since many of them were also well-aware of the hype. The motivation factor would also increase because GUI programming is more exciting than terminal-based programming. Finally, there was the appeal that perhaps Java really was a significant improvement over C++.

The rest of this paper is organized as follows: First, we discuss the Java course materials that we used and developed. Then we compare Java and C++, noting fewer significant language differences than anticipated, and pointing out

some subtle traps. Next we discuss student reactions, which were very positive, and student performance, which was mixed. Finally we discuss our future plans, and explain how Java will fit into our curriculum.

How We Did It

The students did all the development using Symantec Café in a lab of 22 Pentium 60s (at the time Café was the only development tool available). The consensus seemed to be that Café worked well for the students, and the resource editor was easy to use. The online help was deemed worthless, however. Java compiles source code (*.java* files) and generates *.class* files; the students eventually uploaded the *.class* files to a UNIX web server to demo their applets.

In addition to the Data Structures textbook [4], which the students were instructed to purchase in either the Ada or C++ version, the students were also given some Java textbook options. Two books [1,3] were ordered. [1] is for beginners, and comes with a CD that contains Café Lite (Café minus a few features). The alternative was [3], which is a reference intended for more experienced programmers. The consensus among the students was that [1] was too elementary. The students that chose [3] were pleased. Much of Java is documented on the internet through Sun's home page. Few students chose the no-book alternative. Given that the preferred book was also half as expensive, I would recommend it for anyone teaching the course. Recently, a host of new Java books have appeared, including [2], which is geared for the CS-1 market.

Four lectures encompassing two weeks of the course were spent describing Java. The first lecture described basic Java, including the environment, the built-in types, operators, public static functions, strings, and arrays. The second lecture described objects, classes, packages, and the *javadoc* utility. The third class described inheritance and abstract methods and classes. It included exceptions (as in C++, an object is thrown by the throw clause, but unlike C++, only objects that are subclasses of `Throwable` can be thrown), interfaces (the pretty alternative to multiple inheritance), and a workaround for templates, which are missing in Java. The fourth lecture described the Abstract Window Toolkit and applets. No attempt was made to discuss threads, though many students were inspired enough to learn it on their own. Lecture slides are available at `http://www.xxx.edu/~xxxxx/cop3530`.

Although two weeks were invested early in the course, eventually they were made up because the Java programming that relates to data structures is a little simpler, and questions were limited to one programming language. The downside was that although the same material was covered, the material covered by the programming assignments did not include as much as usual (there were no programs on sorting and graph algorithms). Whereas the second assignment would typically cover stacks, queues, etc, these topics were not covered until the third assignment, because of the two weeks invested in discussing Java. The assignments that I eventually gave were as follows:

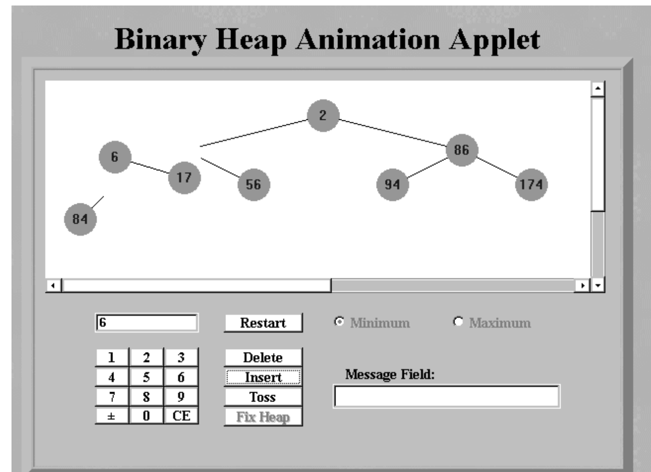

Figure 1. This applet shows the insertion of 6 into a binary heap. At this point, 6 and 17 are exchanging places to restore heap order (submitted by Peterjohn Hugh and Alberto de la Serna).

1. (Java warm up) Implement a `Date` class with a simple GUI to test subtraction of two `Date` objects.
2. Implement an applet that demonstrates the binary search algorithm.
3. Implement an applet that demonstrates the operator-precedence parsing algorithm, showing both the operator and operand stacks.
4. Implement an applet that prints a binary search tree after each insertion or deletion.
5. Implement an applet that demonstrates linear probing and quadratic probing hash tables.
6. Implement an applet that demonstrates the basic binary heap operations.

