A Java Execution Simulator

Steven Robbins
Department of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

ABSTRACT

This paper describes JES, a Java Execution Simulator that allows users to explore how a Java program executes. This interactive simulator displays a representation of a Java program and animates the running of the program. Instructors can use JES to demonstrate how data is moved when variables are assigned, when parameters are passed, and when values are returned by a method. JES is useful for comparing how primitive and object values are manipulated. The simulator also demonstrates scope rules, object creation, inheritance and polymorphism. While the simulator only supports variables of type double and object and does not support conditionals or looping, it allows users to write general Java programs that might be used as examples in the first weeks of a CS 1 course. JES also has support for arrays of doubles and objects. The simulator is written in Java and can be run as an application or an applet. Support for the simulator includes a simple mechanism for quickly running the simulator on a program developed with a standard Java development system.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*

General Terms

Languages

Keywords

Java programming, simulation, tracing

1. INTRODUCTION

A key skill usually taught in the first computer science course is how to trace a program. Tracing is a useful for understanding how programs behave and for finding logical errors in programs. Textbooks have difficulty describing how to trace a program, since tracing produces diagrams that change as the trace progresses. Each variable is typically represented by a box containing the value of the variable. As the trace proceeds, the contents of these boxes change. When a variable is of primitive type, multiple values can be put in its box with the old values being crossed out as the new values replace it. For reference variables the process is more difficult, and diagrams can become overly complicated. Students who copy a trace diagram from the board (if they can write fast enough while trying to understand what is happening) are left with the end result of the trace in their notes. This end result does not reveal the steps needed to reproduce the trace, and so it is not very useful as a study aid. The situation becomes even more complicated when objects are created and destroyed dynamically.

The difficulty in teaching students how to trace programs motivated me to develop this simulator. The simulator is itself a Java program and can be run on any system used to teach Java programming. Users can write their own programs to use as input to the simulator. Although they have a number of limitations, acceptable programs are sufficient to illustrate the general procedure for tracing a program and include a large subset of the programs usually discussed at the beginning of the first programming course.

The simulator comes with a number of premade examples that are described below and illustrate the scope of the simulator.

Example 1
- declaration of a primitive variable without initialization
- declaration of a primitive variable with initialization
- assignment of a double expression to a variable
- creation of an object using new
- passing double parameters
- assignment of an object to a variable
- changing the state of an object using an alias
- use of "this"
- a method that returns a primitive value

Example 2
- comparison of primitive and object parameter passing

Example 3
- declaring an array variable
- creating an array, either with new or with a list of values
- array assignment
- passing an array as a parameter
- changing elements of an array in a method
- polymorphism

Example 4
- inheritance: shape - rectangle - square

While many development environments can show class diagrams and list objects as they are created, none that I have seen can show the detailed execution behavior illustrated by this simulator. Describing a simulator of this type is best done by a demonstration of the running program. Since the simulator can be run as either an applet or an application, the simplest way to run a demo is through

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

a web browser. All of the examples in this paper can be run from a standard Java-enabled browser without the need for any installation or configuration [2]. The purpose of this paper is to illustrate enough of the power of the simulator to entice the reader to try it out on the web. The printed version of this paper is in black-and-white, while the simulator uses color to enhance the display and make it easier to understand. The reader will need to use his/her imagination to envision the full power of the simulator.

The rest of this paper is organized as follows. Section 2 describes the first example program in detail and shows how the simulator can be used to illustrate basic concepts in Java programming. The next three sections cover parameter passing, arrays, and inheritance. Section 6 describes some additional features and Section 7 discusses some of the limitations of the simulator. Section 8 describes the availability if the simulator.

2. A SIMPLE EXAMPLE

Figure 1 shows the simulator display when it starts up with a simple program. The main class is displayed. When the user pushes the **Start Main** button, JES allocates variables (boxes appear next to the variables declared in **main**) and displays an arrow pointing to the first line to be executed. Figure 2 shows this. The boxes next to the declarations of **var1**, **var3**, and **shape2** are empty, indicating that they have not yet been explicitly initialized. Although Java initializes all attributes, the simulator shows an empty box for all attributes that have not been explicitly initialized. The box next to **var2** shows the initial value of **1.23** and **shape2** is initialized to null. The arrow next to the first line of code shows that it will be executed next. Since this is an assignment statement, a box appears to the right of the statement to hold the value of the expression.

