

An Overview of MiniJava

Eric Roberts
Stanford University
eroberts@cs.stanford.edu

ABSTRACT

This paper describes the implementation of MiniJava, a teaching-oriented programming language closely based on the Java language developed by Sun Microsystems [6]. The core of the MiniJava environment is a restricted subset of the standard Java release and is designed to reduce the intimidation factor introductory students experience when they encounter a system as large as the Java environment. The paper outlines the particular restrictions and extensions that define MiniJava along with pedagogical justifications for each.

1. INTRODUCTION

Since its release in 1995, Java has grown significantly in popularity as a language for teaching programming. It is used extensively in intermediate programming courses at the CS2 level and is proving to be increasingly popular as an introductory language for CS1. An ad-hoc committee to study the Advanced Placement exam in computer science has recommended strongly that Java replace C++ in the AP program [2].

At the same time, there continues to be concern about several aspects of Java's design, particularly over its suitability for novice programmers [3, 5, 7, 8, 17]. While Java has much to recommend it as a language for advanced students, the standard implementation—which includes the various libraries in the Java Development Kit provided by Sun—forces introductory students to assimilate too much information too fast, making it difficult for many students to understand the critical conceptual issues involved in programming and algorithmic design.

The level of concern about Java within the SIGCSE community was illustrated in early 2000, when a short message to the SIGCSE-MEMBERS mailing list [13] generated a flood of responses, most of which advocated the creation of a teaching library for Java that would circumvent the problems that instructors have found using the language. Although the idea of such a library has surfaced from time to time in the education community, Sun has shown little interest in developing a simple set of standard packages explicitly designed for pedagogical use.

This paper describes an alternative approach to the concerns raised by Java's design. Instead of having students begin their computer science education with an industrial-

strength implementation of Java, students can be introduced to programming using a more restrictive environment that is closely linked to Java, but which avoids both the complexity of the full-scale version and the specific shortcomings identified by the computer science education community. By using this simplified implementation—which is called MiniJava here—students will be able to focus on the important conceptual issues of programming without getting bogged down in details.

The MiniJava system was originally designed as part of the software library for a forthcoming text intended for use in the nonmajors course, where students have even less tolerance for complexity than those who have committed themselves to computer science as a field of study. Because there has long been significant prejudice among educators against tools that are not part of an industry-supported standard, my initial assumption was that MiniJava would have no place in a traditional CS1 environment. After giving several informal demonstrations and talks about the MiniJava environment, however, I was surprised at the level of interest—not only from those who teach the nonmajors course—but also among faculty responsible for the introductory sequence. Based on that experience, I concluded that it was worth presenting the MiniJava design, both to get a better sense of the level of interest and as a way of soliciting feedback about the design choices.

2. WEAKNESSES IN JAVA

Java has several shortcomings that limit its effectiveness at the introductory level. In general, the problem arises from the fact that Java forces students to understand a number of conceptually sophisticated concepts in order to write even the simplest programs. As a result, students who are new to programming must sift through a mass of complicating details before they can understand the underlying ideas. While the best students can overcome this hurdle, others will become frustrated and abandon programming without giving it a fair chance.

The most significant sources of complexity in Java for new programmers include the following:

- *The size of the language and its libraries.* Although the Java language itself is relatively simple, novices do not understand the boundary between the language and its associated class libraries. In Java, those libraries are huge. New programmers are uncomfortable with the sheer number of packages, classes, and methods, making it hard for them to learn programming using systems of this scale.
- *Continued instability of the Java platform.* In the five years since its initial release, Java has undergone nearly constant upheaval. The 1.0, 1.1, and 1.2 releases are all quite different—involving considerably more significant changes than, for example, in the evolution from K&R to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to servers requires prior specific permission and/or a fee.