Students were allowed to work either by themselves or in groups of up to three. About half the students worked by themselves; the other half worked in groups of two. Students in groups started with a lower base grade and had to do a fancier job to get an equivalent grade. I also required students in groups to submit "timecards" documenting who spent what time doing what. Figure 1 shows a submission that was notable for using animation (animation was extra credit).

As we found during the course, the use of Java has both good points, bad points, (mostly) points that sound good, but really are not much better than C++, and also points that seemed like they would be bad, but were not. We list some of the entries in each of the four categories.

Java Wins

Several areas of Java are nice improvements to C++ and simplify life. There is no denying their utility.

The first improvement is that the C++ header files and class interface are gone. This removes the annoying `#ifndef` `#endif` idiom. The *javadoc* utility allows one to extract a meaningful documentation page from a properly commented java class. My students were also happy to know that java programs must end in *.java*, in contrast to C++ programs which have differing suffixes (*.cpp*, *.cc*, etc.), depending on the compiler and machine.

Another nice feature is that the basic types have fixed ranges. For instance, an `int` is always 32 bits, and always ranges from -214783648 to 214783647. This fixes an annoyance that seems to pop up when moving from SUN workstations to PCs. There are no unsigned types, which saves lots of problems. Additionally, we do not have to worry about “memory models” that are common for C++ implementations on PCs. In particular, one can have 20000-node binary search trees without claims of running out of memory (however Java is a lot slower than C++!).

Java provides a simple mechanism for using GUIs and drawing graphics. Although it is not as powerful as the MFC classes in Visual C++, it is certainly more than adequate for use in an early CS course.

Java Wins That Are Less Significant

Java has a number of improvements to C++ that, while nice, either have equivalent constructs in C++, or are not widely used in a Data Structures course.

Java defines the `boolean` built-in type, which makes many of the common C++ errors (for example `if(X=Y)`) go away. While there is no doubt of its utility, it is part of Ada and will be part of the new C++ standard, as `bool`. Many C++ compilers already support it.

Java has a predefined `String` object and an array object that includes bounds checking. So does Ada. C++’s standard has these incorporated, and in the interim, it is a simple matter to write two small classes for this. Java’s string object also has some problems. Since it is not a built-in type, the == operator does not return true if the two strings are equal, but rather returns true if they refer to the same object. (Our textbook had errors in which == was used instead of the `equals` method). The = operator for arrays is simply a reference assignment, rather than a copy of array elements. With C++ and Ada, strings and arrays can be made first-class types.

Java does not have pointers. This is billed as a monumental improvement. In reality, except for a built-in type, an object name is a reference, and thus is actually a pointer. Thus, in some respects, all Java **has** is pointers. While it is true that the avoidance of direct pointers will reduce programming errors, in our course the most typical use is in linked lists and search trees. Here one hardly sees a difference in the Java, C++, or Ada code. The other use of pointers is typically to simulate arrays and strings, but if the `Vector` and `String` classes are used, this goes away.

Java supports garbage collection, so one does not have to perform `delete` or `delete[]` or (in Ada) `Unchecked_Deallocation`. Once again, given that arrays and strings in C++ will use a class that hides pointer details, `delete[]` becomes unnecessary. The common use of `delete` is in dynamic data structures. Here, using `delete` typically requires an insignificant extra two lines of code (saving the pointer to the deleted object, and then freeing the object). Even if we accept that Java wins in this instance, Java is not uniform about collecting garbage: Programmers must remember to explicitly close open files in Java (in C++ they are automatically closed by a destructor when the file stream exits scope), and programmers using `Graphics` objects in Java must call a dispose routine explicitly.

Another win for Java concerns the related issues of copy constructors, initializer lists, parameter passing mechanisms, return mechanisms, and constant member functions, all of which are needed in C++, and none of which are part of Java. Java passes and returns all built-in types by value, and all other types (objects) by reference. Java does not have the notion of an initializer list or a constant member function. Constant member functions are a detail of C++ that students put on a par with commenting. Deciding on the correct parameter passing and return type among value, reference, and constant reference is confusing for beginning students, and at the very least, leads to a lot of typing.

Although Java simplifies C++ in this regard, professors can simplify C++ themselves with the following strategy:

1. Use only `String` and `Vector`
2. Do not use `delete` at all
3. Do not use initializer lists or `const` at all
4. Do not implement copy constructors or `operator=`.

