# Java Class Visualization for Teaching Object-Oriented Concepts

**Herbert L. Dershem and James Vanderhyde**
**Department of Computer Science**
**Hope College**
**Holland, MI 49422-9000**
**dershem@cs.hope.edu**

## Abstract

Visualization is a useful tool in many areas of computer science education. This paper describes the use of visualization in the introduction of object-oriented concepts. A Java application has been developed that allows the user to interact with a visualization of any Java class through the instantiation of objects, the movement of those objects around the class environment, and the activation of class methods. The user may also move conveniently between classes in this visualization.

This Object Visualizer is useful for classroom demonstration, individual student use in the laboratory, and class debugging and testing.

## 1. Introduction

Students who are learning object-oriented programming frequently have a difficult time understanding this new paradigm and how it operates. As with any difficult subject, students best learn this topic if a variety of learning modes are used. In particular, visual learning can be effective and the visualization of classes, objects, and methods can be an important learning tool.

Previous work on the visual approach to learning object-oriented concepts includes that of Kölling and Rosenberg [1] who have created a program development environment including a language, debugger, and editor, that uses graphics for the visualization of class inheritance relations and the dynamic creations of objects. Another approach is that of Haddad, Curtis, and Brage [2]. They have built a graphical user interface for the visualization of object-oriented concepts. Jerding and Stasko [3] have developed a tool for visualizing the execution of C++ programs with the intent of enhancing program understanding.


SIGSCE 98 Atlanta GA USA


The testing of object-oriented programs is also a problem for both teachers and students in beginning courses. Since teachers focus on the design and implementation of encapsulated classes, student construction of a test program is usually an unwelcome diversion, introducing many details in its user interface that distract the student from the issues of class design and implementation. The actual evaluation of the test can also be difficult due to the encapsulated nature of classes. The use of a symbolic debugger is less than satisfactory since it operates below the conceptual level of the class structure. The relationship between visualization and debugging and testing is well-known and has been explored by Baecker, DiGiano, and Marcus [4]. The applications of visualization to object-oriented debugging has been demonstrated by Mukherjea and Stasko [5].

The need for a visual environment that will enhance student understanding of object-oriented concepts and facilitate testing and debugging is therefore desirable. Rosenberg and Kölling [6] have provided such facilities within their system. Their work requires the development of the system within the Blue environment whereas the present work permits the visualization of any Java class.

This paper describes a tool that can be used by students to visualize the interaction of class components and assist in debugging student-written classes. This tool is a Java application that accepts a Java class as input and produces a window that contains a visualization of the class' methods. Objects of this class may be instantiated and manipulated within the environment. In particular, objects may be placed as arguments for the invocation of methods, which may result in the instantiation of further objects. This visualization and manipulation of class components provides a useful tool for learning object-oriented concepts.

## The Object Visualizer

We will illustrate the features of the Object Visualizer through the examination of a simple example fraction class called `Frac`. The Java view of this class is found in Figure 1 and the view of this class produced by the Object Visualizer is shown in Figure 2.

```
public synchronized class Frac extends java.lang.Object
{
    public int Num;
    public int Den;
    public Frac(int,int);
    public Frac();
    public void reduce();
    public static Frac add(Frac,Frac);
    public static Frac subtract(Frac,Frac);
    public static Frac multiply(Frac,Frac);
    public static Frac divide(Frac,Frac);
    public java.lang.String toString();
}
```

Figure 1. Frac class methods defined in Java

Each class is represented by a unique color. These colors are difficult to discern in the Figures found in this paper since they are represented by shades of gray. Each method in the class is represented by a button-box along the left edge of the window. Eight methods including two constructors are defined in the class `Frac`. The box contains the name of the method and the color of the box is the color associated with the return value of the method. For example, the two constructors and functions `add`, `subtract`, `multiply`, and `divide` all return an object of class `Frac` and have their function boxes colored with `Frac`'s assigned color. Methods that return `void`, such as reduce, are colored black. The function `toString` returns an object of `String` class and is colored appropriately.

