A Lightweight Visualizer for Java

John Hamer
Department of Computer Science
University of Auckland
New Zealand

J.Hamer@cs.auckland.ac.nz

1 Introduction

We report on a lightweight tool (LJV) that provides high-quality visualizations of Java data structures. The key distinguishing characteristic of the tool is its *effortlessness*. For instructors, LJV can be easily adopted without requiring any change to the teaching environment; i.e., you can keep the same editor, Java environment, code examples, etc. For users, the interface to LJV is truly simple: a call to a single (static) method that takes a single (Object) argument.

LJV works by using Java reflection to traverse the fields of an object (and the fields of the fields, etc.), generating a textual description of the connectivity as it goes. The description is then passed to the graph drawing program GraphViz (North and Koutsofios, 1994)[1]. GraphViz automatically produces an image of the graph, in a choice of bitmap or vector formats.

Each visualizations depicts a single Java object. A sequence of visualizations constitutes an animation that the user can step through, forwards and backwards, at their leisure.

LJV includes a configuration mechanism that allows broad and precise elision of detail. Users can choose to hide certain fields or groups of fields, or show entire objects in summary form.

This paper discusses the design of LJV and its intended learning objectives. While no formal evaluation of the tool has been undertaken, some classroom experiences are reported. A conclusion reflects on tools and pedagogy.

2 An illustration, and commentary

Examples of the visualizations produced by LJV are shown in Figure 1. The first diagram shows a binary (red-black) tree containing four entries. The second is a circular doubly-linked list with three entries. Both examples happen to be from the standard Java library, but this is not necessary; LJV has no built-in knowledge of any particular data structures, and will work just as well with data structures written by instructor or student.

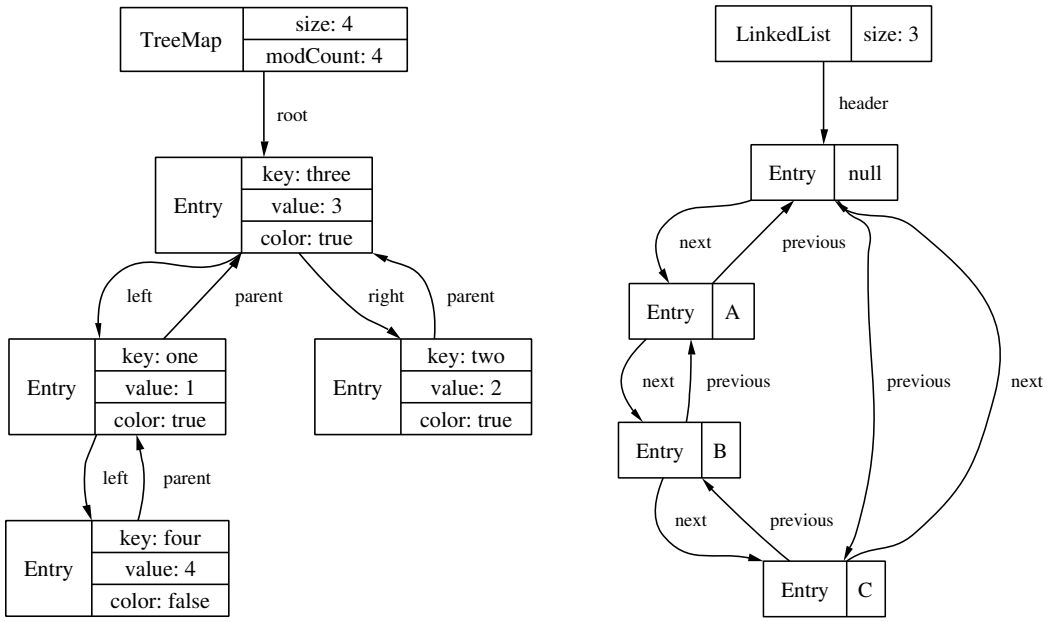
Two further observations are in order:

æsthetics Despite the logical symmetry of the data structures, the diagrams exhibit a certain amount of arbitrary variation:

- the entries in the linked list are not evenly spaced;
- some edges are straight while others are curved;
- the curved edges "wobble" in some places.

