Using Contour Diagrams and JIVE to Illustrate Object-Oriented Semantics in the Java Programming Language

James T. Streib
Department of Computer Science
Illinois College
Jacksonville, IL 62650 USA
jstreib@ic.edu

Takako Soma
Department of Computer Science
Illinois College
Jacksonville, IL 62650 USA
tsoma@ic.edu

ABSTRACT
Organick, Forsythe, and Plummer extensively used contour diagrams to illustrate the semantics of procedural and functional programming languages. However, many of the benefits of contour diagrams can still be utilized in explaining the semantics of object-oriented languages and is evident in the software JIVE created by Gestwicki and Jayaraman. This paper illustrates the use of both contour diagrams and JIVE to help students understand passing objects via parameters, returning objects from methods, and using inheritance in the Java programming language. The results from a small informal survey are included. This paper is useful to those teaching Java in a first-year computer science sequence.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education – Computer Science Education.

General Terms
Design, Languages.

Keywords
Contour Diagrams, JIVE, Object-Oriented Semantics, Java.

1. INTRODUCTION
As colleges have shifted from teaching procedural languages to object-oriented languages in the first-year computer science course sequence, there are a number of object-oriented concepts that are now introduced much earlier than they might have been previously. These concepts are sometimes very difficult for first-year students to comprehend and this is especially true when students are just examining code and only have verbal descriptions to supplement the code.

Although UML diagrams are useful in understanding the overall design of an object-oriented program, sometimes students have difficulty understanding the semantics of object-oriented code. In various attempts to help with this understanding, textbook authors such as Malik and Nair use diagrams to help distinguish between reference variables and objects [4]. In Wu’s textbook, he expands the notion of a UML diagram into “state of memory diagrams” to assist students in understanding passing and returning objects to and from methods and the concept of inheritance [6].

Although these attempts are very good and have been used by the authors of this paper in their classrooms, there is a supplemental method used by the authors to help students with the understanding of the semantics of Java programs.

2. CONTOUR DIAGRAMS AND JIVE
Organick, Forsythe, and Plummer used the idea of snapshots utilizing contour diagrams in their text Programming Language Structures to illustrate how sub-programs work, how various types of parameter mechanisms transfer data to and/or from sub-programs, and how recursion can be modeled [5]. At the time their book was published (1978), the mainstay of programming involved procedural languages and they also included sections on functional languages. They pointed out that the original designer of the Contour Model was Johnston [2]. Others to follow Organick, et al, included MacLennan in his programming languages text [3].

Although contour diagrams do not seem to be used as frequently in texts today, the authors of this paper have found the basic model has remained quite durable and modifiable to help in presenting many object-oriented concepts. The authors of this paper use modified versions of contour diagrams to help students understand many of these fundamental concepts and have also found Gestwicki and Jayaraman’s JIVE (Java Interactive Visualization Environment) software [1] useful to help facilitate visual learning of advanced object-oriented concepts and help correct various student misconceptions. Although there are a number of features in the original model that are helpful to students in the classroom, some have been changed here to help save time and space in this paper. The first example uses contour diagrams and the second example uses the JIVE software.

3. PASSING AND RETURNING OBJECTS USING CONTOUR DIAGRAMS
The passing, comparing, and returning objects can be difficult concepts to grasp, especially in a first-year sequence in Java programming. The following example program illustrates the passing of an object from one method to another as well as returning an object from one method to another. In actual classroom discussion, homework, quizzes, or exams, these concepts could be separated into different examples to help
students understand the concepts individually, but they are combined here for efficiency. Also a number of contours have been omitted here to save space, where the omitted contours can be added at the discretion of the instructor based on the needs of the students.

```java
public class Geometry1 {
    public static void main(String[] args) {
        double radius1, radius2;
        Circle circle1, circle2;
        circle1 = new Circle(1.0);
        circle2 = circle1.makeObj();
        circle2.copyRadius(circle1);
        radius1 = circle1.getRadius();
        radius2 = circle2.getRadius();
    }
}
```

It should be noted that both classes and methods are drawn as contours, except the main method in order to save space. When a contour is nested, it has access to all the variables that encompass it, but does not have access to any other contours. Further, in the original model, superscripts were used on contours of subprograms to help illustrate recursion, however since recursion is not being used here, superscripts are used only to indicate instances of objects. Also, note that the name of a variable is in the first column, the type is in the second column, and the contents are in the third column, where a blank cell indicates an uninitialized variable. The original model included a return value and that upon return from the new method, a reference to the new instance of the class Circle is returned and then assigned to the variable circle1 by the assignment symbol in line 5. After a contour is shaded, such as the constructor Circle(r) below, it no longer needs to be shown in subsequent figures in order to save space. However, it can also be retained to allow one to refer back to previous logic or for the purposes of grading an assignment, quiz, or exam. The contour below shows the state of execution just prior to the execution of line 6 in the main program (fig. 1).

