Adapting the Tracing Method to Java

TOM M. WARMS
Pennsylvania State University Abington College

KAVON FARVARDIN
Pennsylvania State University

TOM M. WARMS

Tom M. Warms is a faculty member in the department of computer science and engineering at Penn State Abington. He received S.B. and M.S. degrees, both in mathematics, from MIT and NYU, respectively. He received his Ph.D. in formal linguistics and mathematical logic from the University of Pennsylvania in 1988. He has published papers in pattern recognition, psycholinguistics and computer science pedagogy. Dr. Warms may be reached at t1w@psu.edu.

KAVON FARVARDIN

Kavon Farvardin is an undergraduate student majoring in computer science at the Pennsylvania State University. He may be reached at kff5027@psu.edu

Adapting the Tracing Method to Java

Tom M. Warms
Pennsylvania State University Abington College
Kavon Farvardin
Pennsylvania State University

Abstract - The tracing method and its software implementation RandomLinearizer are proving to be effective tools in the teaching of C++. This paper discusses the issues involved in adapting the method to the Java programming language, and presents several typical Java programs that are presented to beginning students, and the corresponding traces. It then speculates on the usefulness of tracing to students of Java.

Introduction

The tracing method provides a set of notations in which to represent the execution of programs in a limited subset of a programming language. It is a method[1] by which an instructor can demonstrate some of the features and algorithms of a programming language, and students can demonstrate their understanding. It is a useful tool for beginning students because it helps them learn elementary techniques such as decision structures and looping; it is also useful to more advanced students because of its capability for helping students learn more advanced techniques such as pointers and linked lists. It supplements the kind of verbal and pictorial descriptions of the execution of elementary programs that one finds in textbooks.

The method has been shown[2] to clarify the execution of some programs and explain why certain techniques that may seem to work will not. A software program named RandomLinearizer[3] was created in support of the tracing method; an experiment showed[4] that students enjoyed using the method and software and found it useful. The method and software were written for C++; the current paper is a preliminary evaluation of the method's suitability for use with the Java language; it speculates on the utility of tracing to students who are learning Java as their first programming language, and to students for whom the first language was C++ and who learn Java in subsequent courses.

Tracing

In the method, names of identifiers are placed on the left side of a vertical line and the identifiers' values on the right. The name of the function being executed appears above the vertical line. Boxes indicate output, underlines indicate input, and ↵ represents the RETURN character. Values returned by functions are enclosed in circles. Figure 1 shows a simple C++ program and its trace, assuming an input value of 23. When tracing a statement in a program, the student has available the result of tracing the previous statements in the program.

```
// This program doubles an input value.
#include <iostream>
using namespace std;

int main()
{
      int number, doubleNum;

      cout << "Enter an integer ->";
      cin >> number;

      doubleNum = 2 * number;

      cout << "The number = " << number << endl;
      cout << "Twice the number = " << doubleNum << endl;

      return 0;
}
```

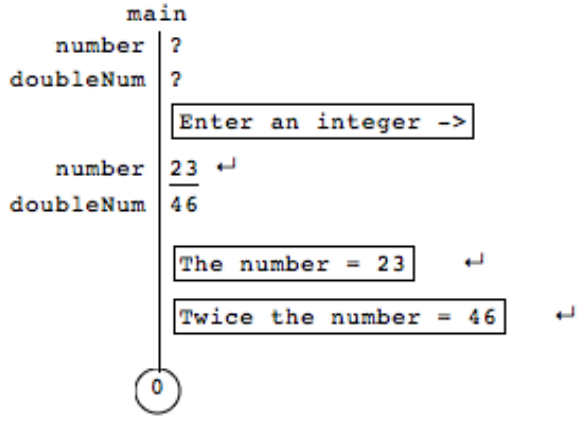


Figure 1

Elements of the trace of a program—the underlined material, material in boxes, and return characters—can be used to predict the contents of the console screen. The trace in Figure 1 suggests the following contents:

```
Enter an integer ->23
The number = 23
Twice the number = 46
```

A C++ program that uses a user-defined function to do the same calculation, and its trace, are shown in Figure 2. When control is transferred to the function calculateAnswer, the trace moves to the right, and the values of the formal parameters are indicated. Then the trace shows the execution of statements of the function block, the value returned by the function is encircled. The trace then moves back to the left.

```
// This program uses a user-defined function to calculate
// twice the input value
#include <iostream>
using namespace std;
int calculateAnswer(int num);

int main()
{
      int number;

      cout << "Enter an integer ->";
      cin >> number;

      int doubleNum = calculateAnswer(number);

      cout << "The number = " << number
                  << endl;
      cout << "Twice the number = "
                  << doubleNum << endl;

      return 0;
}
int calculateAnswer(int num)
{
      int answer = 2 * num;
      return answer;
}
```

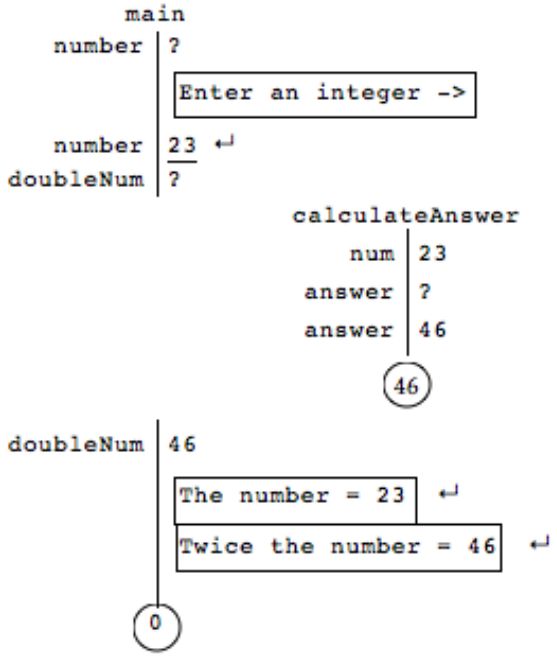


Figure 2

Java is similar in many ways to C++. An excellent exposition of similarities and differences is provided by Eckels[5]. Elementary C++ programs such as those of Figures 1 and 2 carry over without major change. In Java, however, a class must be created even to accomplish simple tasks. Figure 3 contains a Java program that is equivalent to the C++ program of Figure 2, along with its trace.
The header of the main program in Figure 3