SIGCSE 2001 2/01 Charlotte, NC, USA

© 2001 ACM ISBN 1-58113-329-4/01/0002...\$5.00

ANSI C. The 1.1 release in particular included several significant changes to the language itself, along with a completely redesigned event model and a wide variety of additional packages and classes. Such changes have a profound effect on education. The rapid changes make it difficult to develop reusable materials or to write effective textbooks; many authors who were writing Java books at the time of the 1.1 upgrade were forced to pull their books out of production. Ultimately, such instability forces instructors to compromise on the quality of instructional materials and ends up hurting students who seek to learn programming in the Java domain.

- *Separation of interface and implementation.* One of the central ideas behind object-oriented programming—and indeed most modern approaches to software development—is that of encapsulation, which emphasizes the separation of the programming interface for a particular operation and its underlying implementation. Unlike C++ and even many of the prevailing disciplines for C, Java does not support this separation and includes the implementation details as part of the public representation of a class. This problem has been noted by other authors [3, 8] and represents a major pedagogical weakness in the Java design.

- *Standard input.* Although output tends to be quite simple using the Java I/O classes, input operations are much more difficult for students to understand. Reading an integer from the standard input stream, for example, requires a considerable amount of code. In the libraries associated with *Core Java* by Cornell and Horstmann [4], the method to read an integer from consists of 30 lines of code involving such complexities as **while** loops, boolean variables, **String** operations such as concatenation and **trim**, the **Integer** wrapper class, and exceptions from two different packages. Beginning students cannot understand how to write this type of code on their own, leading many Java authors in the past few years [4, 10, 12, 18] to define their own classes for keyboard input. Because no standard exists for these classes, these extensions have met with resistance in the CS education community.

- *Exception handling.* Outside of a few common runtime exceptions, Java requires every method to catch the exceptions generated in its body or to document the possibility of those exceptions in a **throws** clause. As a result, Java programmers are forced to use exceptions even for very simple programs. Although this problem is most severe in the I/O classes, it comes up in other contexts as well. Delaying the current thread for a specified interval, for example, requires the following code:

```
try {
    Thread.sleep(time);
} catch (InterruptedException ex) {
    This exception is typically ignored.
}
```

Some authors have gone to extreme lengths to avoid the exception problem. In their introductory text, Arnow and Weiss [1] teach students to write all methods with a **throws Exception** clause. This subterfuge constrains the effect of exceptions to a few noise words in each method header at the cost of encouraging poor programming style.

- *Wrapper classes.* In Java, values of primitive types like **int** and **double** are not objects and cannot be manipulated as objects are. To enable the use of primitive types where objects are required, the **java.lang** package includes a wrapper class corresponding to each of the primitive type. The distinction between the primitive types and their associated wrapper classes is very hard for students to understand. Using these wrapper classes also adds complexity to the code. For example, to store a value of type **int** in a table, the students must first allocate a new **Integer** with the desired value, store that value in the table, and then reconstruct the **int** by calling **intValue** when the corresponding lookup operation is performed.

- *The graphics model.* As I argue in a paper presented at the 1998 ITiCSE conference [17], the Java graphics model is not well suited to introductory teaching for three reasons: the use of “forgetful bitmaps” that require clients to repaint the window, statelessness of the Java graphics system, and the use of an integral, pixel-based coordinate system.

