Java Code Trace: A Constructivist Tool for CS Education

Tao Wang

Georgia Institute of Technology  
North Ave NW  
Atlanta, GA 30332  
+1 (973) 986-8243

twang@gatech.edu

**ABSTRACT**

Constructivism is a theory of learning which holds that students build their own understanding through experiences and reflection. It has been highly influential in the development of pedagogical practices but more can be done in terms of the creation of tools that support those practices. In this paper, I discuss why constructivist learning tools are important; survey those tools that are currently available for introductory computer science education; and introduce Java Code Trace, a tracing and debugging tool for learners of Java.

**CCS Concepts**

**• Applied Computing • Education • Computer-assisted instruction**

**Keywords**

education; tool; code; trace; constructivism

# INTRODUCTION

"Programming languages are the least usable, but most powerful human-computer interfaces ever invented" [1].

We can think of computing culture as split into two broad camps: users and programmers [2]. First-year computer science students largely come from user culture, where software is a tool that can be used to accomplish a specific task. These tools have well-defined interfaces and affordances that clearly define how subtasks are completed while also limiting the user from being able to do too much. To be successful in computer science, students must be acclimated to programmer culture, where software is open-ended, and where programmers accomplish tasks by communicating with software using an artificial language.

To be able to program well, students then must develop mental models of how the computer will interpret the instructions written in that language as well as how different sets and orderings of instructions will combine and interact with each other. Unfortunately, current methods in programming education do not emphasize the development of such a model [3].

This may be due to the legacy of a traditionalist practices in teaching that place emphasis on how to accomplish certain tasks as well as the amount of time it takes to prepare accurate illustrations of program execution. There is also little evidence of the development of debugging tools that are friendly to beginners. This may be because such a tool would be of little use to veteran programmers.

A beginner-friendly debugging tool could provide a lot of benefit to students and teachers. Students would be able to analyze execution traces of their programs even if their conception of how to do these traces by themselves with pen and paper may be limited at the beginning of their studies.

Teachers would also have access to a simple way to demonstrate program execution in class, which would allow for accurate on-the-fly modifications to respond to student questions.

# CONSTRUCTIVISM AND CS EDUCATION

Ben-Ari [3] contrasts a classical education paradigm with a constructivist one in the following way:

* Within the classical education paradigm, there is one absolute truth that can be discovered through empirical observations and logical deduction. That truth can be transmitted from person to person through lectures and writing.
* Within a constructivist paradigm, "absolute truth is unattainable." Instead, individuals create their understanding by combining new observations with personal experience and prior knowledge. Knowledge must be obtained recursively with assistance and feedback.

Ben-Ari's contends that "a (beginning) CS student has no effective model of a computer." Unlike new physics students who begin a course with some model of how the world works (even one which may be flawed, new CS students' model of the inner workings of a computer "At most… is limited to the grossly anthropomorphic 'giant brain'... [or] 'superbug'... the idea that a 'hidden mind' within the programming language has intelligence."

Therefore, much of CS teaching in the beginning should focus on helping students develop a better model of the computer and of programming languages.

"The relevance for [CS education] is that courses… must explicitly address the construction of a model, and not limit themselves to… practices of the form 'to do X, follow… these steps."

Meerbaum-Salant, et al. [4] created a middle school curriculum around the Scratch language focusing on CS concepts rather than features of the language. Their conclusion was that it was possible to teach broad CS concepts in an age appropriate manner. However, they found that students had the most problems with three concepts: initialization, variables and concurrency. In their words, “these concepts are more abstract than the other concepts that we investigated. One possible explanation is that these three concepts involve several Scratch instructions and their relationships, whereas the others can be expressed in Scratch by one instruction."

I hypothesize that this difference in outcomes is a result of the flawed or complete lack of a model that new students have of the computer. Initialization, variables, and concurrency are things that happen "behind the scenes": their implementation is abstracted from the programmer and controlled by the operating system and the cpu. However, without a proper model of how this is done, this layer of abstraction impedes learning in an amount proportional to how much it aids development.