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

is traced by

```
        main
args | null
```

The identifier args is a String array, initially null. Although console output is convenient to use in Java, as in the statement in Figure 3

```
System.out.println("The number = " + number);
```

console input is not. System.in on its own only allows for fetching bytes from the standard input; Scanner provides a wrapper for this object to make it usable by beginners. The statement

```
Scanner console = new Scanner(System.in);
```

contains a call to a constructor of the Scanner class, and thus creates an object of this class. Memory is allocated by new, and its machine address is returned as a reference and assigned to console. A complete trace sequence would indicate that System.in is the parameter and would provide details of the constructor's execution, but perhaps be confusing in its complexity. The solution arrived at is to abbreviate the sequence to

```
console | null
console | ADDR0
```

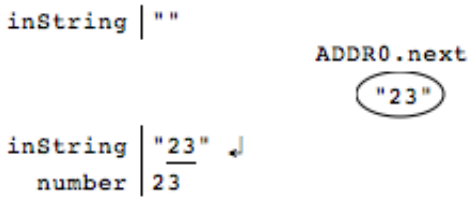
to indicate that console's initial value is null, and then it is assigned the object reference. The student can consider this to be an idiomatic trace sequence. The statement

```
String inString = console.next();
```

assigns to inString the next string that is entered at the console. It is then converted to int and the result is assigned to number:

```
number = Integer.parseInt(inString);
```

The sequence



traces this segment. The quote marks indicate that the value of inString is indeed a String value, while the underline indicates that 23 is visible on the console.

```
class Main{
      public static void main(String [] args){
            int number;
            Scanner console = new Scanner(System.in);

            System.out.println("Enter an integer ->");
            String inString = console.next();
            number = Integer.parseInt(inString);

            int doubleNum = calculateAnswer(number);

            System.out.println("The number = " + number);
            System.out.println("Twice the number = " + doubleNum);
      }

      public int calculateAnswer(int num)
      {
            int answer = 2 * num;
            return answer;
      }
}
```

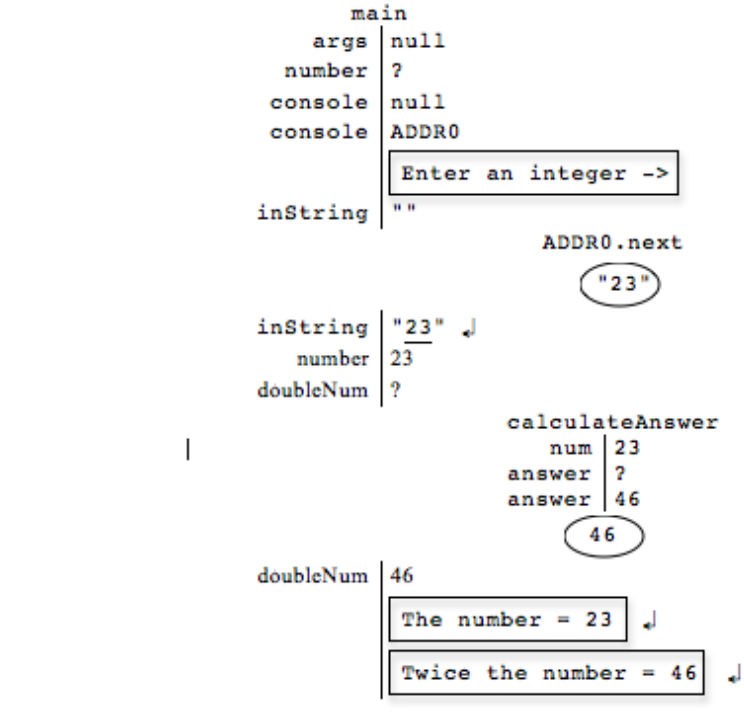


Figure 3

Java's graphic user interface provides methods of the JOptionPane class to prompt for input and report output--for example, JOptionPane.showInputDialog provides a value for inString in the same way as a call to a Scanner method, and JoptionPane.showMessageDialog prints output.

Pass by value and pass by reference

In C++, a user-defined function that returns more than one value does so by reference. That is, the function's parameters, called reference parameters, are placeholders for the actual parameters in

the calling program. The program in Figure 4a uses reference parameters in a function that calculates the sum and product of two input values. In the trace of Figure 4b, the arrows between actual parameter sum and formal parameter total, and between actual parameter product and formal parameter indicate the connection.

```
// Program to prompt user for two integers and calculate and print
// their sum and product.  The calculations are performed in a
// user-defined function.
#include <iostream>
using namespace std;

void getAnswers (int num1, int num2, int &total, int &prod);

int main()
{
      int x1, x2, sum, product;

      // Prompt for input
      cout << "Enter two integers ->";
      cin >> x1 >> x2;

      // Call function to do the calculations
      getAnswers(x1, x2, sum, product);

      // Print results
      cout << "The integers are " << x1 << " and " << x2 << endl;
      cout << "Their sum is " << sum << endl;
      cout << "Their product is " << product << endl;

      return 0;
}
void getAnswers (int num1, int num2, int &total, int &prod)
// Function getAnswers calculates the sum and product of
// its first two parameters and places the answers
// in the third and fourth parameters
{
      total = num1 + num2;
      prod = num1 * num2;
}
```