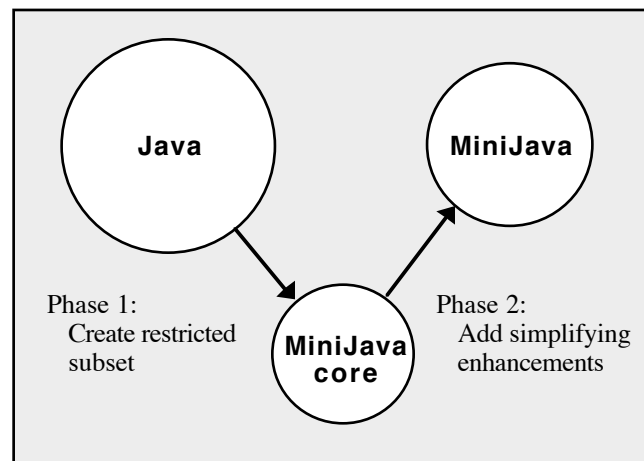
3. MINIJAVA VS. JAVA

Although MiniJava is designed as a simpler version of Java, it is not a strict subset. The relationship between Java and MiniJava is illustrated in Figure 1, which reflects two distinct sets of changes. Starting with the complete Java language represented by the large circle in the upper left, the first phase in the definition of MiniJava consists of introducing several restrictions that lead to a much simpler subset called the MiniJava core. The second phase consists of adding simplifying enhancements to the MiniJava core, which are collectively designed to make it easier for novices to use. In general, students begin their study of MiniJava with these simplifying enhancements in place. As they gain greater mastery of programming concepts, those students learn how those simplifying enhancements work so that they can operate without them.

4. RESTRICTIONS IN MINIJAVA

The subsections that follow outline the restrictions that reduce Java to the subset that forms the core of MiniJava. For each of the changes, I have also included the pedagogical rationale for imposing that restriction.

Figure 1. Relationship between Java and MiniJava



4.1 Simplification of the class libraries

The most significant problem for new users is the sheer size of the library packages that come with the standard Java release. In Java 1.1, the `java` branch of the package hierarchy contains 25 packages with 705 public classes and over 3200 distinct public methods. The Java 1.2 release, which incorporates entire new class hierarchies such as the Swing interface toolkit, is larger still. No one—and particularly no one just learning about programming—can understand it all.

Of course, understanding it all is hardly essential to writing programs. Students who are solving simple programming problems won't need to use more than a handful of this vast panoply of classes and methods. To a large extent, programming today—even for experts—is characterized by this process of finding the structures you need in a massive collection of available tools. Working with large libraries and learning to ignore those parts that aren't needed for the application at hand are essential programming skills that students must master at some point in their study of programming. For students just learning about programming, however, it is often difficult to adopt this approach. They are, of course, not sure what they need, and the array of possibilities listed in the index is overwhelming. Even students whose mode of discovery consists of trying things out—the approach Sherry Turkle identifies as *bricolage* [19]—run into trouble simply because there are too many possibilities to try. Size does matter. If one must learn to find a needle in a haystack, it helps to begin with an extremely small haystack.

Many computer science educators have recognized the advantages of starting out with a simple system. The success of Richard Pattis's *Karel the Robot* system [12] demonstrates the value in introducing the essential concepts of programming without overwhelming the students with details. *Karel* works because it's simple. Introductory students can easily learn it all in a very short time.

MiniJava adopts a similar approach. One of the design goals of the system is to ensure that students can learn everything about it in a single introductory course. To make such a simplification possible, it is essential to reduce the number of built-in classes from the 700+ in Java 1.1 to something more manageable. MiniJava has 17 classes in its standard core, although the MiniJava framework makes it easy to add new classes that extend the core functionality.

4.2 Elimination of inner classes

Inner classes—a term that actually encompasses a variety of distinct structures including *member classes*, *local classes*, and *anonymous classes*—were incorporated into the Java language as part of the Java 1.1 release. In many cases, inner classes are extremely convenient and eliminate the need to define top-level classes. On the other hand, inner classes add no fundamental power to the language and represent a significant complication for new students.

4.3 Restrictions on control statements

The current implementation of MiniJava eliminates the following control statement forms:

- *The **do-while** statement.* The **do-while** statement adds no power to the language and means that students must

choose between two distinct statement forms—**while** and **do-while**—for loops terminated by some exit condition. Typically, students who choose to use the **do-while** construct do so in applications in which it doesn't apply. More often than not, the resulting programs are buggy, since they fail in those cases in which the loop should not be executed at all.