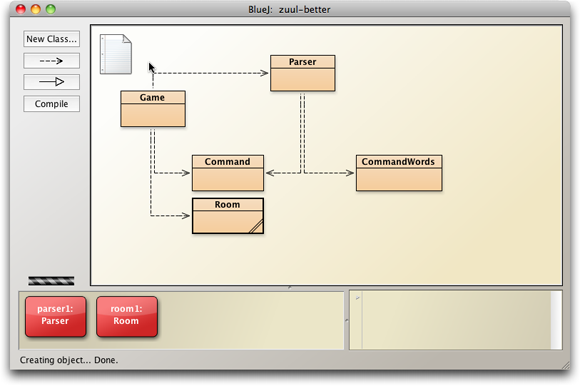
# CONSTRUCTIVIST TOOLS

There currently exist a number of tools that aid beginning students by making explicit the model under which programming languages work. In this way, these tools are constructivist tools. They provide an additional experience which students can use to form a better understanding of the underlying principles.

## BlueJ

Kolling, et. al. [4] developed the BlueJ IDE to teach object orientation. "Our hypothesis", they write, "is that teaching object orientation is not intrinsically more complex, but that it is made more complicated by a profound lack of appropriate tools and pedagogical experience with this paradigm."

**Figure 1. The BlueJ interface.**



In BlueJ, a Java project is presented not as a series of files, but as a UML diagram. Users can create instances of objects that are placed on the "workbench" at the bottom of the interface or modify each class by double-clicking and editing the source code line-by-line. Objects on the workbench can be manipulated via context menus that provide access to each object's public interface. If a method is called that requires a parameter, then a pop-up window asks for the appropriate parameters. "The BlueJ environment… allows reversal of the order of introduction. Interaction with objects can be presented first," before "the main method, array parameters, object creation, variable declaration and dot notation."

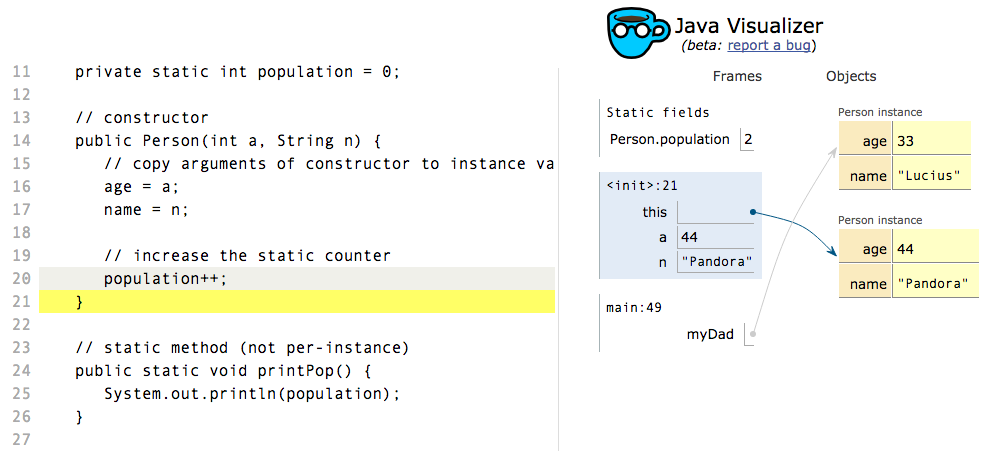
BlueJ’s interface gives explicit insight into the model of object orientation. Through this visualization, students can

## Online Python Tutor and JavaVisualizer

Online Python Tutor is a web-based program visualization tool [6]. Its author, Philip Guo, was motivated to create it by his experience teaching Python and "drawing messy stack and heap diagrams on the whiteboard" [6]. The resulting web application allows users to input Python code through the HTML front-end. An HTTP GET request is made, and the back-end produces an *execution trace* in JSON format which is returned to the front-end. The front-end then interprets the JSON trace, allowing a user to (a) step through the execution of the code one line at a time and (b) see a visual diagram of existing constants, variables, and object data.

JavaVisualizer [7] uses the PythonTutor front-end and replaces the back-end with one that traces Java programs.

