# Visualising Java Data Structures as Graphs

**John Hamer**

Department of Computer Science
University of Auckland
Private Bag 92019, Auckland New Zealand
Email: *J.Hamer@cs.auckland.ac.nz*

## Abstract

We present a simple, general-purpose tool for visualising Java data structures. The tool uses Java reflection and an open-source graph drawing program to produce text-book quality depictions of arbitrary Java objects.

The tool offers certain pedagogical advantages over other "heavy-weight" visualisation systems. Its simplicity and generality means that students are able to visualise their own data structures, rather than passively observing a limited range of "correct" visualisations prepared in advance by the lecturer.

The tool supports an active, exploratory style of learning, and is ideally suited for use in CS1-level courses that introduce Java references and arrays, as well as a range of CS2-level data structure material. Initial classroom results are encouraging.

## 1 Origins

Our visualisation tool was originally conceived as an aid to students learning about linked lists. It started one evening, when the author wrote a "pretty-printer" for linked lists that generated a description suitable for input into the GraphViz graph drawing utility (see figure 1). GraphViz notation is quite simple, essentially requiring just the graph edges to be listed. Node and edge annotations are given in square brackets. The graph generated from this description is shown in figure 2. A variety of output formats are supported by GraphViz, including the widely understood "PNG" (for on-screen viewing) and encapsulated postscript (for printed documents).

The linked list "pretty-printer" turned out to be of limited use, however. Students could only generate pictures of "correct" linked lists from code supplied to them by the lecturer. They were not able to apply the technique to their own (often buggy) linked list code, or to view the state of any other data structures. Furthermore, code for generating GraphViz output was included inside the linked list code, and this tended to obscure the more important parts of the code.

All of these limitations were eliminated with the use of Java *reflection*. Unlike many "statically linked" languages like C and C++, Java retains a great deal of type information in its compiled (".class") files. This information includes the names, types and modifiers[1] of each class field. Every Java object maintains

[1] I.e., private, protected, etc.

```
digraph NoName {
  n1 [label="LinkedList|{size: 3}",
          shape=record];
  n1 -> n2 [label="header"];
  n2 [label="Entry|{null}",shape=record];
  n2 -> n3 [label="next"];
  n3 [label="Entry|{A}",shape=record];
  n3 -> n4 [label="next"];
  n4 [label="Entry|{B}",shape=record];
  n4 -> n5 [label="next"];
  n5 [label="Entry|{C}",shape=record];
  n5 -> n2 [label="next"];
  n5 -> n4 [label="previous"];
  n4 -> n3 [label="previous"];
  n3 -> n2 [label="previous"];
  n2 -> n5 [label="previous"];
}
```

Figure 1: A graph specified in GraphViz notation

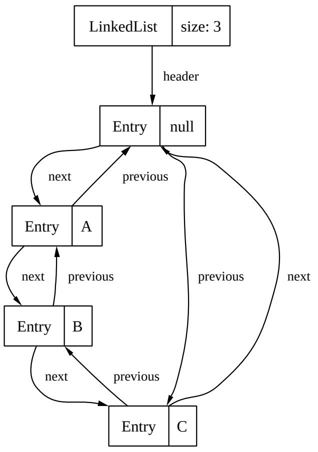


Figure 2: Sample output from GraphViz

a link to its class description[2], from which other fields of the object can be discovered. Reflection thus allows a GraphViz description to be generated for any arbitrary Java object by traversing the transitive closure of object references. The code that generates this description no longer needs to be attached to the code under study—a simple library call to "generate a snapshot of this object" is sufficient. A series of snapshots constitutes an animation (albeit free of any smooth interpolation between frames).

Some further difficulties presented themselves at this stage. In particular, some built-in Java types (notably strings) are actually implemented using character arrays, although the details of the implementation are usually kept hidden from the programmer. Java reflection is sufficiently powerful that these internal structure are revealed, as shown in 3. Here it can be seen that a `String` object consists of a reference to a (possibly shared) character array. A string is a subsequence of this array, specified by an offset and length (a `hash` value is also stored, to avoid the cost of recomputation).