- *Multilevel **break** statements.* In my 1995 SIGCSE paper on “Loop Exits and Structured Programming” [16], I argue that single-level **break** statements must be made available to students because of extensive evidence that prohibiting its use often leads students—and textbook authors—to write incorrect code. The multilevel **break** statement provided in Java is not critical in the same way. When the code has become complex enough to require nested loops, it usually makes more sense to achieve the effect of a multilevel **break** by returning from the current method or by throwing an exception.

- *The **continue** statement.* Unlike the **break** statement, which is required to solve the loop-and-a-half problem in a way that students can easily comprehend, the **continue** statement adds nothing to the language and simply encourages poor programming style. Leaving it out of MiniJava ensures that students will learn to code algorithms without this crutch.

- *The fall-through option in **switch** statements.* Novice programmers often forget to include a **break** statement at the end of each **case** clause. In the absence of the **break** statement, execution continues into the next branch. This fall-through semantics has long been recognized—even by C's designer Dennis Ritchie—to be a flaw in the language, but the problem of maintaining legacy code that uses this feature makes it impossible to correct the design. In MiniJava, which has no legacy-code problem, falling through from one nonempty case clause to the next is treated as a runtime error, so that students who forget to include **break** statements will at least have their failure caught.

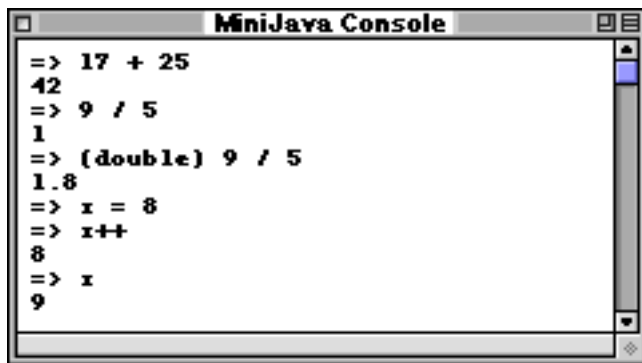
5. SIMPLIFYING EXTENSIONS IN MINIJAVA

The following subsections describe several of the more interesting extensions that MiniJava adds on top of the core. These extensions are included to make MiniJava easier to teach to novices, specifically targeting the areas of greatest concern outlined in section 2. These extensions can be selectively disabled in the interpreter if one wants to force students to learn the standard Java form. Typically, the best pedagogical strategy is to introduce the feature in the simplified form, returning to the topic at a later point to discuss how one would go about implementing that construct in Java without using the MiniJava extensions.

5.1 The read-eval-print loop

Introductory textbooks and courses tend to use Java in one of two modes. Some have the students create applets that can be viewed using a browser, while others focus on developing complete application programs. In either case, students must work with a fair amount of machinery just to get the simplest programs running. In the applet world, students must write their programs as subclasses of the class `java.applet.Applet` and define methods like `paint`

Figure 2. The MiniJava read-eval-print loop



to implement their own operations. To run the applet, the student must also create an HTML file to establish the link between the browser and the applet code. In the application world, students must construct a class containing a method with the signature

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

exposing them immediately to the concepts of visibility, static methods, return types, strings, and arrays.

MiniJava takes a different starting point. When the MiniJava interpreter starts up, it creates a console window, which—in a manner reminiscent of Lisp and similar purely interpretive languages—reads expressions from the user, evaluates those expressions, and then prints the results. Figure 2 shows the MiniJava read-eval-print loop in operation. In this example, the output illustrates very quickly such important concepts as the distinction between integer and real division, the use of type casts, and the behavior of the ++ operator. Being able to evaluate simple expressions on the fly is a wonderful tool for students who have trouble understanding how the underlying concepts work. The console window and its read-eval-print loop support experimentation with the language features, which in turn helps many students reinforce their understanding of the underlying concepts.