My own opinion is that only strategy 1 should be adopted; the complexity of managing memory, using initializer lists, worrying about const-ness and writing copy constructors and `operator=` is not significantly overwhelming and must eventually be mastered by C++ programmers anyway. In many instances I simply disable the copy constructors and `operator=` by placing their prototypes in the private section of the C++ class interface.

Java’s implementation of inheritance is far superior to C++’s. All binding is dynamic by default; we do not have to clutter up the code with virtual functions. Multiple inheritance is not allowed, but an elegant alternative is provided

by the *interface*. However, inheritance is used relatively sparingly in a data structures course. Those contemplating an extensive object-oriented course might well find that Java's handling of inheritance is the single reason to choose Java over C++ or Ada.

Java Problems That Were Not Huge Problems

The most glaring omissions from Java are the lack of templates (or generics in Ada) and operator overloading. Since both these features are present in both Ada and C++, many of our students were surprised to find out that they were missing in a modern language like Java.

Not having operator overloading is annoying mostly because it means that the typical examples of rational or complex numbers can no longer be used seamlessly.

The lack of operator overloading also means that we cannot rewrite our own array class to incorporate resizing. The `Vector` class, which does resizing uses named methods `elementAt` and `insertElementAt` instead of `operator[ ]`. It's not an appealing alternative. Even so, writing the array doubling code ourselves requires only a few lines of code that can be made a public static method of a utility class.

What really seemed to be problematic was the lack of templates. Instead of writing a template stack, one writes a stack of `Object`. The idea is that since everything is an object, we can have a stack of anything. There are some problems though. First, not everything is an object (built-in types are the exception). However, Java provides wrapper classes for the eight built-in types. Second, when one does a `top` operation to access the top item in the stack, the return type is an `Object`, which must then be type-converted to what is actually on the stack. The syntax is thus a bit ugly, but it is acceptable for the data structures course. For a stack of integers, we can push an `int` 3 onto a stack `S`, and extract it into the `int Val` with:

```
Stack S = new Stack( );
S.Push( new Integer( 3 ) );
Val = ( (Integer)S.top() ).intValue();
```

For a `SearchTree`, we store objects that are of type `Comparable`, which is an interface that we provide. The interface specifies a set of abstract methods. A class implements the interface if it provides implementations for all of these functions. This alternative to multiple inheritance avoids the problem of inheriting multiple implementations. Only objects that implement the `Comparable` interface (that is, provide some comparison routines) can be inserted into the `SearchTree`. This is an improvement over C++, where the requirements of the template parameter can only be given by comments, and Ada, in which the instantiation types and the operations required by the types are provided separately in the generic instantiation.

Java Annoyances

Java is by no means perfect. If you think all of C++'s problems are gone, think again. A stray semicolon after a while clause will still get an infinite loop, and the switch statement with the default step-though to the next case remains. Class methods can declare local objects with the same name as class data members, yielding hard-to-find shadowing bugs. Java applets have severe security restrictions and cannot do much. Java applications run slowly and may be too slow for large data sets.

Text files in Java are harder to work with than in C++, as is parsing input lines. It is no simple task to write a program that reads two integers and echoes them back. However, one can always write a class to do the dirty work of I/O.

Exceptions in Java work in a similar way as C++, but they are much improved in some details, such as the throw list. The fact that exceptions must be caught or propagated can be annoying, as in the following code that empties a stack:

```
while( !S.isEmpty( ) )
    S.pop( );
```

This code does not compile, if, as would be common, the `pop` routine is written to throw an exception when the stack is empty. In this case, the `pop` must be enclosed in a `try` block with a subsequent `catch` clause. This is good style for large industrial type programs, but very annoying for students, especially for smaller programs.

The GUIs in Java do not work precisely as advertised. Students had a difficult time positioning things exactly as they wanted; they typically settled for what the system would give them. The `repaint` method (when not using threads) does not always get called immediately. Students found that some code that was placed after the `repaint` was being called prior to the actual repainting, leading to lots of frustration. The fonts didn't always fit into the text fields as advertised. This seemed to vary depending on what platform the applet was viewed on.