At the bottom of the left edge are three boxes that are included by the Object Visualizer in every class. The first

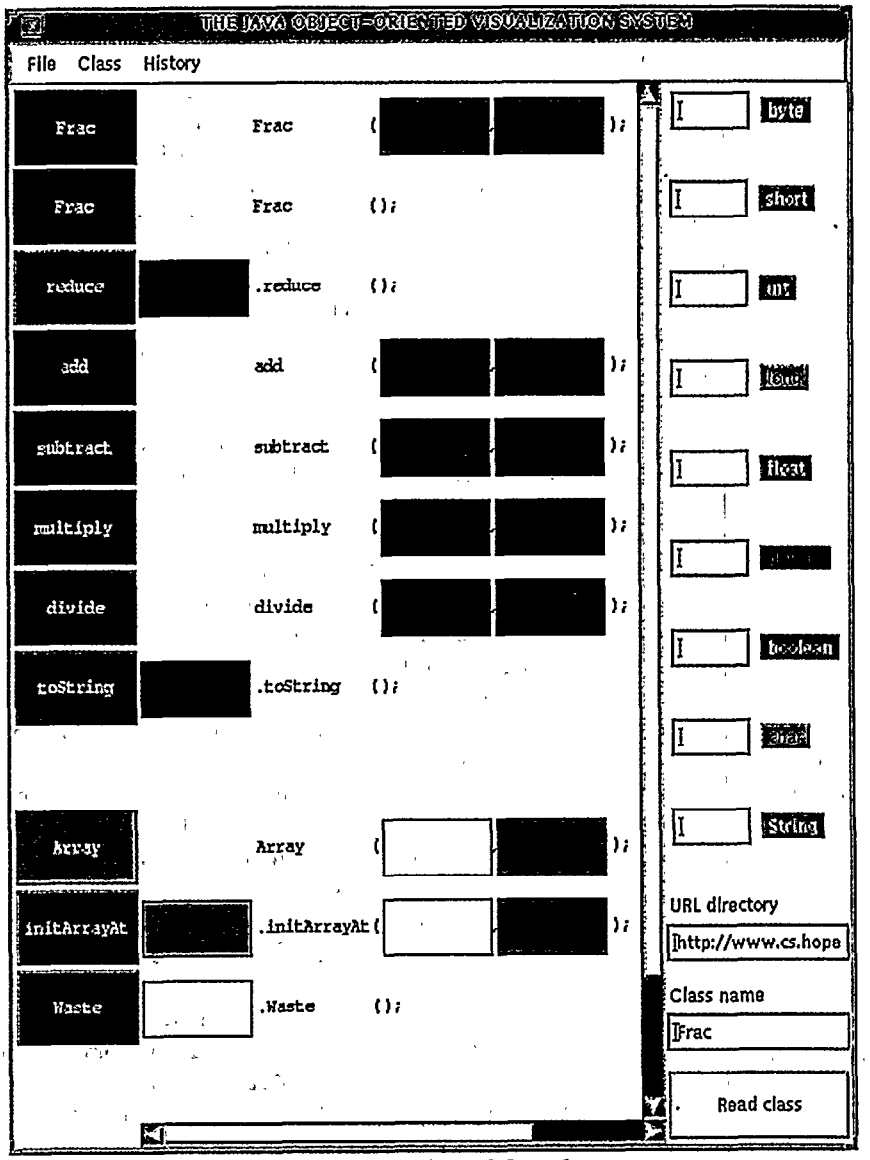


Figure 2. Object Visualizer view of Frac class

two enable arrays of the class to be instantiated and assigned values. The third is a `Waste` box where objects are placed to remove them from the environment.

To the right of each method box, a prototype of that method's invocation is found. Within that prototype, container boxes represent method arguments and, for each object method, a container box of the defining class represents the invoking instance. Once again, all boxes are color-coded to the class or type that is represented. For example, the top constructor has two arguments, both of `int` type, so the boxes are of `int`'s color. All other boxes in the class methods of Figure 2 are of class `Frac`.

The right frame of the window in Figure 2 contains initializer buttons for the Java primitive data types and for the class `String`. This frame remains the same for any class being visualized. Each of these buttons permits the creation of an object of the corresponding type when a value is typed into the adjacent text field and the button is clicked.

The directory and class text fields at the bottom of the right frame are used to specify the location of the class that is being visualized. These can be changed at any time, and pressing the "Read Class" button results in the loading of the new class into the viewer.

## Object Manipulation

We will now work through some manipulations of objects within the class visualization shown in Figure 2. This will illustrate some important features of the Object Visualizer.

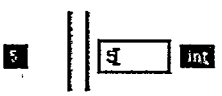


Figure 3. Instantiation of 5

In order to instantiate an object of class `Frac`, we must first instantiate some `int`s. Figure 3 illustrates the instantiation of the integer 5. The user first types 5 in the text field next to the `int` button. A mouse click on that button then results in the appearance of an object to the left of the button and inside the left frame of the window. This object has the value 5 displayed within it and is colored with the color of `int`. Like all other objects, this object may be dragged via the mouse to any location within the left frame. In particular, Figure 4 shows it located within the left argument container of the `Frac` constructor. It also shows an `int` object containing 7 in the right argument container and the `Frac` object "5/7" that appears when the `Frac` constructor button on the left edge is clicked with the `int`s in the argument containers as shown.