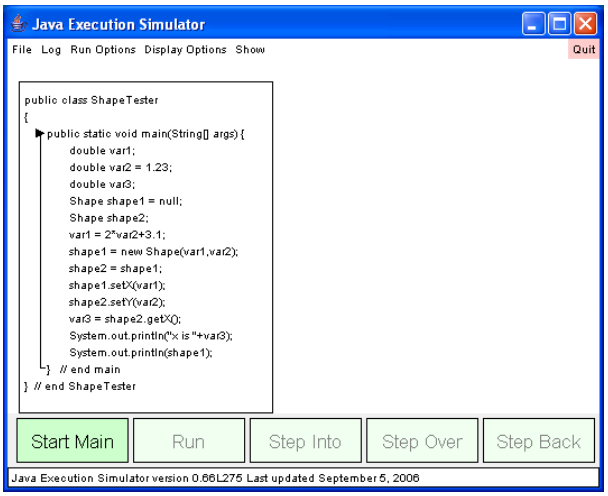

Figure 1: The initial display when JES starts.

JES allows you to step through the program. Each line of code takes one or more steps to execute. The simple assignment statement for a primitive variable, as shown in Figure 2, executes in two steps. In the first step, the expression on the right hand side is evaluated and the value is stored in the box. JES shows an arrow from the box next to the expression (the source box) to the storage location of the assigned variable (the destination box). (See the left half of Figure 3.) In the second step the value is moved from the source box to the destination box. If the simulator is set to show animation, it animates the movement of the value. In either case the program counter moves to the next line of code as shown in the right half of the figure.

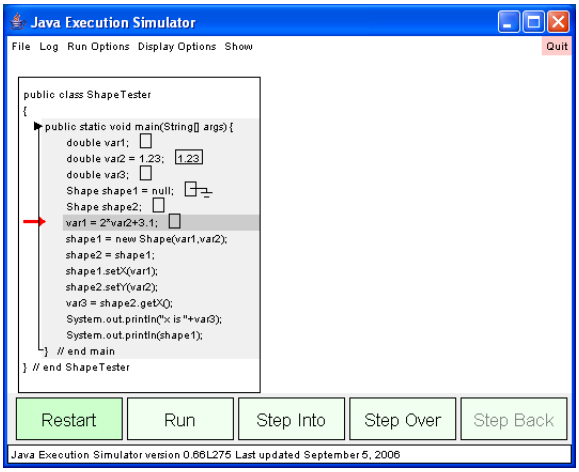

Figure 2: The Start Main button has been pushed.

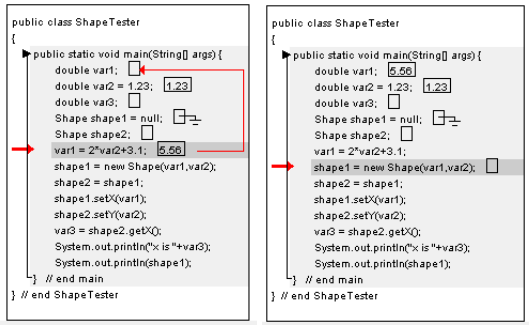

Figure 3: The two parts of the execution of an assignment statement.

The next statement is also an assignment, but it creates a **Shape** object using **new**. In the right half of Figure 3, a box is shown to hold the reference to the created object. The first step in executing this line of code is to create a new **Shape** object and start the execution of its constructor. This is shown in Figure 4. To save space, only the header of each method of a class is shown unless that method is being executed. Only the constructor is shown in its expanded form in Figure 4. An arrow from the statement in the main method to the parameters of the constructor in the **Shape** object indicates that the parameters must be moved from the main program into the local variables of the constructor. This is done during the next step of execution. The result is shown in Figure 5. After the two assignment statements of the **Shape** constructor are executed, the new shape is returned. Figure 6 shows the simulator just before this occurs. The arrow indicates where the flow of control returns. Figure 7 shows the situation after the constructor returns by falling off the end. The box at the end of the assignment statement contains a reference to the created object. An additional arrow shows that this reference will be moved to the **shape1** variable. JES moves the reference to the assigned variable in the next step as shown in Figure 8. If animation is used, this movement is animated. Now each of the methods of the **Shape** object is shown in its contracted form since none is active. At any time the user can click on one of the little triangles at the start of a method header to expand that method and show all of the code.