In C++ and Ada, there is occasion to pass a pointer by reference (meaning that where it points to can be changed). An example is the C++ or Ada code to insert into a tree rooted at `T`. The prototypes are:

```
void      insert( TreeNode * & T )
procedure insert( T: in out PtrNode )
```

There is no equivalent for this in Java, requiring the C-like

```
TreeNode  insert( TreeNode T )
```

In Java, to test if two objects have identical contents, you must use the `equals` method. Thus the typical error for strings is to use == instead of `equals`. When defining new classes, the programmer must provide a definition of `equals` to override the one provided in `Object`. The `equals` in `Object` always returns false. Unfortunately, it is hard to remember that the signature must be

```
boolean equals( Object OtherObject );
```

Typically, a student will write for a new class `MyClass`

```
boolean equals( MyClass OtherObject );
```

which leaves the needed `equals` function in tact (with a return that is always false).

Debugging Java programs was typically difficult. The program always seemed to crash with a `NullPointerException`, suggesting that not all pointer problems have disappeared with Java. Applets that worked in the provided appletviewer failed under Netscape, and vice-versa. Applets that worked for an early version of Netscape failed for later versions.

Student Evaluations

Prior to the beginning of the semester, rumors of Java in the course had spread and most in the class knew what was up. Approximately double the enrollment listed in the class appeared for the first two weeks, just to hear about Java. As soon as the class returned to Data Structures, the extras left.

The final class reaction, based on written student evaluations, was extremely positive. Not surprisingly, large numbers of students were excited to be learning cutting-edge technology. Although many students indicated that they worked harder in this course than any other, most seemed eager to do so, because they felt they were learning a marketable skill. Students had a high opinion of Java, and uniformly recommended that it be placed in the curriculum.

However, although many of the programming assignments were very impressive, performance on in-class exams, which is essentially language independent, appeared to be identical to previous semesters. Additionally, the drop-out rate was essentially unchanged from previous semesters. In fact, our experience suggests no significant differences in basic CS knowledge compared to prior versions of the course, although one could argue that since the students had the additional burden of learning a new language, had they already known Java, they would have done better in the exams.

Future Plans

Prior to the summer, the School had voted to change the curriculum. Essentially, we found that we wanted to teach more and more topics but were bound by a new state rule that insisted that we not require additional course work. As a result, we reluctantly dropped Ada from the CS-1.5 course, and elected to start with C++.

Our CS-1.5 course will start with C++ but will avoid advanced C++ features such as inheritance. It will cover templates and exceptions and will incorporate Visual C++ and GUI programming. The Advanced Programming course will discuss advanced C++ features and object-oriented concepts, and, given recent developments, will cover Java.

Although the Java experience was positive, we currently have no plans to move the first course to Java. Having taught C++ for a while, we have a set of teachers who now know it well enough to avoid many of the common past mistakes. Furthermore, we believe that C++ is not going to go away and will have to be learned eventually; we find that playing with pointers really is important; and we think that the C++ compilers are much less fragile and offer a better development environment than they used to. We feel that a safe C++ is roughly equivalent to Java. Furthermore, we feel that both faculty and students need some measure of stability, and that it is unwise to change the core of the curriculum every time a new fad comes along.

During our transition period, which will extend for a while, we will repeat the Java experiment. Eventually, Data Structures will be taught in C++, though students who know Java will have the option of using it. We expect that many students will elect to take Advanced Programming prior to Data Structures (as they have been), and then use Java for the Data Structures course (in which case we'll be back to the two-language problem, but with two languages that are somewhat closer to each other).

Conclusions

All in all, Java is an excellent language for Data Structures. So is C++ and so is Ada. Java has the advantage that its core features can be quickly taught to students with C++ knowledge, it has a simple GUI interface, students can avoid thinking about some of the details required of C++ programmers, and it is the hot language this year. It looks like a keeper.

Additionally, we feel that universities that have moved to C++ in CS-1 will stay with that choice, but those who are still using Pascal due to fears of C++ will move to Java.

More information on this course is available at the previously mentioned web site. Class lectures, sample Java code for many data structures, as well as the class assignments (and some submissions) can be found there.

References

1. Cornell, G. and Horstmann, C., *Core Java*. Prentice-Hall, NJ., 1996.
2. Deitel, H. and Deitel, P., *Java: How To Program*, Prentice-Hall, NJ., 1997.
3. Flanagan, D., *Java in a Nutshell*. O'Reilly & Associates, CA., 1996.
4. Weiss, M. A., *Data Structures and Algorithm Analysis in Ada*. Addison-Wesley, Mass., 1993. (also in C++, 1994).