Figure 4. Instantiation of Frac 5/7

The "5/7" `Frac` object can now be dragged to any location in the frame. Figure 5 shows the result of clicking function `add` after the two `Frac`s shown have been dragged to the argument holders.


Figure 5. Activation of add function

Object methods such as `reduce` may modify the contents of the invoking object. Figure 6 shows a before and after view of a `reduce` method invocation.


Figure 6a. Before activation of reduce


Figure 6b. After activation of reduce

The text that appears in the object box is the result of the `toString()` method applied to the object. To view the data components of an object, the user must meta-click the mouse button on the object as shown in Figure 7.

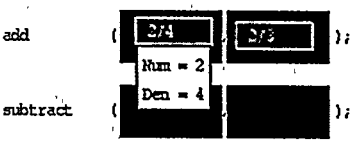


Figure 7. View of object components

## Other Features of the Object Visualizer

The Object Visualizer window presents the view of one class at a time. The class being viewed may be changed at any time, however, by specifying a new class at the bottom of the right frame and clicking on the "Read Class" button. The

specified class will have its methods appear in place of the methods of the original class. The objects that are in the window persist, however, until they are eliminated by placing them in the "Waste" receptacle.

This feature permits the user to move easily from one class to another within the Visualizer. Objects can be created in one class and then used in methods of another. It is particularly easy to move from the currently viewed class to its parent class by selecting "Move to Parent" from the "Class" pull-down menu at the top of the window. The "History" menu facilitates returning to a view of classes that were viewed previously.

A class that has not appeared in the Visualizer during the current execution is represented by the color black until it is loaded into the Visualizer for the first time. At this time, a color is assigned to that class and its future appearance in the Visualizer within other classes will include that newly-assigned color. Colors chosen to represent classes and primitive types are arbitrarily chosen by the system with two exceptions. White is used for `java.lang.Object` class. Arrays are represented by a lighter shade of their base class.

If method argument containers do not contain an object of the correct type or class at the time the method is invoked by a button click, an "Illegal Argument" window appears.

Larger classes can be conveniently viewed by using the horizontal and vertical scroll bars in the viewing frame. This is particularly helpful when viewing built-in Java classes such as java.awt.Component as shown in Figure 8.Implementation

## Implementation

An overview of the implementation of the Object Visualization Application may be of interest. The graphical user interface is actually quite simple. The application hears mouse-down and mouse-drag events and responds with a change in location of the "Animated Object" (the `AnimObject` interface) and a repaint of the visible methods. The Animated Objects are drawn as a colored rectangle with a helpful summary of the contents. If the class has a `toString()` method, it is used as this summary. Each primitive maker has a default value that is created when the button is hit if the input field is empty. Scrollbars are used with separate Scrollpanes to view the class a little at a time. Below the class' methods is a Waste method that is used to lose the reference to an object. When this is activated, the draggable object disappears and the instance itself will be garbage-collected.

The invocation of the methods and constructors is accomplished through use of the `java.lang.Class` class and the `java.lang.reflect` package. To get an instance of the `Class` class you simply call `getClass()` on the object in question. You then use this `Class` object to get the members of the class using `getDeclaredFields()`, `getDeclaredConstructors()`, and `getDeclaredMethods()`. These each return an array of a class from the `java.lang.reflect` package. You call `newInstance()` on a `Constructor` and `invoke()` on a `Method`. Each of these two methods takes an array of `Objects` as a parameter to pass to the function being called.