Beyond the advantage of allowing users to experiment with expression evaluation, the existence of the read-eval-print loop makes it much easier to invoke program operations. From the read-eval-print loop, the student can instantiate new class instances or invoke methods, which can either be public methods in the objects that have already been created or static methods from any defined class. In MiniJava, programs no longer need to be distinct from other kinds of code. To get something running, the student simply loads the appropriate class definitions and then invokes a method with the appropriate parameters. Particularly when combined with the extensions described in the next few sections, the read-eval-print loop simplifies enormously the problem of getting a program started, leaving students with more time to focus on the actual task.

5.2 The Console class

When Java is used in CS1, the problem that generates the greatest concern is the lack of any simple mechanism for performing console I/O. Most Java textbooks that have appeared in the last few years define a class for this purpose

and use that class in the introductory chapters of the text. This strategy has not as yet converged on any standard, but there is widespread feeling that some such facility is necessary.

In MiniJava, the core set of built-in classes includes a **Console** class that gives the programmer access to the MiniJava console running the read-eval-print loop. The **Console** class includes the following static methods:

- **print** and **println**, which are identical to their Java counterparts in **PrintStream**
- **readInt**, **readDouble**, and **readLine**, which read values of type **int**, **double**, and **String**, respectively

When given illegal input values, the input methods do not raise exceptions but instead offer the user a chance to reenter the data.

5.3 The Program and GraphicsProgram classes

The standard approach to writing programs in MiniJava is to define a new class that extends either the **Program** or **GraphicsProgram** classes, both of which are built into the MiniJava core. The **Program** class makes it easy for students to write simple programs that are similar to those used in a more traditional introduction to programming in the context of a procedural languages. The **Program** class creates a console window and exports its own versions of the **Console** class methods so that beginning programmers do not have to learn immediately about qualified names.

An example of a simple program definition in MiniJava appears in Figure 3. The MiniJava interpreter includes a **Run** button that automatically instantiates an object of the most recently loaded **Program** subclass and then invokes its **main** method, making it very easy for students to write and execute their code.

Students using MiniJava, however, are not limited to traditional paradigms of console-based interaction. By subclassing the **GraphicsProgram** class, students can use an interactive graphical environment that supports two distinct graphical models: a model similar to that provided by the **java.awt.Graphics** class, which is essentially procedural in its operation, and an object-oriented package based on the collage model described at ITiCSE '98 [17].

Figure 3. A MiniJava program to add two numbers

```
// File: add2.mj
// This program adds two numbers.

class Add2Program extends Program
{
    void main()
    {
        int n1, n2;

        println("This program adds two numbers");
        print("Enter 1st number? ");
        n1 = readInt();
        print("Enter 2nd number? ");
        n2 = readInt();
        println("The sum is " + (n1 + n2));
    }
}
```

5.4 Primitive types as objects

The most fundamental change in the structure of MiniJava lies in the treatment of the primitive types. In Java, these types—**byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**—are not objects and cannot be used where objects are required. For example, it is illegal to store a value of type **int** as an element of a **Vector** or as a value in a **Hashtable**. To enable programmers to use these primitive types in those contexts, the **java.lang** package includes wrapper classes—**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, and **Boolean**—that correspond to each of the primitive types. For example, if you want to add the value of the integer **n** to the vector **v**, you cannot write

```
v.addElement(n);
```

because **n** is not an object. To achieve the same effect in Java, you need to encapsulate the value of **n** in an **Integer** object, like this:

```
v.addElement(new Integer(n));
```

Conversely, when you want to extract the value from the **Vector**, you must use methods from the **Integer** class to retrieve the value as an **int**, as illustrated by the following expression, which extracts the value of the element at index position **i**:

```
((Integer) v.elementAt(i)).intValue()
```

For novices, all this extra baggage just gets in the way.

In MiniJava, all values—even the primitive types—are objects. MiniJava retains both the primitive type names and the wrapper class names from Java, but makes the two synonymous to reduce the level of confusion when students move on to Java.