We feel that the *organic* nature of the diagrams makes them more interesting, and therefore likely to retain a viewer's attention longer than would a rigid layout. This consideration was one of the factors in choosing GraphViz as the drawing tool.

[1] GraphViz is a free, open-source program, available for all major platforms from `www.research.att.com/sw/tools/graphviz`



(a) A binary tree

(b) A linked list

Figure 1: Some example visualizations

shallow semantics The left subtrees appear to the left in Figure 1(a) by good luck. They could just as easily have appeared on the right. This "luck" ran out in the linked list diagram, where the list elements appear in an unconventional anti-clockwise order.

Our visualizer does not provide any means of hinting at a deeper semantic meaning in the data structures depicted. At times, this can lead to momentary confusion by the reader. However, the tradeoff is one of greatly simplified usage. LJV is able to draw objects without any prior knowledge of their data type, and it generally does a good job.

3 Design

In designing LJV, we have adopted the following principles:

- students must be engaged in active learning (Hundhausen and Douglas, 2000);
- the tool must be simple to use. As a guideline, setting the Java CLASSPATH has proven to be "too hard" for a significant number of students;
- avoid unnecessary features that could distract students from substantive course material (Naps, 1998);
- minimise the effort required by instructors to integrate tools into the curriculum (Naps et al., 2003);
- software must be reliable.

LJV has a number of features that distinguish it from more traditional "heavy-weight" visualizers:

- Setup is straightforward. Once an administrator has installed the GraphViz "dot" utility, a user needs only to copy a single, small (600-line) Java source file into her working directory to be immediately able to generate visualizations.

- The tool is easy to use. A user inserts calls to a static method, passing the object he wishes to display. As these methods are executed, a numbered sequence of pictures is written to disk. He can then view the animation frame-by-frame using any standard picture viewer.
- The visualizer works on any Java program. No specific programming conventions need to be followed.
- Active learning is supported, as the user must decide which objects to display, where to place the calls to draw these objects, and what to elide.
- "Wrong" data structures can be viewed (as well as correct ones). The visualizations faithfully depict actual code behavior.
- Feedback is *not* immediate. The visualizations are not seen until the program completes.

  We believe that a delay between "doing" and "seeing the result" can be important, as it provides a time for anticipation and reflection on the expected outcome[2].
- Visualizations can be easily incorporated in reports, www pages, and presentations.

4 Configuration

A *drawing context* is used to control the appearance of graph nodes and edges, to elide classes and fields, and to control the format and naming of output files.

A default drawing context can be configured to give reasonable defaults, such as ignoring private fields or treating all system classes as primitive types.

Configuration of the drawing context is ongoing. The operations currently supported include:

treatAsPrimitive The specified class is treated as a primitive value; i.e., the result of calling **toString** on the object is displayed in-line, rather than showing the object as a separate node.

This is an effective mechanism for reducing the amount of clutter in a visualisation. Most Java classes provide a **toString** method that conveys the content of the object as a string. This includes all the Java collection types (lists, maps, etc.). Ellipsis ("...") may be inserted to replace the middle of excessively long strings.

ignoreField Suppresses display of the given field; i.e., LJV pretends the field does not exist.

ignoreClass, ignorePackage Suppresses display of any field of a given class, or from a given package.

ignorePrivateFields A boolean value. If set (the default), fields that are not normally accessible are not displayed. This includes private and protected fields in other classes, and package-visible fields from other packages.

setClassAttribute, setFieldAttribute Attributes include border and font colour, font size, fill style, etc. For example, a binary tree node can be made bright pink and different colours given to the left and right links as follows:

```
Context ctx = getDefaultContext( );
ctx.setClassAttribute( Node.class, "color=pink" );
ctx.setFieldAttribute( "left", "color=red" );
ctx.setFieldAttribute( "right", "color=blue" );
```

[2] In support, we note anecdotally that a similar sentiment can be heard from "old timers," who often claim that the delays inherent in punch-card programming led to more a more thoughtful coding process than that observed in the instantaneous environments of today.