The next line of code shows a simple object assignment. After JES executes this line, both **shape1** and **shape2** reference the same object, as shown in Figure 9.

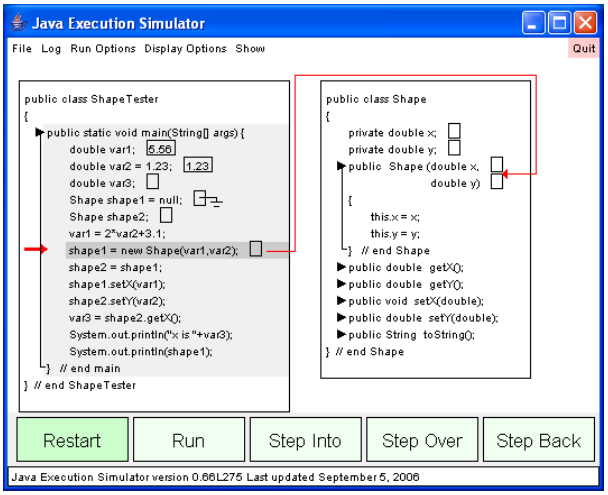

Figure 4: After an object has been created.

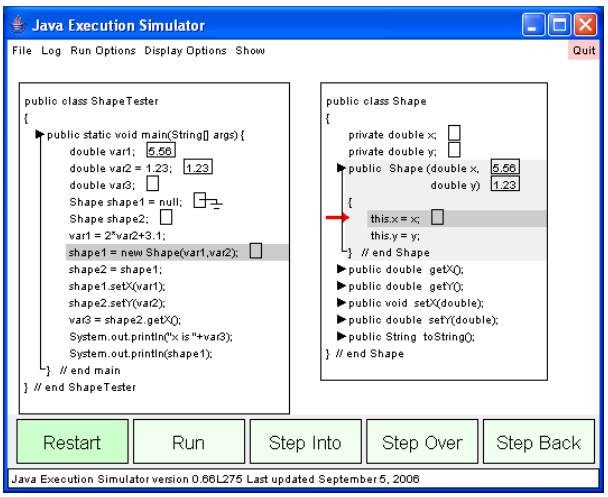

Figure 5: During execution of a constructor.

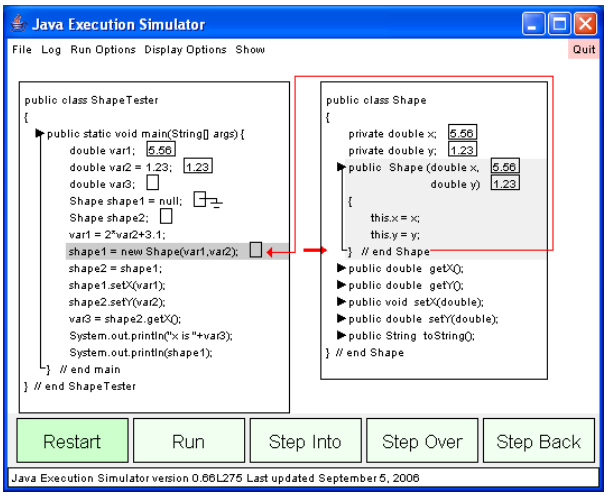

Figure 6: Before returning from a constructor.

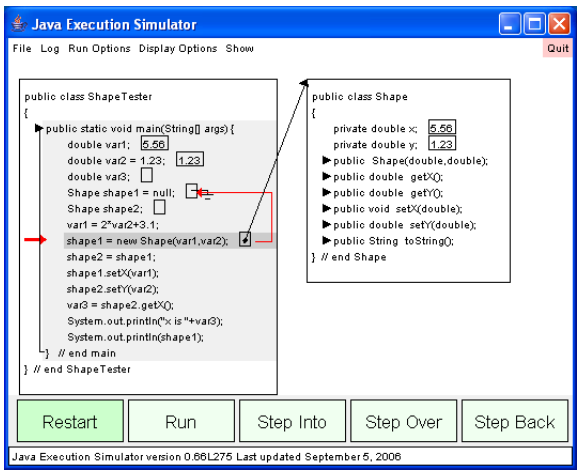

Figure 7: After the constructor has been executed.