**Figure 2. JavaVisualizer.**

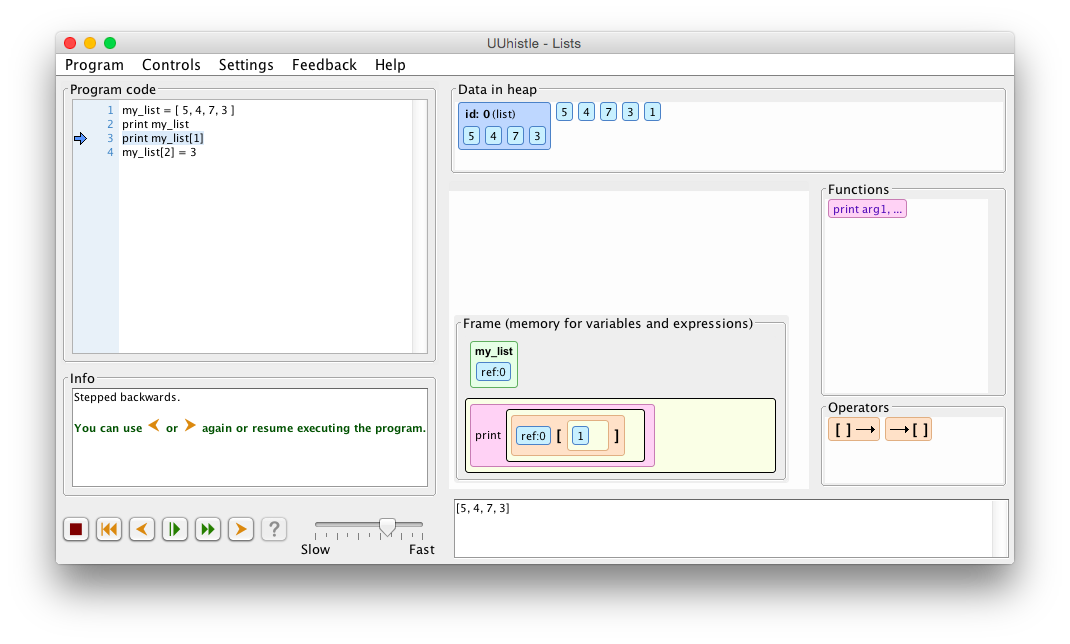


Python Tutor and JavaVisualizer [7] address one of the issues that Meerbaum-Salant, et. al., faced in their Scratch study by visualizing the state of variables and highlighting the order of operations within a program.

## UUhistle

UUhistle (pronounced "whistle") is a desktop application that provides animated program traces for Python programs [8]. UUhistle has similar functionality to Online Python Tutor and Java Visualizer but also provides details of expression evaluation. A user can step through a statement such as print my\_list[1] and see how the reference my\_list[1] is reduced to the value contained at index 1 of my\_list before being passed to the print function.