![Figure 1: Contour Diagram prior to Line 6](image1)

After starting line 6, the contour for the value returning method makeObj is created in Circle1 below. After invoking the new method in line 25, and the execution of line 15 in the constructor Circle(), note that memory cells are not allocated for local variables or parameters. After line 15 in the constructor and the completion of the statement in line 25, just prior the return statement in line 26, the variable radius has the value 0.0 stored in it and the constructor is again shaded. This helps illustrate to students that the reference to this new instance of the class Circle, indicated via an arrow to Circle2, is assigned to the memory location temp in Circle1 and not to a variable in the main program.

The following drawing (fig. 2) then shows the state of execution after the return statement in line 26 has been executed, and just prior to the execution of line 7 in the main program. As with the constructors previously, note that both the constructor and the value returning method makeObj are shaded to indicate that they are deallocated. Also, a copy of the reference to the second instance of the Circle class has been returned to the main program and then assigned to the variable circle2 as an arrow. The previous reference is optionally shown as a dotted arrow to illustrate that it too has been deallocated along with the method.

![Figure 2: Contour Diagram prior to Line 7](image2)
After starting execution of line 7 and just prior to line 29 in copyRadius method, a copy of the argument circle1 from the main program appears in the parameter c of type Circle in the method, copyRadius. Then after executing line 29, the radius, 1.0, of the object c pointing to circle1 is assigned to the variable radius in the current instance of Circle, circle2, illustrating to students how values can be copied from one object to another. After the assignment statement is complete, the control returns back to line 8 in the main program and the copyRadius contour is deallocated and shaded, and the reference shown as a dashed arrow (fig. 3).

Another source of confusion to beginning students, is when they try to compare two different instances of a class. Sometimes they try something like, “if(circle1 == circle2)” and are confused that “1.0 == 1.0” is false. The advantage of a contour example such as the one above is that the student can see what they are in fact comparing is not the contents of radius, but rather the references to each object, where the references represented by the arrows are not equal. The result is that the student should see that a method is necessary to gain access to the private data members within each instance and be able to correctly write the necessary code.

### 4. ILLUSTRATING INHERITENCE USING JIVE

This second example is more in depth than the first example and shows a number of complicated features of object-oriented programming in Java such as typecasting and the methods this and super. As with the first example, these features can be broken up into simpler examples for classroom presentation but are combined here to save space. Further, as was done toward the end of the previous example, various steps are skipped and can also be omitted in the classroom, however they can be reintroduced if necessary to review key concepts.

```java
public class Circle {
    double radius;
    public Circle(double r) {
        radius = r;
    }
    public double area() {
        return Math.PI * radius * radius;
    }
    public double circumference() {
        return 2 * Math.PI * radius;
    }
}

public class Cylinder extends Circle {
    protected double height;
    public Cylinder(double r, double h) {
        super(r); // radius = r;
        height = h;
    }
    public Cylinder(Cylinder c) {
        this(c.radius, c.height);
    }
    public double area() {
        return 2 * super.area() + height * circumference();
    }
}
```

```java
public class Geometry2 {
    public static void main(String[] args) {
        Circle shape1, shape2;
        double area1, area2;
        shape1 = new Cylinder(2.5, 5.7);
        shape2 = new Cylinder((Cylinder)shape1);
        area1 = shape1.area();
        area2 = shape2.area();
    }
}
```

Obviously similar to the hand written contours, the JIVE contours sometimes have more information than needed and use a slightly different notation, however after students have mastered the hand written contours, the transition to the JIVE model is relatively easy. Some of the obvious differences are that JIVE uses “rpdl” and a light gray arrow for the return label, constructors are labeled as “<init>” and it can be determined which one is which by their nested location, and references to other objects are referred to as “instance of…”, which can be a little confusing.

When line 5 in the main program is executed, the constructor Cylinder is allocated as a subclass and it invokes the corresponding constructor in the superclass Circle via `super(r)` in line 26. The following figure (fig. 4) shows the state of execution after line 14 in the Circle constructor, but prior to the execution of line 27 back in the Cylinder constructor. The Circle constructor is
shown inside the contour for Circle because, that is where it is defined and this helps illustrate to students what variables are accessible, such as radius in Circle, but not height in Cylinder.

Figure 4: JIVE Screenshot prior to Line 27

After the execution of line 14, the Circle constructor is deallocated. Then the following figure (fig. 5) shows the state of execution just prior to the execution of line 6, where the Cylinder constructor has also been deallocated and the value in h has been copied to the variable height. Further, it should be noted that the new method has returned a reference to an object of type Cylinder and that reference has been placed in the memory location of shape1 of type Circle. At first this might bother some students, but a reminder at this time is usually helpful that the class Cylinder is a subclass of the Circle class and the assignment is perfectly legal.