The available attributes are determined by the GraphViz tool.

Fields can be set by name (as above) or by specific reference.

showFieldNamesInLabels Determines whether field names should be displayed in nodes or not.

qualifyNestedClassNames Nested classes are given compound names by the Java compiler; e.g., a class **Entry** nested in a class **LinkedList** will be named **LinkedList$Entry**. Normally, only the last part of these names is displayed. Setting this option results in the full name being used.

outputFormat This string field determines the output format. The default is `png`, a widely used image format. The `ps` (encapsulated postscript) format can also be used for generating high-resolution scalable graphs suitable for including in reports.

baseFileName A numeric suffix is added to give a unique name for each output graph (e.g., `graph-0.png`, `graph-1.png`, etc.) The set of graphs so generated can then be viewed as a slide show using any standard image viewer.

5 Implementations

In addition to a static Java method, LJV has been integrated into a modified debugger (Fiedler, 2004) and a Java interpreter (Niemeyer, 2004). These integrated environments provide immediate display of the visualization, and may prove useful in some learning contexts.

We note that these variations sacrifice some of our design principles. Setup is not as simple, and various constraints are imposed by the environment. We feel it is best to regard these versions as "experimental."

5.1 BeanShell integration

The BeanShell is a Java interpreter[3]. As well as being used as a stand-alone interpreter, the BeanShell has also been embedded in several popular Java development tools, including GNU Emacs (Kinnucan, 2004), Sun Forte (Microsystems, 2004), BEA WebLogic (Inc., 2004), and Apache Ant (Foundation, 2004). More information on the BeanShell can be found at `www.beanshell.org`.

One of our students (McCall, 2003) integrated LJV into the BeanShell, providing a **display** command that opens a window containing the visualization of an object. A screen dump of the system in action is shown in Figure 2. The white-on-black window in the bottom right is the BeanShell console, in which the user has typed a number of Java statements. Two **display** commands have been issued, resulting in two image frames (one for a linked list, and the other for an array list). The frames are sized to fit the image, and are refreshed after each console command.

5.2 JSwat integration

JSwat is a stand-alone, graphical Java debugger written by Nathan Fiedler (Fiedler, 2004).

We have developed an experimental integration of LJV into JSwat. The interface is through a "graph" menu command that is available when browsing values in a stack frame. Images are refreshed each time the debugger steps or stops at a breakpoint. The integration uses a viewer called Grappa (Mocenigo, 2004), which allows the visualization to be scrolled and zoomed.

[3]Technically, a Java-compatible scripting language. The language is an extended version of Java that supports dynamic typing.