Figure 4a

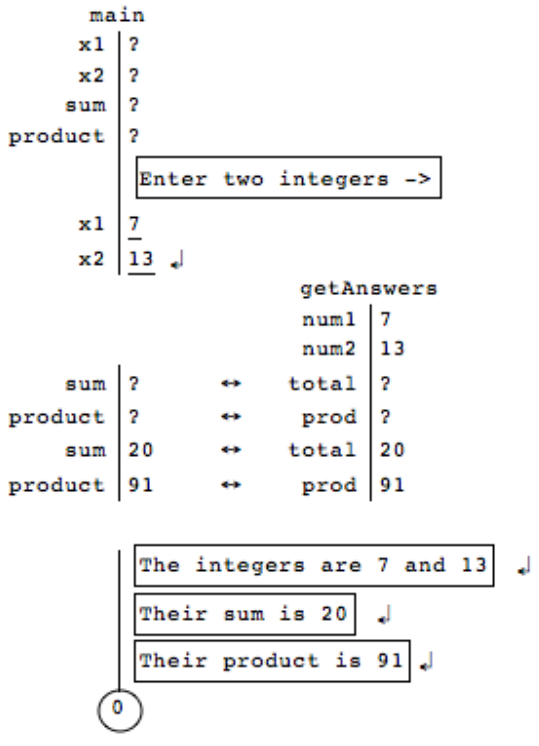


Figure 4b

Returning values from a function in Java to a calling program can be accomplished using accessor methods of a class. For example, the Java program shown in Figure 5 prompts the user for the length and width of a rectangle, and calculates and prints the area and perimeter.

The student must learn to frame problems in terms of classes in order to succeed in Java. Although C++ programs may also be constructed using multiple files, it is hardly possible to write a substantial program in Java without multiple files, as each public class must be contained in a separate file. Any tracing implementation for Java should emphasize the shifting from one file to another during execution of a multi-file program.

```
public class Rectangle{
      private double length, width;
      private double area, perimeter;

      public Rectangle(){
      }

      public void getDimensions(){
            System.out.print("Enter length and width:");
            Scanner myInput = new Scanner(System.in);
            String inString = myInput.next();
            length = Double.parseDouble(inString);
            inString = myInput.next();
            width = Double.parseDouble(inString);
      }

      public void doCalculations(){
            area = length * width;
            perimeter = 2.0 * (length + width);
      }

      double getArea(){
            return area;
      }

      double getPerimeter(){
            return perimeter;
      }
}


public class Main{
      public static void main(String args[]){
            Rectangle r = new Rectangle();
            r.getDimensions();
            r.doCalculations();
            System.out.println("Area = " + r.getArea());
            System.out.println("Perimeter = " + r.getPerimeter());
      }
}
```

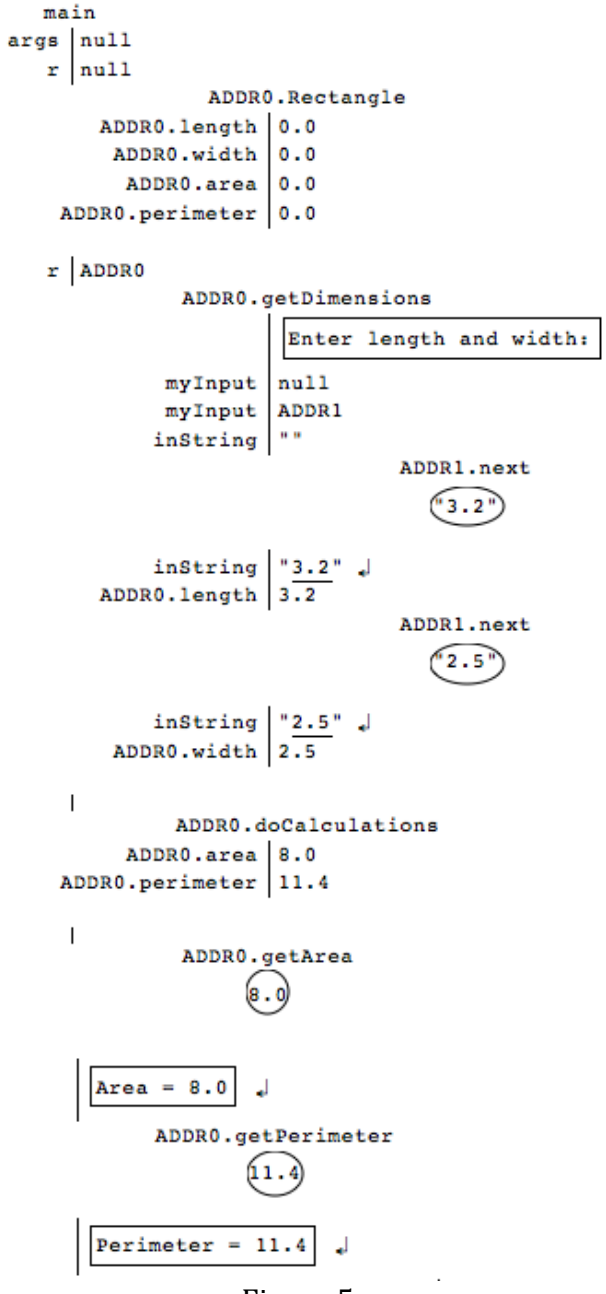


Figure 5

The importance of object references in Java makes it worth examining whether or not a Java program using an array of objects can be traced without adding complexity. The program of Figure 6 creates an array of elements of the Student class, and prompts for a name and grade for each of the elements. It calculates and prints an average and then a list of those students whose grades are above that average.