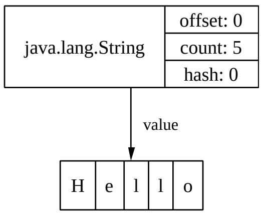


Figure 3: Exploring `String` internals

This is an insightful representation of a string that may prove helpful to students in certain contexts (for example, explaining the memory consumption of substring operations), but on the whole it provides far too much detail. The last stage in achieving a practical visualisation tool was the inclusion of a configurable *drawing context*. The tool's drawing context allows certain classes to be treated as if they were primitive, fields to be hidden, and colour, shape and font attributes to be set on graph nodes and edges. In most cases, a reasonable default context can be compiled into the visualiser. Changing the context is quite easy, and provides students with some scope to explore alternative visualisations.

## 2 Objectives and non-objectives

In designing the visualisation tool, we have been mindful of the serious pitfalls cited in the literature. Foremost amongst these is the need to actively engage students: "Visualisation technology . . . is of little educational value unless it engages learners in an active learning activity" (Naps, Rößling, Almstrum, Dann, Fleischer, Hundhausen, Korhonen, Malmi, McNally, Rodger & Velźquez-Iturbide 2003), a view echoed by Hundhausen & Douglas (2000): "the more actively involved learners are in the visualisation process, the better they perform." Also of concern "is the time and effort required for instructors to integrate the technology into their curriculum" (Naps et al. 2003). This effort includes not only the time required to learn new tools and to develop and/or adapt appropriate visualisations, but the costs of installing (often unreliable) software.

Active learning typically involves students creating visualisations of their own programs. Here, two immediate difficulties present themselves. First, class time is lost in explaining the use of the tools. Second, students are notoriously open to distraction, making it all too easy for the substantive course material to be sidelined as students become absorbed with the intricacies of the visualisation tool. We argue that an effective visualisation tool must therefore be extremely simple to use and provide limited scope for experimentation. A similar view is expressed by Naps (1998).

Our tool:

- is trivial to setup and easy to use[3];
- engages students in active participation;
- helps students connect their program code with the Java data model;
- is usable on any Java program, without requiring any specific programming conventions to be followed;
- allows "wrong" data structures to be viewed as well as correct ones;
- can be configure to elide unnecessary detail;
- allows incorporation of visualisations in reports and other presentations.

The tool presents all of its visualisations in terms of the Java data model. It is not possible to visualise, say, arrays as bar-charts, or incorporate other such abstractions. We have also consciously resisted complicating the tool to support, for example, visualisation of object-oriented (class inheritance) relations. Simplicity has been the overriding concern.

## 3 Overcoming student misconceptions

Java has a complicated data model that includes primitive types, object types, references to objects, arrays of primitives, arrays of object references, etc. Little wonder then that students routinely become confused. We have found that our visualisation tool is highly effective in clarifying student misconceptions about Java object semantics.

For example:

**String is not primitive** Java goes to some trouble to create the illusion that String is a primitive type. A special syntax for constructing and initialising static strings is provided (double-quotes), which gives a strong impression that no objects are involved. The default context for our visualisation tool partially plays along with this illusion, presenting Strings as objects with no internal structure. The illusion can be revealed, however, by using an "empty" context, in which no elision is performed; e.g., the graph in fig. 3 was produced with the code fragment:

```
drawGraph( newContext( ), "Hello" );
```

**Assignment does not create a new object** The difference between *value* and *reference* assignment is an eternal source of confusion for students. The visualisation tool clearly shows shared references as links to the same object (fig. 4(a)).

[2]Primitive types, such as `int`, `boolean`, etc., are linked to standard "wrapper" classes (`Integer`, `Boolean`, etc.) that describe these built-in types.

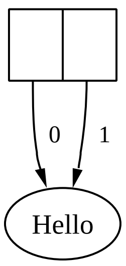
[3]To install the system, the system administrator will need to make sure `GraphViz` is available on the execution PATH. Students need just copy a single Java source file into their working directory. No changes to the Java environment, such as setting the `classpath`, are required. Creating a snapshot is a call to a single `static` method. Standard image viewers are perfectly adequate for watching the visualisations.

In the code that generated this example, an *initialised object array* is used to include two objects in a single graph. This is a general technique that can be trivially extended to any number of objects.

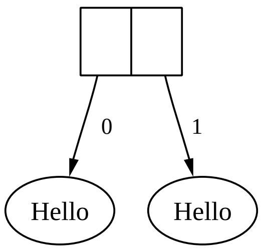
```
String x = "Hello";
String y = x;
drawGraph( new Object[]{ x, y } );
```

The alternative to reference assignment is a copy assignment, exemplified by the code fragment below (fig. 4(b)).