**Figure 3. UUhistle.**

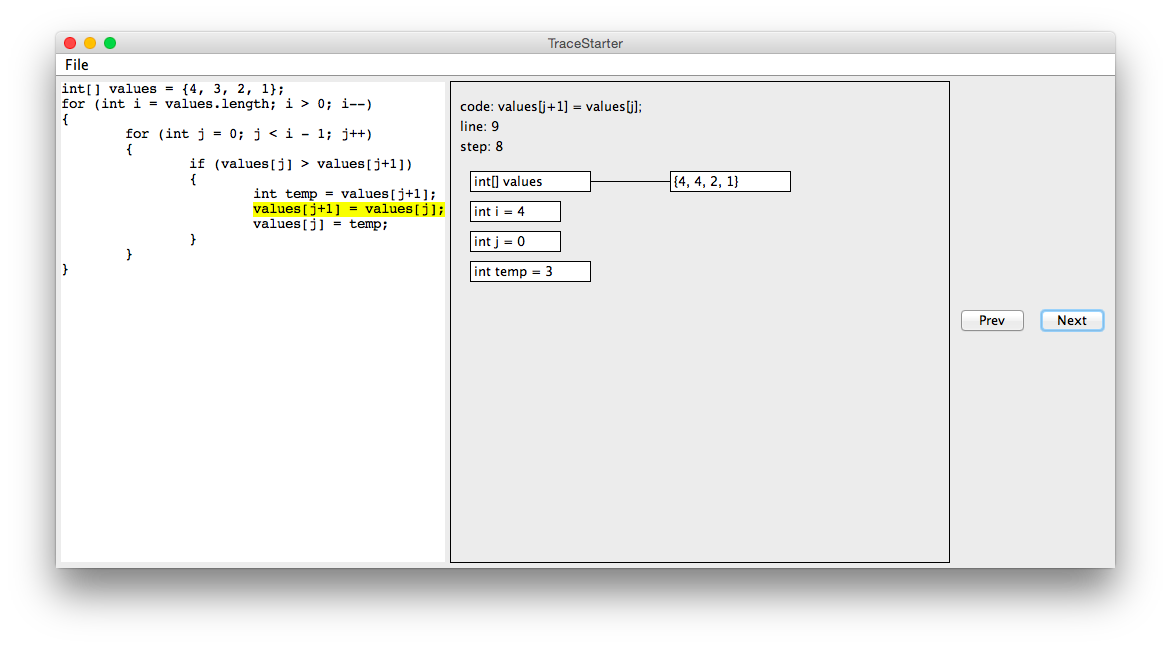


The heap trace is not the only novel feature of UUhistle. The program also includes interact “quizzes”. In each quiz, a program is loaded in the left pane, and the user is asked to perform each step the trace, with UUhistle verifying and providing hints along the way. Though the interface is unresponsive sometimes, the idea of built-in assessments offers even more feedback for a new learner.

# JAVA CODE TRACE

The goal of Java Code Trace was to build a constructivist tool along the lines of Online Python Tutor, JavaVisualizer, and UUhistle. Code tracing is a fundamental skill required of beginning computer science students but can be tedious and error-prone [6]. Tools that can automatically show execution traces are much more convenient. In addition, these tools can help students build up a mental model of code execution through its choice of visualization.

**Figure 4. Java Code Trace.**



The features of Java Code Trace include:

* *A scripting engine for Java*. Java is one of the most popular programming languages in use today [9]. It is the language of much academic research as well as that of the Advanced Placement curriculum. Java Code Trace uses the BeanShell [10] package to execute Java statements at runtime. This allows Java code to be written outside of defined classes. The main benefit of this approach is the ability to rapidly test code without having to go through the overhead of setting up and compiling class files.
* *Visualization of primitive variables, Strings, and arrays*. Taking the cue from Python Tutor and JavaVisualizer, variable state is represented visually. This is a subtle difference from how debuggers display variables in tables. However, it is an important difference for new learners, especially with regards to learning about object references.
* *Statement-by-statement code highlighting*. Statements are highlighted as a user steps through the execution of a code segment. In addition, with lines like for-loop headers in which only parts of the statement are relevant at any given moment, the most relevant portion of the line is highlighted.

## Design

The Java Code Trace project is built around three classes: TraceStarter, Trace, and Snapshot.

### TraceStarter

*TraceStarter* is the front-end of the program. It creates the user interface, reads and writes to disk, and handles the display of snapshots.

### Trace

*Trace* is the main scripting engine. It is initialized with a file that contains a Java code segment, which it immediately breaks up into discrete statements.

*Trace* reads through the file, executing each line of code on initialization. It maintains the following structures while doing so:

* *A stack of frames*. Each frame is tied to the most recent scope, and variables that are initialized within that scope are stored in each frame.
* *A program counter*. The PC points to the current line of execution.
* *A list of snapshots*. After each line or segment is executed using BeanShell, a snapshot of the current frame is taken and added to a list.

### Snapshot

A *Snapshot* is a structure that contains:

* *The program counter*. The number of the line of code that was executed.
* *The current line or code segment*. The actual Java code that was just executed.
* *A step counter*. A number used to sort related Snapshots.
* *A dictionary of variable names*. Variable names are stored along with their values, their types, and their hash codes.