The array of Students is declared in the statement

```
Student [] sts = new Student[numStudents];
```

The trace of this statement shows that
sts is initially null:

```
sts | null
```

space is allocated for an array of 3 elements of type Student, or more accurately, an array of 3 object references, whose initial values are listed sequentially in the trace:

```
ADDR1 | null null null
```

and finally a reference to that array is assigned to sts:

```
sts | ADDR1
```

Each time the statement

sts[i] = new Student();

is executed, a new Student object is created and a reference to it is assigned to one of the elements of the array sts. Figure 6 shows that after creation of the first Student object, the elements of the array referenced by sts are ADDR2, null, and null.

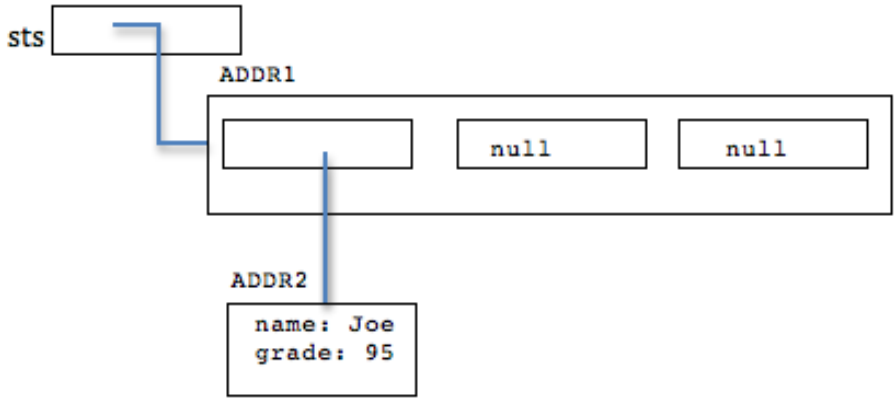
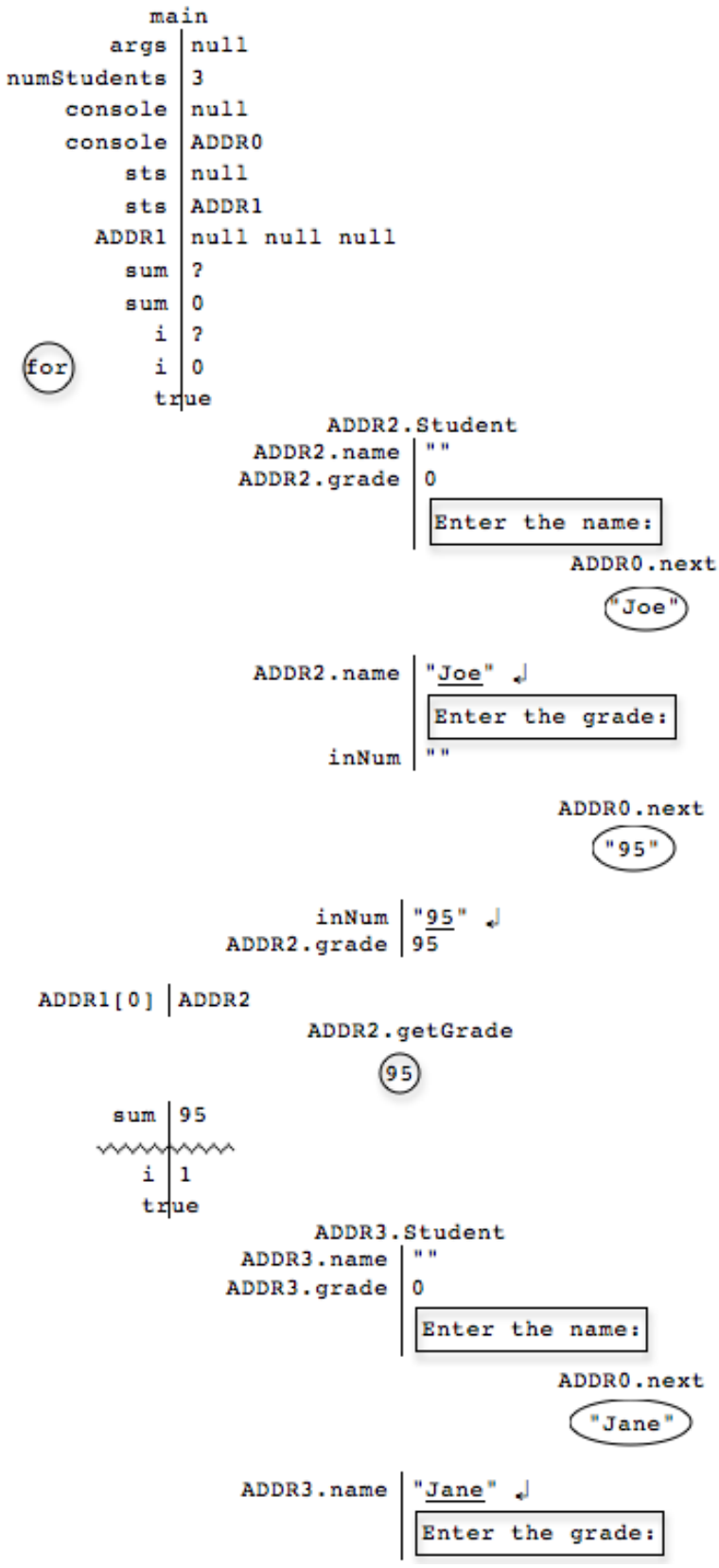
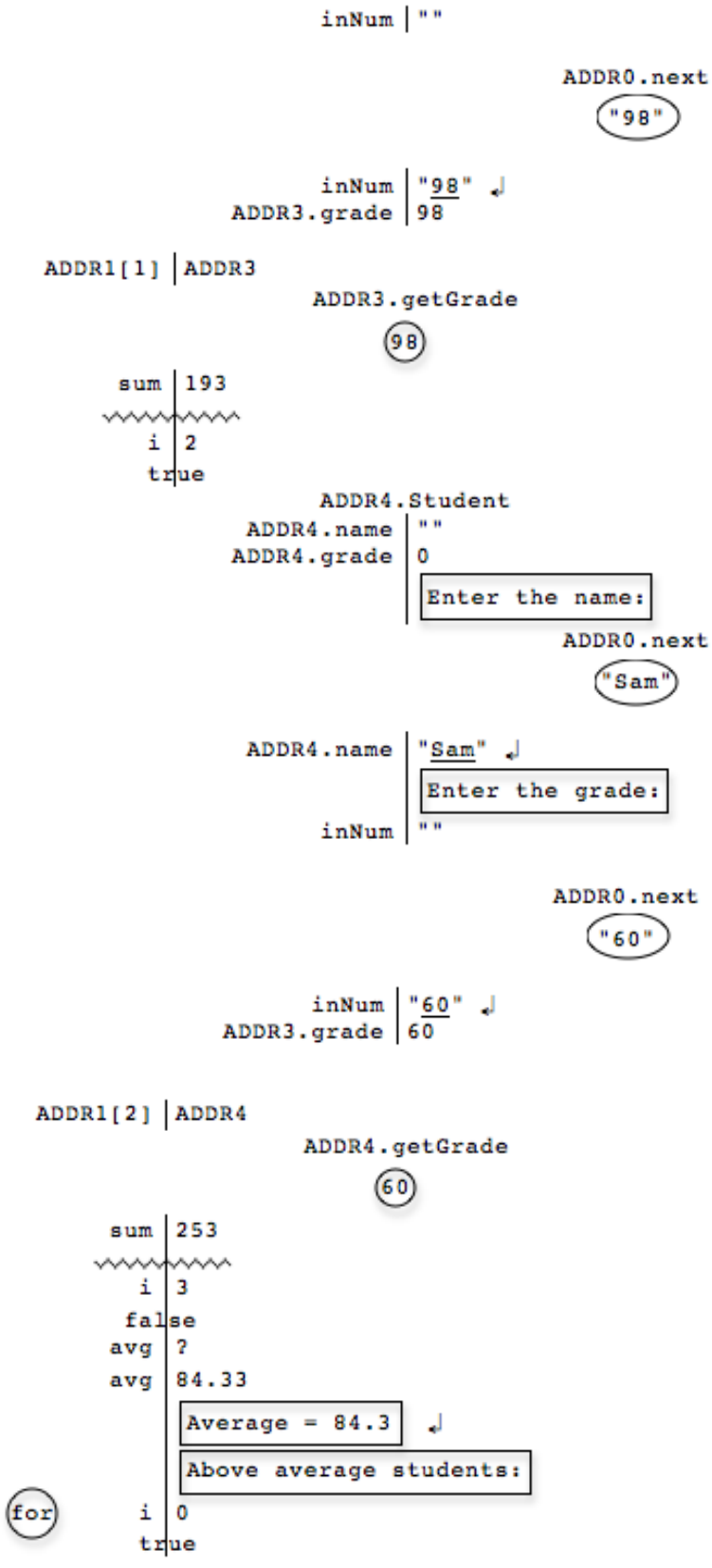