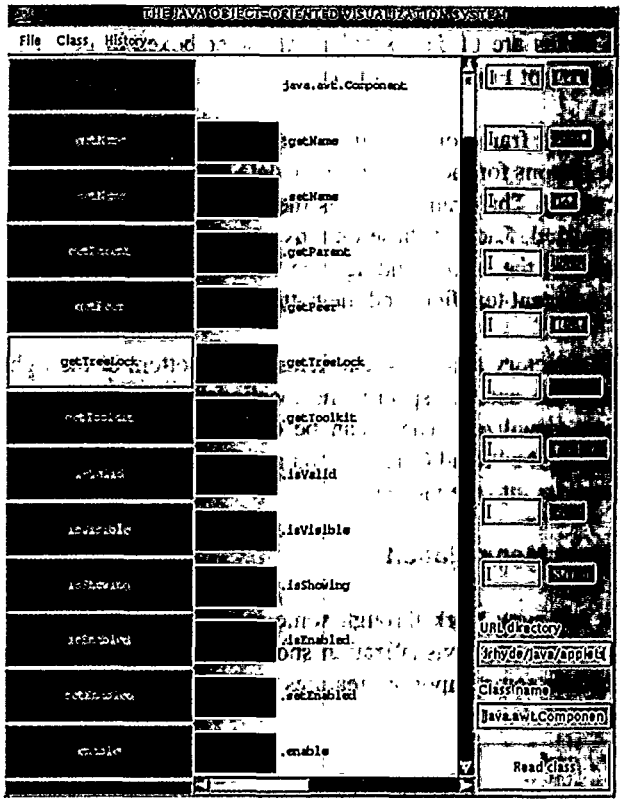


Figure 8. Object Visualizer view of java.awt.Component

The Internet class loading is also quite simple. First the program creates a `URLConnection` to a class file and downloads an array of bytes. This can then be turned into a class using the `defineClass()` method in the `java.lang.ClassLoader` class. `ClassLoader` is an abstract class, so it is extended to create an `InternetClassLoader` class. Once a class is defined, it may then used as much as needed. Unfortunately, only public methods and constructors may be used.

## Educational Visualization

The Object Visualizer is a valuable tool in courses where the object-oriented paradigm and Java are taught. This software is very flexible and can be used in a wide variety of settings and fulfill many roles.

Classroom discussion of the object-oriented paradigm and of specific classes can benefit from demonstrations with the Object Visualizer. For example, the `Frac` class in Figure 1 can be presented in class as both Java code and in visual form. The visualization of class actions promotes better student understanding and gives the instructor an opportunity to point out key activities and relationships.

Visualizations can also provide useful laboratory activities since students are able to manipulate objects within the class and between classes. Since the Object Visualizer works with any Java class, student-written classes can interact with instructor-provided classes, resulting in interesting activities that can be feasibly directed and completed within the time constraints of a closed laboratory period.

The Object Visualizer also provides an excellent environment for students to test and debug classes that they write. This eliminates the necessity of creating elaborate test harnesses for every class and allows the student to test the interaction of classes. This use is particularly helpful in lower-level courses that introduce the object-oriented paradigm since it allows the students to concentrate on class design and implementation rather than the language and user-interface details that are need to construct a test program.

The ability of the Object Visualizer to accept any Java class also makes it an interesting tool for students in studying external classes. In particular, all classes within the Java API are viewable, a useful tool to help students understand the role and function of these classes. As students learn about the `java.awt` package, for example, they can visually create components and place them within containers, seeing the results of the methods as they are invoked.

Finally, students find the Object Visualizer to be a useful tool for exploring remote classes that can be downloaded from the web but whose source code is not available. Many interesting approaches can be explored by examining the structure of a class.

## Availability

The Object Visualization class and its Java source code are available at `http://www.cs.hope.edu/~alganim`.

## Acknowledgments

The authors would like to thank Peter Brummund, who contributed many helpful ideas to this project. This work was partially funded by the National Science Foundation, Grant Number CDA-9423943-03.

## References

[1] Kölling, M. and Rosenberg, J. An Object-Oriented Program Development Environment for the First Programming Course. *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)*, 28,1 (Mar. 1996), 83-87.

[2] Haddad, H., Curtis, E., and Brage, J. Visual Illustration of Object-Orientation: A Tool for Teaching Object-Oriented Concepts. *The Journal of Computing in Small Colleges*, 12, 2 (Nov. 1996), 83-93.

[3] Jerding, D.F. and Stasko, J.T., Using Visualization to Foster Object-Oriented Programming Understanding. *Technical Report GIT-GVU-94-33*, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, July, 1994.

[4] Baecker, R., DiGiano, C., and Marcus, A. Software Visualization for Debugging. *Communications of the ACM* 40, 4 (Apr. 1997), 44-54.

[5] Mukherjea, S. and Stasko, J.T. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger. *ACM Transactions on Computer-Human Interaction*, 1, 3 (Sep. 1994), 215-244.

[6] Rosenberg, J. and Kölling, M. Testing Object-Oriented Programs: Making it Simple. *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (SIGCSE Bulletin, 29, 1 (Mar. 1997), 77-81.