A separate Snapshot is created after each line of code is executed. All Snapshots are stored in a list within an instance of Trace. TraceStarter takes these Snapshots and displays them.

## Planned Work

Java Code Trace is a work in progress. At the time of writing, these are the main areas in which the project could be improved.

* *Handling exceptions and improper syntax*. Lines that are not proper Java are simply ignored at this point.
* *Infinite loops*. The entire execution trace is built when a file is saved or loaded. If some program hangs in an infinite loop, JCT will hang as well. This could be solved by putting a maximum on the number of steps of execution that can be evaluated.
* *Importing non-standard classes*. The BeanShell library is a limited in its access to Java classes outside of the standard library.
* *Improved interface*. Further research can be done on how best to visualize execution traces.

# DISCUSSION

In its present form, Java Code Trace offers one thing that JavaVisualizer does not: a more lightweight scripting environment that may be better for rapidly testing code. However, it is a little behind the tools mentioned earlier in this paper in terms of polish and usability.

## Feedback

I submitted this project to ten current AP Computer Science students for their feedback. The ten students then discussed their first attempts at using Java Code Trace to explore some commonly used algorithms. Feedback from this discussion was moderately positive.

Students generally liked the code highlighting indicating which line was being executed. The visualizations were deemed useful though “very plain and basic.” Stepping through algorithms that involved nested loops was somewhat of a pain, and it was deemed difficult to spot errors. An option to “play” the algorithm was requested so that code that required more than a hundred steps of execution could be viewed without excessive clicking.

## Future Study

Since the main goal of Java Code Trace is to improve students’ mental models of computation, the most obvious question is on how successful it would be at doing so. Related to that question is the problem of how to measure students’ conceptual understanding in computer science education. Performance on standardized tests is one metric, but there are likely to be many others that are better. This problem should be explored first before a full analysis of this project’s efficacy.

For the immediate future, I plan to incorporate use of Java Code Trace into the AP Computer Science curriculum. This will provide at the very least first-hand anecdotal evidence of its usefulness.

## Conclusion

I believe Java Code Trace is a good example of the types of tools that are needed for the future of computer science education. Returning to the idea of user and programmer culture [2], tools designed for beginners should acknowledge that new students come from user culture and that effort should be made to make the transition to programmer culture easier.

This transition happens once students are able to develop an accurate mental model of computation and become comfortable enough with that model in order to solve problems. Constructivism informs us that students will only achieve this learning through self-experience and reflection. So the tools that students use must give enough valuable feedback so that this kind of learning can take place.

# ACKNOWLEDGMENTS

My thanks to Bobbie Eicher for mentoring this project and the AP Computer Science students at The Chapin School for their feedback.

# REFERENCES

1. Ko, Andy. "Programming languages are the least usable, but most powerful human-computer interfaces ever invented". 25 March 2014. <http://blogs.uw.edu/ajko/2014/03/25/programming-languages-are-the-least-usable-but-most-powerful-human-computer-interfaces-ever-invented/>
2. Guo, Philip. "The Two Cultures of Computing". December 2013. <http://www.pgbovine.net/two-cultures-of-computing.htm>.
3. Ben-Ari, Mordechai. "Constructivism in Computer Science Education". 1998.
4. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M. "Learning Computer Science Concepts with Scratch". 2013.
5. Kolling, M., Quig, B., Patterson, A., Rosenberg, J. "The BlueJ System and its Pedagogy". 2003.
6. Guo, Philip. "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education". *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 2013.
7. Gwozdz, Will & David Pritchard. Java Visualizer. Accessed 21 February 2016. <http://cscircles.cemc.uwaterloo.ca/java_visualize/#>.
8. J. Sorva and T. Sirkiä. UUhistle: a software tool for visual program simulation. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling ’10, pages 49–54, New York, NY, USA, 2010. ACM.
9. “Most Popular Coding Languages of 2015.” <http://blog.codeeval.com/codeevalblog/2015#.VyX8zqMrL_Q>=
10. BeanShell. [http://www.beanshell.org/](http://www.beanshell.org/docs.html).