Figure 6

It is interesting that Figure 6 is representative of the kind of figure often found in elementary Java texts. The student can follow the trace of Figure 7 and draw a similar figure.

```
// This program prompts the user for the names and grades of
// three students and prints the names of those above average
public class Student
{
	static Scanner console;

	public static void main(String[] args) {
		final int numStudents = 3;
		console = new Scanner(System.in);
		Student [] sts = new Student[numStudents];
		int sum = 0, i;
		for (i = 0; i < numStudents; i++){
			sts[i] = new Student();
			sum += sts[i].getGrade();
		}
		double avg = (double) sum / numStudents;
		System.out.println("Average = "
			+ Math.round(10.0 * avg) / 10.0);
		System.out.print("Above average students:");
		for (i = 0; i < numStudents; i++)
		     if (sts[i].getGrade() > avg)
				System.out.print(sts[i].getName() + " ");
		System.out.println();
	}

	String name;
	int grade;

	 public Student(){
		System.out.print("Enter the name:");
		name = console.next();
		System.out.print("Enter the grade:");
		String inNum = console.next();
		grade = Integer.parseInt(inNum);
	 }

	 public void display(){
		System.out.println("Name:" + name + ", Grade:" + grade);
	 }

	 public int getGrade(){
		return grade;
	 }

	 public String getName(){
		return name;
	 }
 }
```


main
args | null
numStudents | 3
console | null
console | ADDR0
sts | null
sts | ADDR1
ADDR1 | null null null
sum | ?
sum | 0
i | ?
for
i | 0
true
ADDR2.Student
ADDR2.name | ""
ADDR2.grade | 0
Enter the name:
ADDR0.next
"Joe"
ADDR2.name | "Joe"
Enter the grade:
inNum | ""
ADDR0.next
"95"
inNum | "95"
ADDR2.grade | 95
ADDR1[0] | ADDR2
ADDR2.getGrade
95
sum | 95
i | 1
true
ADDR3.Student
ADDR3.name | ""
ADDR3.grade | 0
Enter the name:
ADDR0.next
"Jane"
ADDR3.name | "Jane"
Enter the grade:


inNum | ""
ADDR0.next
"98"
inNum | "98"
ADDR3.grade | 98
ADDR1[1] | ADDR3
ADDR3.getGrade
98
sum | 193
i | 2
true
ADDR4.Student
ADDR4.name | ""
ADDR4.grade | 0
Enter the name:
ADDR0.next
"Sam"
ADDR4.name | "Sam"
Enter the grade:
inNum | ""
ADDR0.next
"60"
inNum | "60"
ADDR3.grade | 60
ADDR1[2] | ADDR4
ADDR4.getGrade
60
sum | 253
i | 3
false
avg | ?
avg | 84.33
Average = 84.3
Above average students:
for
i | 0
true

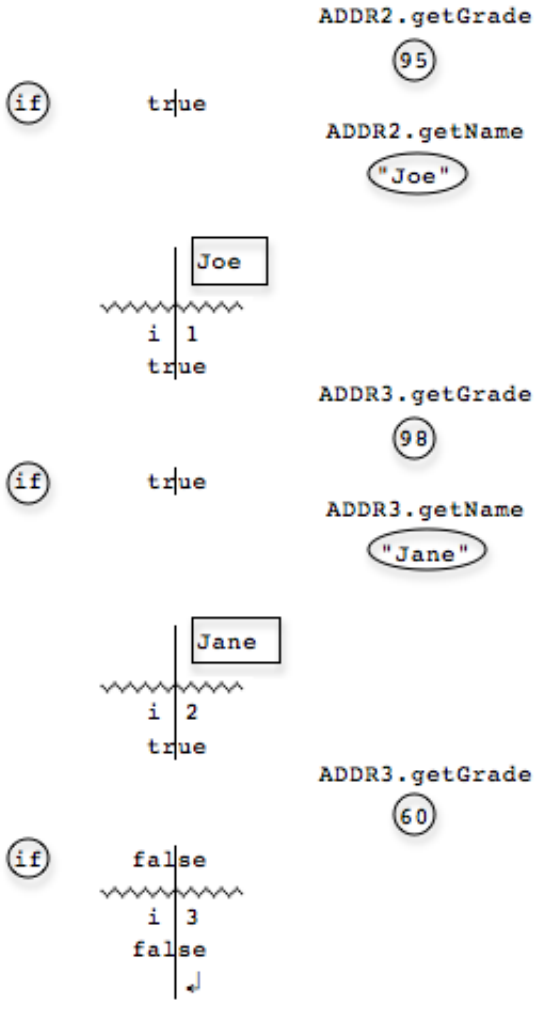


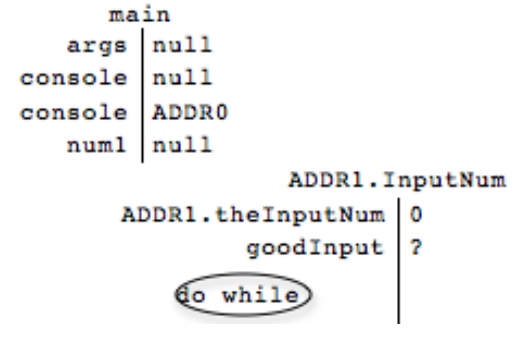
Figure 7

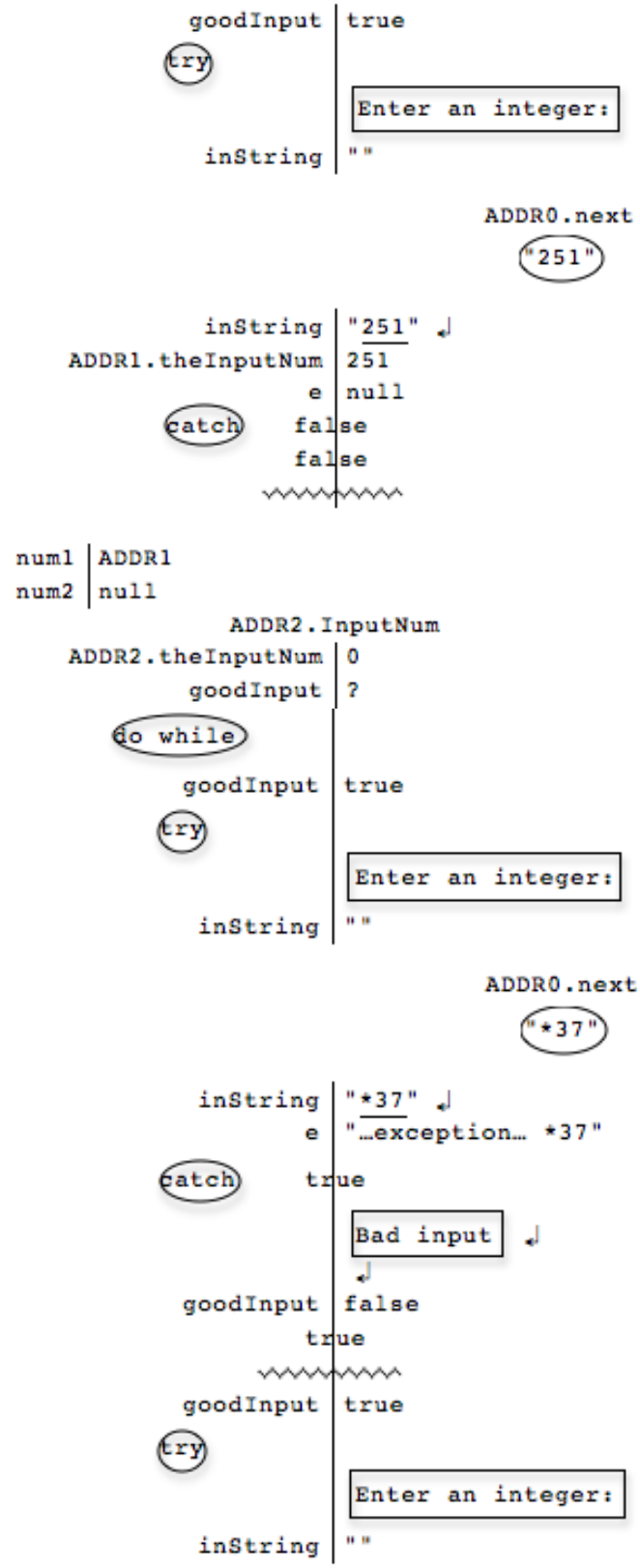
Exceptions and interrupts

The graphical user interface of Java requires that the student be able to handle exceptions and interrupts effectively. When a Java program reads input that is formatted incorrectly, for example, it must handle the resulting exception in a way that allows the user to re-enter the input. The program of Figure 8a prompts the user for two integers and adds them; in Figure 8b, the first input value, 251, is entered correctly, but the user first mistakenly enters *37 for the second input value and then enters 37 when the program detects the error.

```
public class InputNum {

        static Scanner console;
        int theInputNum;

        public static void main(String [] args){
                console = new Scanner(System.in);
                InputNum num1 = new InputNum(),
                        num2 = new InputNum();
                int val1 = num1.getNum(),
                        val2 = num2.getNum(),
                        sum = val1 + val2;
                System.out.println("Sum = " + sum);
        }


        public InputNum(){
                Boolean goodInput;
                do{