Because MiniJava treats all values as objects, no conversion to a wrapper class is ever required. If you want to add the integer **n** to the vector **v**, all you need to write is

```
v.addElement(n);
```

5.5 Other simplifying extensions

In addition to the major changes described in the preceding subsections, there are a number of smaller extensions that address specific shortcomings. These extensions include:

- The **throws** clause has been made optional, eliminating the need to introduce exceptions early.
- The MiniJava editor includes a view option that presents the signature and comments for a method while hiding its implementation.
- Syntactic support has been added to the **Vector** class, making it possible to use vectors in place of arrays, which reduces the confusion students have when they encounter two separate mechanisms for accomplishing the same goal.
- A **foreach** statement has been added to streamline the coding of iterator patterns.
- Many type casts—specifically casts that narrow a more general value to a more specific type—are optional, leading to programs that require far fewer type casts.

Each of these extensions can be individually disabled using an options panel, which allows each instructor to limit the extent to which MiniJava deviates from the Java standard.

6. Conclusion

The current implementation of MiniJava includes the features described here and is sufficiently robust that it serves as the platform for the animated demo programs I've used in talks about the system. MiniJava, however, is still under development and is likely to remain so even after its associated textbook is published in late 2001. I welcome both external testers and any suggestions to make the system more useful in an educational setting.

REFERENCES

- [1] David Arnow and Gerald Weiss. *Java: An Object-Oriented Approach*. Reading: Addison-Wesley, 1998.
- [2] Owen Astrachan, et al. Recommendations for changes in Advanced Placement Computer Science. Reading, MA: Addison-Wesley, 1998. *SIGCSE Bulletin*, March 2000.
- [3] Robert Biddle and Ewan Tempero. Learning Java: Promises and pitfalls. *Proceedings of the 14th Uniform NZ Conference*, 1997.
- [4] Gary Cornell and Cay Horstmann. *Core Java*. SunSoft Press, Mountain View, CA, 1996.
- [5] Ann Fleury. Student conceptions of object-oriented programming in Java. *Journal of Computing in Small Colleges*, 1999.
- [6] James Gosling and Harry McGilton. The Java language environment: A white paper. Sun Microsystems, 1996. <http://java.sun.com/docs/white/langenv/>.
- [7] Jason Hong. The use of Java as an introductory programming language. *ACM Crossroads*, Summer 1998, <http://www.acm.org/crossroads/xrds4-4/introjjava.html>.
- [8] Frederick Hosch. Java as a first language. *SIGCSE Bulletin*, September 1996.
- [9] Michael Kölling and John Rosenberg. Objects first with Java and BlueJ. *SIGCSE Bulletin*, March 2000.
- [10] Elliot Koffman and Ursula Wolz. *Problem Solving with Java*. Reading, MA: Addison-Wesley, 1999.
- [11] Michael Kölling. The BlueJ Tutorial—Version 1.0, <http://bluej.monash.edu/tutorial/tutorial.pdf>.
- [12] John Lewis and William Loftus. *Java Software Solutions: Foundations of Program Design*. Reading, MA: Addison-Wesley, 2000.
- [13] Nick Parlante. Simplified Java I/O for CS1? Posting to the SIGCSE-MEMBERS list, March 3, 2000.
- [14] Richard Pattis. *Karel the Robot*. New York: Wiley, 1981.
- [15] Stuart Reges. Conservatively radical Java in CS1. *SIGCSE Bulletin*, March 2000.
- [16] Eric Roberts. Loop exits and structured programming. *SIGCSE Bulletin*, March 1995.
- [17] Eric Roberts and Antoine Picard. Designing a Java graphics library for CS1. *SIGCSE Bulletin*, September 1998.
- [18] Walter Savitch. *Java: An Introduction to Computer Science and Programming*. Upper Saddle River, NJ: Prentice-Hall, Addison-Wesley, 1999.
- [19] Sherry Turkle. *The Second Self : Computers and the Human Spirit*. New York: Touchstone, 1984.