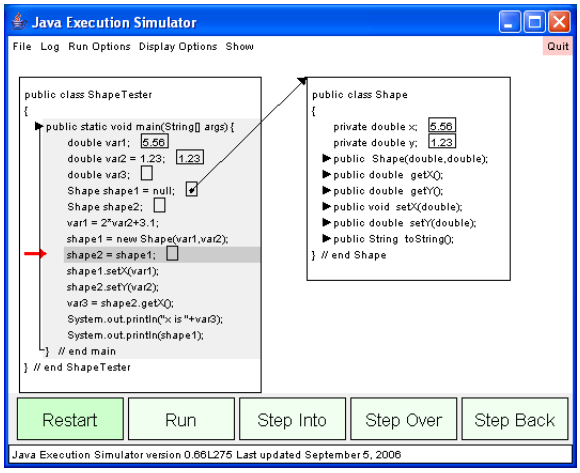

Figure 8: The object assignment statement is complete.

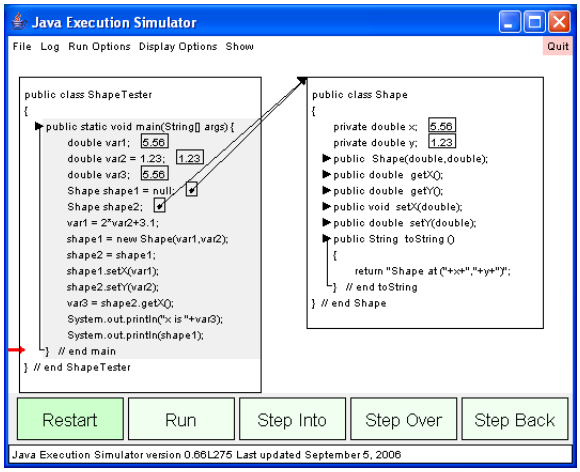

Figure 9: Two variables reference the same object.

The last two lines output the values of a variable and the description of the object. The **toString** method of the object, manually expanded and shown in Figure 9, produces the string used in the last statement. Figure 10 shows the simulator output window.

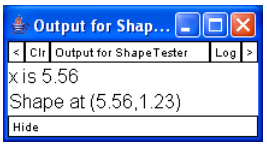

Figure 10: The Simulator output window.

3. PASSING PARAMETERS

One of the more difficult concepts in Java programming is the difference between passing primitive parameters and passing object parameters. The simulator provides a mechanism for illustrating this difference. In Example 2, the **Shape** object has two additional methods, a **badCall** method that attempts to change its double parameter, and a **setToMyPosition** method that sets the position of the **Shape** parameter to the position of this object. The simulator illustrates that the **badCall** method only changes the variable that is local to **badCall** and does not affect the variable in the calling method. Example 2 is not shown in this paper.

4. ARRAYS AND POLYMORPHISM

The simulator supports arrays of primitives and arrays of objects. Example 3 is shown in Figure 11. Some of the methods have been expanded to show the code. The program illustrates two methods of creating such an array, in the declaration with an explicit list of values, and with **new**. Students are often confused about the difference between arrays (which are objects) and array variables which hold references to arrays. The program has three array variables and three arrays. The first array is created using a list of 5 values in the declaration. The other two arrays are created using **new**. The third array is an array of **Shape** objects. Polymorphism is illustrated by two versions of the **setToMyPosition** method, each of which changes the state of its parameter to reflect the position of the shape. One takes an array of doubles as a parameter and sets the first two elements of the array to the position of the shape. The other method takes a **Shape** object as a parameter and changes its position. Figure 12 shows the output window after the program completes. The implied **toString** method for arrays used by the simulator gives useful information.

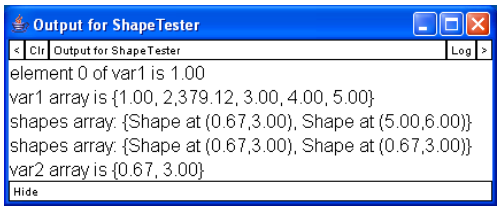

Figure 12: The output after Figure 11 completes.