                        goodInput = true;
                        try{
                                System.out.print("Enter an integer:");
                                String inString = console.next();
                                theInputNum = Integer.parseInt(inString);
                        }
                        catch  (NumberFormatException e){
                                System.out.println("Bad input");
                                        System.out.println();
                                        goodInput = false;
                        }
                } while(!goodInput);
        }

        int getNum(){
                return theInputNum;
        }

}
```

Figure 8a




goodInput | true
try
Enter an integer:
inString | ""
ADDR0.next
"251"
inString | "251"
ADDR1.theInputNum | 251
e | null
catch
false
false
num1 | ADDR1
num2 | null
ADDR2.InputNum
ADDR2.theInputNum | 0
goodInput | ?
do while
goodInput | true
try
Enter an integer:
inString | ""
ADDR0.next
"*37"
inString | "*37"
e | "…exception… *37"
catch
true
Bad input
goodInput | false
true
goodInput | true
try
Enter an integer:
inString | ""

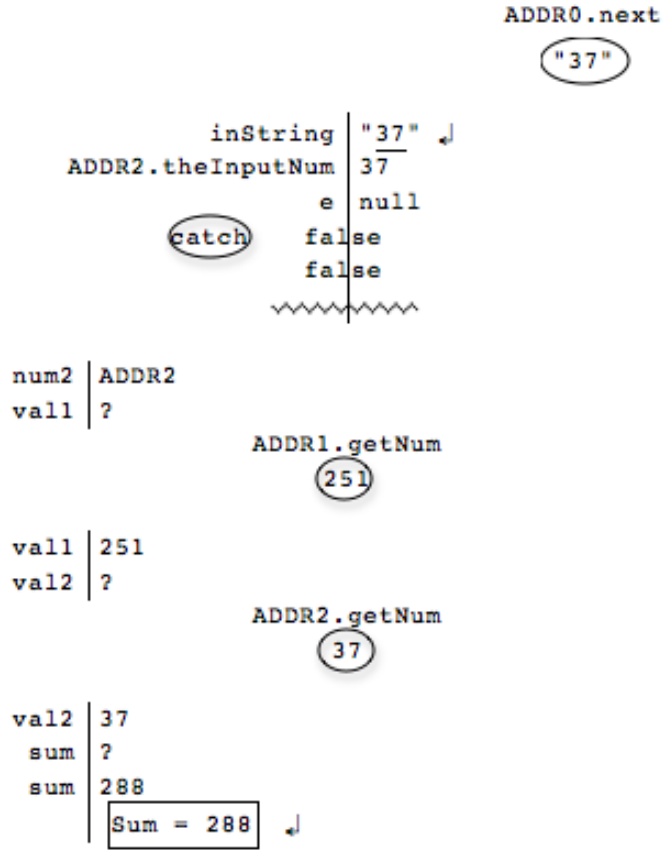


Figure 8b

Copying objects

A topic that is of particular interest in Java programming is the various ways of copying objects. At issue is the question of whether the tracing method can help the student understand the differences among the various techniques**.**

The program of Figure 9 illustrates some of the methods. In the sequence

Triangle t1 = new Triangle(10.0, 15.0, 9.0);
Triangle t2 = t1;

a triangle object, t1, is created and a new object, t2, is set to reference it. In the further sequence