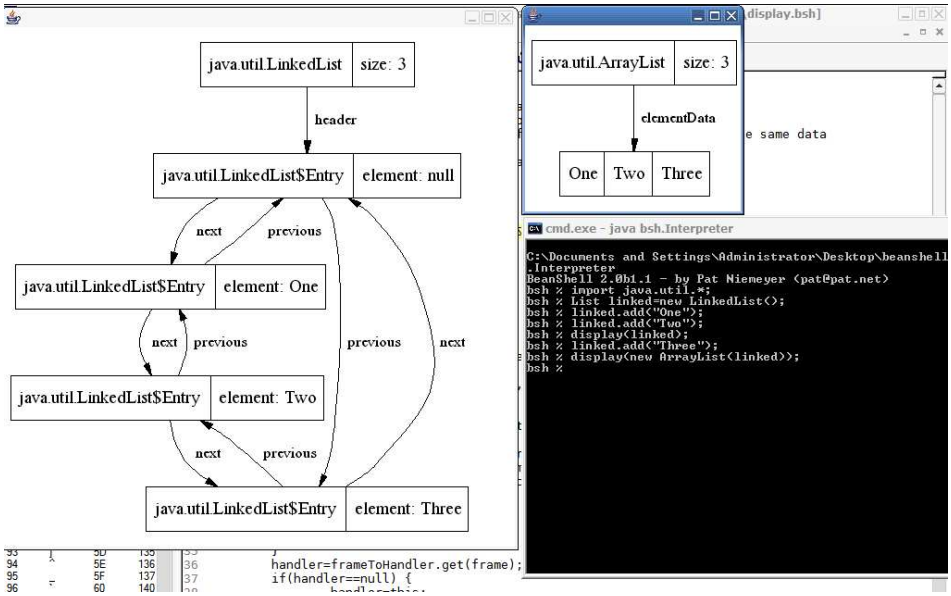


Figure 2: A session using the LJV-BeanShell integration

A screen dump is shown in Figure 3. The stack frame appears on the top left of the window, and a single Graph Viewer window holds all the images. Not shown is a configuration screen that provides access to the LJV elision settings.

Work is in progress in extending the debugger to support to direct manipulation of the graph. This includes updating data values, eliding or expanding nodes, manual adjustment to node positions, setting colour and other node and edge attributes, etc.

6 Evaluation

LJV has been deployed in three classes to date. Two of these courses were CS2-level data structures, and the other a CS1-level introductory programming course. No formal evaluations have yet been undertaken, but some anecdotal evidence has been gathered.

The tool has been used in two educational settings. First, a structured laboratory session was used to both introduce the tool and to identify some common misconceptions. Later, students were left to use or ignore the tool as they saw fit during the remaining course-work.

The laboratory presented a series of Java code fragments, for which students were asked:

> *. . . make a sketch in your engineering notebook of the graph you expect to see* before *you run the program. If you can think of more than one plausible output, sketch them all.*
>
> *If the output differs from your expectation, write down a concise summary of the difference and think about why your prediction was wrong.*

The code fragments explored various **String**s, **StringBuffer**s, one and two-dimensional arrays, and parameter passing. For example,

```
String x = "Hello";
String y = new String(x);
Dot.drawGraph( new Object[]{ x, y } );
```

Feedback from students has been positive, and no problems with the use of the system were reported. A number of students made regular use of the tool later in the course, with one even undertaking a significant extension (see Section 5.1). We intend to survey students in subsequent courses to determine whether they continue using the tool.