5. INHERITANCE

The last example described here shows how the simulator handles inheritance. These diagrams tend to be quite large so only a simple example is shown here. The program creates an object of type **Square**, which extends **Rectangle** which in turn extends **Shape**. See Figure 13. Inheritance is shown in a UML-like way, with an arrow from a class to its base class. The simulator can illustrate how methods in the base class can be overridden and how **super** is used to access methods in the base class.

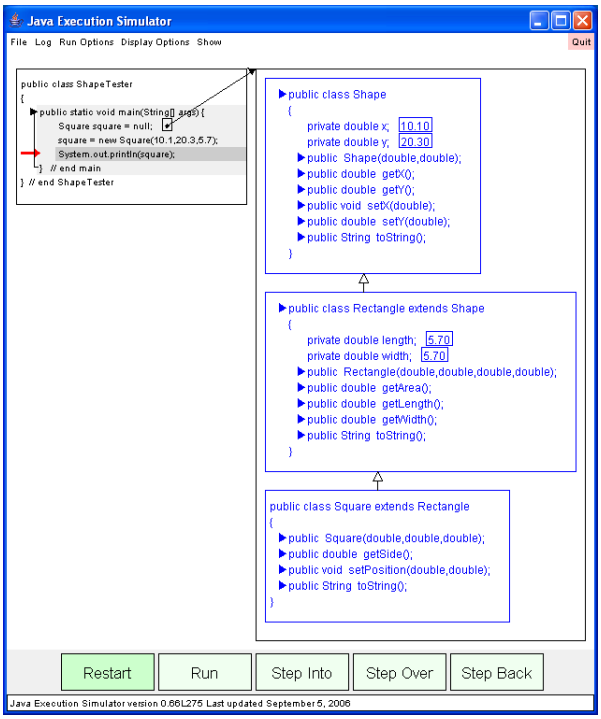

Figure 13: A program and object that uses inheritance.

6. USING THE SIMULATOR

All of the programs illustrated by the simulator are similar to ones I have traced in class on the board. Tracing on the board is difficult for the students to follow, because as the values of variables change, the old value is removed and replaced by the new value. For primitive variables this is not so bad since it is easy to keep a record of the previous values stored in a given location. For reference variables, this is more difficult. The values stored in reference variables are usually represented as arrows to the referenced object (as in the simulator) and it is difficult to store a history of these references. Often when doing a trace on the board a student will ask about a previous step and ask me to explain it again. It is difficult to reproduce the diagram from a few steps back so the previous step can be traced again. The simulator has a **Step Back** button that allows you to step back through the execution of the code.

Since tracing programs that are developed in class is a major use of the simulator, we provide a separate Java utility program to facilitate this. The utility program takes two directories as command line parameters or prompts for these directories if necessary. It takes all of the Java programs from the first directory, modifies them appropriately, and copies them into the second directory. The utility will find the file with the **main** method and create the appropriate configuration file for the simulator to use these Java programs. On some systems, it can also start the simulator. I have used this in class with programs created by JBuilder, but it should work with any development system as long as all of the source files are in a single directory. You can go from creating your project in a development system to running the same program in the simulator in less than 30 seconds.