Triangle t3 = new Triangle();
t3.copyTriangle(t1);

t3 is created with the default sides of 0.0, 0.0, and 0.0, and then by means of the Triangle method copyTriangle takes on the values of the sides of t1. Finally, t4 is created, and the function copyTriangle (not the method) is invoked with actual parameters t4 and t1, ostensibly to copy t1 into t4. However, the formal parameters of copyTriangle are copies of the actual parameters and there is no effect on t4.

```
public class Main {
	public static void main(String[] args) {
		Triangle t1 = new Triangle(10.0, 15.0, 9.0);
		Triangle t2 = t1;
		t1.display("t1");
		t2.display("t2");
		Triangle t3 = new Triangle();
		t3.copyTriangle(t1);
		t3.display("t3");
		Triangle t4 = new Triangle();
		copyTriangle(t4, t1);
		t4.display("t4");
	}
	public static void copyTriangle(Triangle ta, Triangle tb){
		ta = tb;
	}
}

public class Triangle {
	double side1, side2, side3;
	public Triangle(double s1, double s2, double s3){
		side1 = s1; side2 = s2; side3 = s3;
	}
	public Triangle(){
	}
	public void display(String tnum){
		System.out.println("Triangle " + tnum + " -- Sides: " + side1
			+ ", " + side2 + ", and " + side3);
	}
	public void copyTriangle(Triangle t){
		side1 = t.side1; side2 = t.side2; side3 = t.side3;
	}
}
```

```
   main
args | null
  t1 | null
             ADDR0.Triangle
        ADDR0.side1 | 0.0
        ADDR0.side2 | 0.0
        ADDR0.side3 | 0.0
                 s1 | 10.0
                 s2 | 15.0
                 s3 | 9.0
        ADDR0.side1 | 10.0
        ADDR0.side2 | 15.0
        ADDR0.side3 | 9.0

  t1 | ADDR0
  t2 | null
  t2 | ADDR0
             ADDR0.display
                    | Triangle t1 -- Sides:10.0, 15.0, and 9.0 ↵

    |
             ADDR0.display
                    | Triangle t2 -- Sides:10.0, 15.0, and 9.0 ↵

  t3 | null
             ADDR1.Triangle
        ADDR1.side1 | 0.0
        ADDR1.side2 | 0.0
        ADDR1.side3 | 0.0

  t3 | ADDR1
          ADDR1.copyTriangle
                   t | ADDR0
        ADDR1.side1 | 10.0
        ADDR1.side2 | 15.0
        ADDR1.side3 | 9.0

    |
             ADDR1.display
                    | Triangle t3 -- Sides:10.0, 15.0, and 9.0 ↵

  t4 | null
             ADDR2.Triangle
        ADDR2.side1 | 0.0
        ADDR2.side2 | 0.0
        ADDR2.side3 | 0.0
```

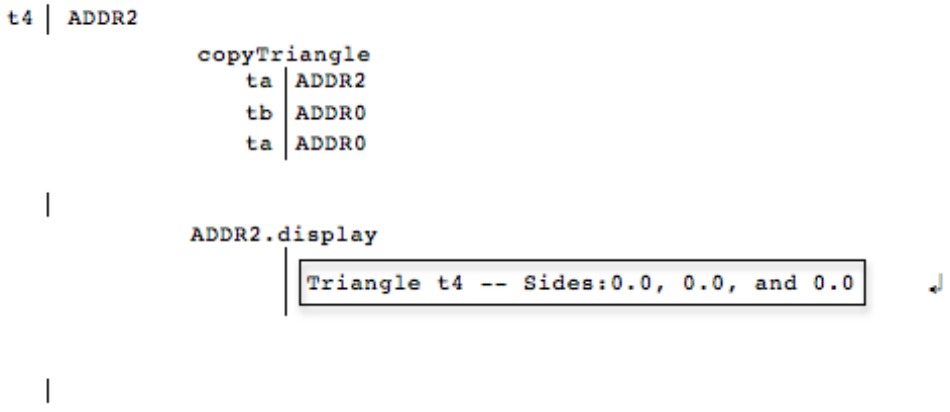


Figure 9

Conclusions

Tracing accounts for many of the features of Java that would be taught to beginning programmers or to students who have had a background in C++. The analysis has touched only lightly on the graphical user interface that is a major part of Java courses; a more detailed analysis might indicate that tracing operations in that environment would resemble those of the exception handling example in Figure 8.

Tracing focuses the student's attention on one step of a program at a time. It can be useful to an instructor who is explaining the execution of some programs, and it can be useful to students who are following the execution of programs that are known to execute correctly. It is not clear whether or how tracing helps students write their own programs. In fact, studies are contradictory whether it is even necessary that students be able to trace programs others have written in order to write their own programs (see Warms[3] for references).

While understanding the techniques covered by the tracing examples might be a necessary step toward having a good knowledge of a language, it is certainly not a sufficient condition. The hope is that using tracing helps a student master basic material and become better able to learn additional material. It may assist students in learning how to understand the use of classes and in dealing with many other features of that language. It is likely that programs that incorporate these features may be traced using the same principles that were used in tracing C++ programs, without introducing complex new notations.

Bibliography

[1] T. M. Warms and R. Drobish, "Tracing the execution of computer programs – report on a classroom method," in Proceedings of the Spring 2007, ASEE Mid-Atlantic Section Conference, Newark, NJ. (CD-ROM proceedings).

[2] T. M. Warms, "The Semantics of Tracing: Transitivity of Reference," Proceedings of FECS'07 — The 2007 International Conference on Frontiers in Education: Computer Science and Computer Engineering, Las Vegas, June 2007, pp. 302 – 307

[3] T. M. Warms, "Using the tracing method and RandomLinearizer for Teaching C++," in Proceedings of the FECS'10 — The 2010 International Conference on Frontiers in Education: Computer Science and Computer Engineering, Las Vegas, NV. pp. 16-22.
[4] T. M. Warms, Qiang Duan, and Kavon Farvardin, "Can Using a Formal System For Tracing Computer Programs Help Students Learn Introductory Computer Science?"
[5] B. Eckels, "Thinking in Java". Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006).