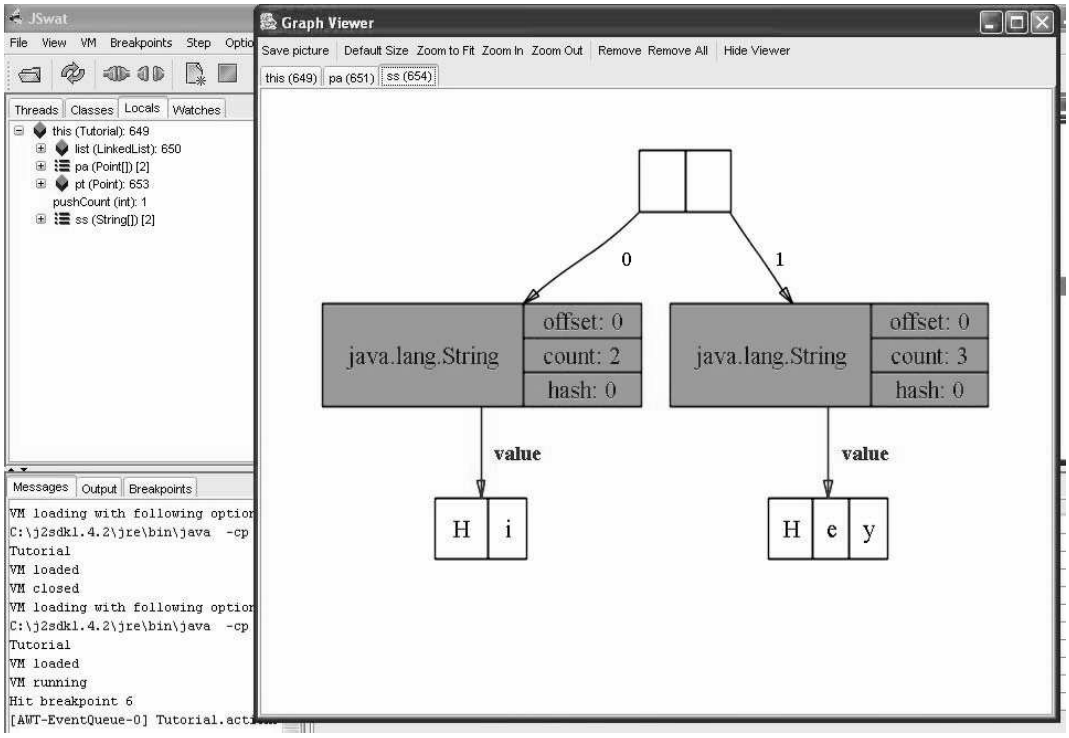


Figure 3: Exploring with the LJV-JSwat integration

7 Misconceptions

The primary benefit of the tool has been in overcoming misconceptions concerning the Java data model. We are compiling an "inventory" of common mistakes held by novices. Such misconceptions are often stubbornly persistent, and are regularly observed in students in advanced courses (Holland et al., 1997). The "inventory" includes the following:

Reference semantics Java assigns object types by reference, but primitive types by value. Many students fail to notice there is a difference, and even when told often fail to register the significance.

Primitive string Strings in Java are objects, but string constants *look* like primitive values. The "primitive string" misconception is the assumption that strings have value semantics.

Object arrays Object arrays hold references, not values. Students sometimes assume that because arrays of primitive have value semantics, all arrays do so.

2D arrays 2-dimensional arrays are constructed from 1-dimensional arrays. Arrays thus have several uninitialised states: a null reference, or a reference to a null array block. Arrays can also be jagged, so the second dimension must be determined on a row-by-row basis. These subtleties are often missed.

Static field Static fields are not part of any object, yet they share the same scope rules as object fields. This manifests in students sprinkling the `static` keyword through their code seemingly at random. The misconception is reinforced when programs compile and "appear" to work.

Inheritance A broad category that needs further refinement. The central issue is that inheritance means objects are often not the same as their declared types. The misconception leads students to use concrete types (**LinkedList**, **Vector**, etc.) in preference to an interface type (say, **List**).

Identity/attribute confusion This is another misconception category, from (Holland et al., 1997). It manifests in various false beliefs, such as "only one variable can reference to a given object at a given time" and "if you have two different variables, they must refer to two different objects."

LJV proved helpful in alerting many students to the fact that they held misconceptions. By turning off all elision, students are presented with visualizations that accurately reflect the Java data model, including deep sharing.

8 Related work

Pedagogically, Naps's `Visualizer` class (Naps, 1998) shares much in common with the work presented here, with both approaches emphasising the need for a tool simple enough for students to use without unnecessary distraction. Our visualizer is more general, does not require any special configuration to handle new data structures, and is easier to use. Naps's approach supports abstract presentations of data (e.g., bar charts for integer arrays) and supports arbitrary customisation.

Another approach to supporting abstract presentations was taken by Korn and Appel (1998), who developed a tool for specifying many kinds of visualizations of Java programs in a declarative pattern-action notation. For example, the tool can produce a visualization for a parse tree that uses bitmap images for the internal nodes (e.g., a large "plus" symbol for an addition node). The notation is expressive enough to construct quite sophisticated diagrams. However, it does not resemble Java code, and would require a considerable effort for students to master. To hide the notation from novice users, a "visual pattern constructor" was provided. The authors report some success in using the system in a compiler course.