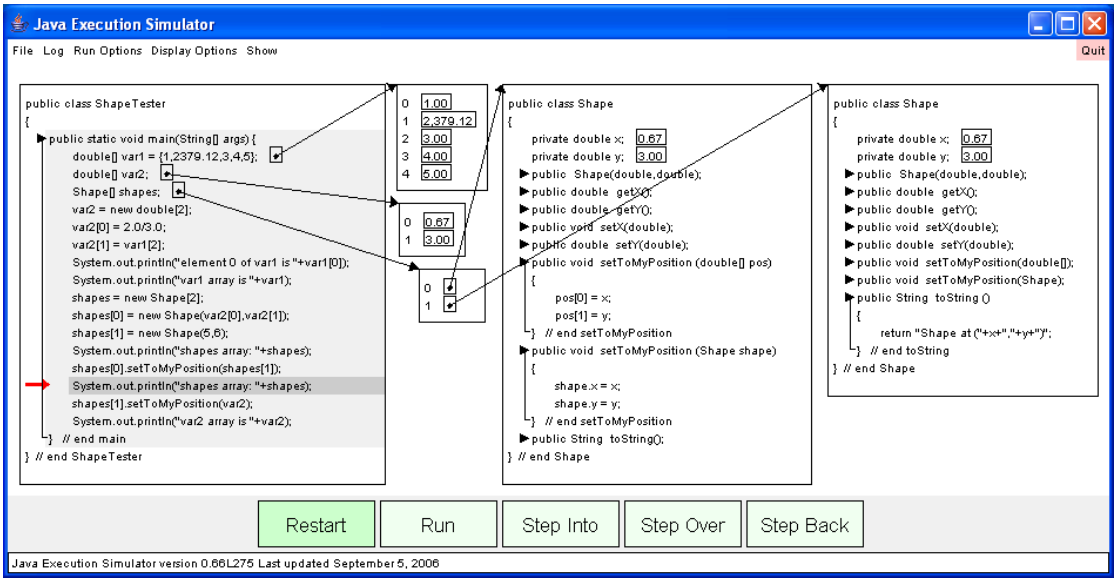

Figure 11: An example with arrays.

Another useful feature of the simulator is the ability to set breakpoints. The user sets a breakpoint by shift-clicking on a line of code. When the user pushes the **Run** button, JES steps through the code until the next breakpoint is reached. When a breakpoint is set, the simulator gives you the option to run the program at a faster speed so that you can get to the breakpoint in a second or two. This is useful if you want to start animating in the middle of the execution, or to demonstrate the action of a line of code near the end of the program. In single step mode the simulator allows both **Step Into** and **Step Over**, similar to many debuggers. The simulator also has support for static variables and methods which are not described in this paper.

7. LIMITATIONS

The original simulator supported the data types **int** and **boolean** but I found that they added no additional functionality. Expressions, either on the right side of an assignment, or within a **System.out.println** are evaluated atomically by the simulator. The simulator does not show the evaluation process. Also, these expressions cannot contain method calls. This is mainly to simplify the tracing of the program. A method call can occur on the right side of an assignment statement only when it stands alone. Without boolean variables or logical expressions, looping is not possible. While looping may be added in a future release, the simulator as it currently exists can handle a large body of programs that typically occur at the beginning of a CS 1 course. By the time students get to loops, they should start using the debugger that is built into most development systems.

The simulator supports variables of type **double** and object. The only objects supported are arrays of doubles, arrays of objects, and objects whose class is explicitly defined by the user. The simulator does not support boolean variables or conditional statements. It has very limited support for strings and it does not support any type of looping.

The simulator does not directly support any of the built-in Java classes. It does not support variables of type **String**, but it does allow **System.out.println** and **System.out.print**. Each of these takes a single parameter which is a sum of terms. Each term can be a string literal or the name of a variable. The values of variables of type **double** are always displayed and printed using exactly 2 decimal places. If a variable of type object has a **toString** defined, the return value of the **toString** is used. The simulator supports a limited **toString** method that contains a single return statement. The returned **String** has syntax restrictions similar to those of the parameter of **System.out.println**, the most severe of which is the inability to contain method calls. Since the simulator does not support variables of type **String** or general **String** expressions, this **toString** method cannot be called directly. However, it is flexible enough to be able to describe the state of an object when implicitly used in a print statement. In Java, the **toString** method for arrays is not very useful as it returns a string representing the serial number of the array object. The simulator replaces this with a list representing the values of the array. For arrays of objects, the **toString** of each element of the array is shown in the list. This is very useful in showing the current state of the array, especially since the simulator does not support loops.

JES is intended to be run on correct programs. If asked to simulate an incorrect program or one with an unsupported feature, the results are unpredictable and few diagnostics are available.

8. AVAILABILITY

The simulator is freely available. You can run the example code from a browser by pointing it to the simulator website [2]. You can also download a zip file containing the required jar files and the data files for the examples discussed in this paper. This allows the simulator to be run as an application so that you can provide your own program files. The web site contains a complete users guide for the simulator.

The simulator is an outgrowth of a set of simulators that I have designed for teaching operating systems [1]. All of the simulators are built upon a common library and have a similar user interface.

9. REFERENCES

[1] S. Robbins, Simulators for teaching operating systems, 2005. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/simulators**

[2] S. Robbins, A Java Execution Simulator, 2006. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/javasim**