Figure 5: JIVE Screenshot prior to Line 6

The next figure (fig. 6) shows the state of execution as line 6 has begun and just prior to the execution of line 30 in the copy constructor. As before, first the contours for Cylinder and Circle are drawn where the variables for height and radius are declared, respectively. Then, the contour for the copy constructor public Cylinder(Cylinder c) is created where the memory cell for the parameter c is given as type Cylinder. However, shape1 in the main program is of type Circle, where Circle is the superclass and Cylinder is the subclass, which is essentially the exact opposite of what happened in the previous figure. Here, a helpful reminder to students is that unlike the previous figure, a reference to a superclass cannot be placed into a memory cell of the type of the subclass. As a result, the use of a typecast operator is required in the argument in line 6. The parameter c now also has a copy of the reference to the same cylinder object as referenced in shape1.

Figure 6: JIVE Screenshot prior to Line 30

Although the copy constructor could have been written similar to the other constructor, as written it allows the opportunity to illustrate the use of this, where the following figure (fig. 7) shows the state of execution after line 30, just prior to the execution of line 7. When line 30 is executed the contents of the arguments c.radius and c.height are sent to the constructor on line 25 and the parameters r and h contain the corresponding values. Then as before, the constructor for Circle is executed via super(r), where the argument r is sent to the corresponding parameter r, and then the value for r is copied to the data member, radius.

Figure 7: JIVE Screenshot prior to Line 27

After the value for r is copied, the Circle constructor is deallocated, and after assigning the value of the parameter h to the variable height in line 27, control is returned to the copy constructor after line 30, where the Cylinder constructor is deallocated. From there, control is transferred back to the main program, where the copy constructor deallocated as well. Then back at line 6, the reference to the newly created Cylinder is placed in shape2, where the following figure (fig. 8) shows the state of execution just prior to line 7 back in the main program.
5. COMPARISON OF CONTOURS AND JIVE

As can be seen in both of the above examples, the use of contour diagrams and JIVE have the potential to help demonstrate and clarify various semantics in object-oriented programming. In an informal survey of a small second semester CS 2 class of nine students at a small liberal arts college, four questions asked of the students were whether the use of each of the following listed below helped in their understanding of objects in Java. The responses were on a scale of 1 to 5, where 1 represents it did not help, 3 represents that it helped somewhat, and 5 represents that it helped a lot. The average for each appears below:

- Contour diagrams in the classroom: 3.78
- Contour diagrams in your homework: 3.33
- JIVE in the classroom: 3.56
- JIVE in your homework: 3.11

The results in all four cases were between 3 and 4, which indicates a positive result, but is a little less than what was expected. However, the specific responses for the first category with a 3.78 average was that two responded with a 5, five with a 4, one with a 3, and one with a 1, where the one low response significantly pulled down the average given the small sample size.

Another aspect of the data that seems somewhat contradictory is that the use of hand-drawn contour diagrams scored correspondingly higher than the use of the software JIVE, where two or three of the responses were either a 1 or a 2 and brought down the JIVE average. One reason why that might have happened is that a couple of students appear to have missed a key lecture on how to use the software. As a result, they might not have adequately learned how to use the software correctly and thus the software might not have been as helpful to them.

Lastly, the students indicated that the use of contour diagrams and JIVE in the classroom helped them more than using either in their homework. At first, this seems somewhat counterintuitive, where it would seem that active learning on homework should provide more understanding than passive learning in the classroom.

However, the reason for this might be similar to when a student thinks they understand how to solve a mathematical problem by observing an example on the board in the classroom, but they subsequently have difficulty when they attempt to actually solve a similar problem in the exercises in the back of the textbook. The result is that when working on the homework it might appear that one doesn’t have as much understanding, but in reality it is the homework that actually helps one gain real understanding.

6. SUMMARY AND CONCLUSIONS

Whichever of the two methods should be used (drawing contours by hand or using JIVE) is largely a matter of preference or need by the instructor and/or students. The disadvantage of drawing is that it can be slow and tedious, which is a real advantage for using JIVE. However, using an automated system of JIVE has the disadvantage that a student might just passively point and click their way through the contours without actually thinking much about the process, whereas when students are drawing the contours, they are forced to think about the next step and draw it accordingly.

The authors of this paper actually use both methods. The drawing of contours can help pinpoint students’ misconceptions when working with the instructor or a tutor. Not erasing deallocated contours can also be helpful to assess the students’ understanding of the material early on in homework prior to the introduction of JIVE, and especially on quizzes and exams, where they do not have access to JIVE. On some medium-sized programs, the authors previously used prepared overhead transparencies with the contours already drawn, where the instructor could then quickly write in the necessary values and references in the pre-drawn spaces for the variables. With more complex programs the hand drawn method becomes untenable, however with the advent of JIVE, both medium-sized and more complex programs can be demonstrated easily due to JIVE’s convenience, speed, and the ability to move forward and backward through the contours, at will. In the end, both contours and JIVE appear to have made a difference in students’ understanding of various object-oriented concepts. It is expected that with further experience using both of these tools by the authors, that students should gain an even better understanding of object-oriented semantics in the future.

7. REFERENCES