JavaViz (Oechsle and Schmitt, 2002) can display UML object and sequence diagrams of running Java programs. The object diagram layout is simplistic, and looks similar to DDD. Some support is provided for visualizing concurrent threads in the sequence diagrams. No elision support is provided.

DDD is a widely used graphical debugger (Zeller and Lütkehaus, 1996) that runs on Unix platforms. The system supports a notation for specifying how to display the contents of a node, including nested layout of aggregate structures. References are expanded manually, and the system supports a simplistic automatic layout that will redisplay the current graph. Shared references are not recognized, which can lead to diagrams in which a single object appears in multiple places.

9 Conclusions and future research

Effort has been identified as a major impediment to the uptake of visualization tools for teaching. Many tools come attached to large or hermetic environments, and cannot be easily incorporated into an existing teaching setting. LJV's primary distinction is the very low effort required to install and run, and its ability to integrate easily into virtually any Java environment.

Most of the tools that operate on arbitrary (student) Java programs provide limited support for controlling the amount of detail displayed. The elision controls provided by LJV offer a powerful solution to this problem, and should prove adequate for most introductory courses.

The usefulness or otherwise of a visualization tool is ultimately determined by how widely it is adopted. Seeing students continue to use LJV after the end of formal instruction is evidence that the tool has real educational value. However, more formal research is needed to drive further development. For example, we have no evidence as to whether the delayed feedback model contributes to improved learning over a debugger model that provides instant feedback. A formal study comparing plain LJV to the JSwat integration may shed some light on this question. We also note a perceived demand from educators for tools that can provide

abstract presentations of data structures. It remains to be seen whether LJV can be extended to support the necessary transformations without sacrificing its essential simplicity.

References

Nathan Fiedler. JSwat, a Java debugger, April 2004. URL `www.bluemarsh.com`.

Apache Software Foundation. Apache Ant project, August 2004. URL `ant.apache.org`.

Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. In *SIGCSE'97 Twenty-Eighth Technical Symposium on Computer Science Education*, pages 131–134, San Jose, 1997. ACM Press. ISBN 0-89791-889-4.

Christopher Hundhausen and Sarah Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *IEEE International Symposium on Visual Languages*, pages 21–30, September 2000.

BEA Systems Inc. BEA WebLogic platform, August 2004. URL `www.bea.com`.

Paul Kinnucan. Java development environment for Emacs (JDEE), August 2004. URL `jdee.sunsite.dk/`.

Jeffrey L. Korn and Andrew W. Appel. Traversal-based visualization of data structures. In *Proceedings IEEE Symposium on Information Visualization*, pages 11–18. IEEE Computer Society, 1998. ISBN 0-8186-9093-3.

Sam McCall. private communication, October 2003.

Sun Microsystems. Sun Java studio, August 2004. URL `www.sun.com/forte/`.

John Mocenigo. Grappa: A Java graph package, August 2004. URL `www.research.att.com/~john/Grappa`.

Thomas Naps. A Java visualizer class: Incorporating algorithm visualizations into students' programs. In *ITiCSE'98 Innovation and Technology in Computer Science Education*, pages 181–184, Dublin, Ireland, August 1998.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152, 2003. ISSN 0097-8418.

Pat Niemeyer. BeanShell: Lightweight scripting for Java, April 2004. URL `www.beanshell.org`.

Stephen C. North and Eleftherios Koutsofios. Application of graph visualization. In *GI'94 Graphics Interface*, pages 235–245, Banff, Alberta, Canada, 1994. URL `citeseer.nj.nec.com/221206.html`.

Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In S. Diehl, editor, *Revised Lectures on Software Visualization, International Seminar*, volume 2269 of *LNCS*, pages 176–190. Springer-Verlag, 2002. ISBN 3-540-43323-6.

Andreas Zeller and Dorothea Lütkehaus. DDD — a free graphical frontend for Unix debuggers. *SIGPLAN Notices*, 31(1):22–27, January 1996.