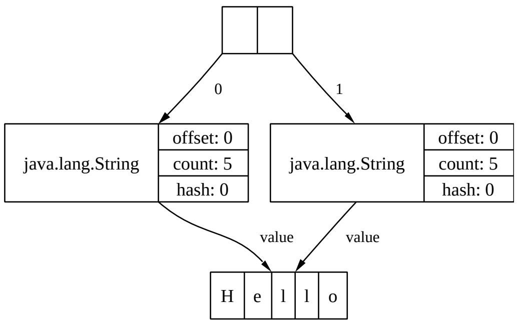
```
String x = "Hello";
String y = new String(x);
drawGraph( new Object[]{ x, y } );
```



(a) Reference assignment



(b) Value assignment



(c) Shallow copying

Figure 4: Reference and value assignment

Alternatively, an "empty" context can be used to reveal the extent to which copying of strings actually occurs (fig. 4(c)). Examples such as this are useful for introducing the concept of "shallow" versus "deep" copying.

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

**Object arrays hold references** An array of integers and an array of strings are modelled very differently in Java. Our visualisation tool clearly captures this distinction, displaying primitive values inline and displaying object values as separate nodes (fig. 5).

```
drawGraph( new Object[]{
    new String[]{ "a", "b", "c" },
    new int[]{ 1, 2, 3 } } );
```

**2-dimensional arrays** A very simple illustration of how 2-dimensional arrays are represented in the Java data model is given in fig. 6. The code that produced this visualisation is trivial; it simply create a rectangular 2-d array (so that the row and column ordinates can be deduced).

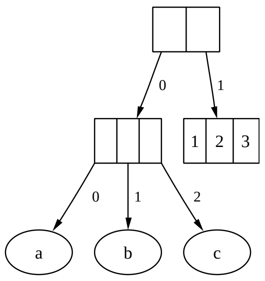


Figure 5: Object and primitive arrays

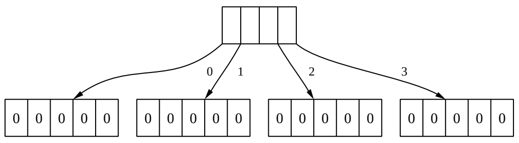


Figure 6: Java 2d arrays

```
drawGraph( new int[4][5] );
```

**Static fields are not part of any object** This fact is reinforced in every visualisation. Static fields are not displayed except in the (unusual) case of an explicit reference to a static object.

We have found this an especially helpful property of the visualisation tool, as students quickly refrain from adding spurious `static` modifiers to fields when they find nothing appears in their graph!

We have also found that the visualiser helps discourage students from declaring unnecessary fields, since these add clutter to all their graphs.

## 4 Customising the drawing context

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 inaccessible fields or treating system classes as primitive types.

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

**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" );
```

The available attributes are determined by the GraphViz tool.

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

**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 a surprisingly 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 ("…") can be inserted to replace the middle of excessively long strings.

**ignoreField** Suppresses display of the given field.

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

**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 a standard image viewer.

## 5 Data structure examples

We present several visualisations generated by our tool for the major data structures studied in a CS2-level course.

The first example (fig. 2) is a visualisation of the standard Java linked list class, and was generated from the following code fragment.

```
List xs = new LinkedList( );
xs.add( "A" );
xs.add( "B" );
xs.add( "C" );
defaultContext( ).ignorePrivateFields = false;
drawGraph( xs );
```

A *deque* data structure is shown in fig. 7. The snapshot, taken after 8 elements were added and then 3 removed, shows the relationship between the deque indexes (`_startM`, etc.) and the map and element buffers. The free space at both ends of the deque and the contiguous storage of elements are clearly apparent.

A *red-black tree* is shown in fig. 8. This example suggests the need for a drawing context that will select the colour attribute from an object field.

Finally, we present a chained hash map (fig. 9). This diagram was obtained from the Java library class, using a display context that included private fields.

```
Map hm = new HashMap( 4, 1.0f );
hm.put( "one",   new Integer(1) );
hm.put( "two",   new Integer(2) );
hm.put( "three", new Integer(3) );
getDefaultContext( ).ignorePrivateFields = false;
drawGraph( hm );
```

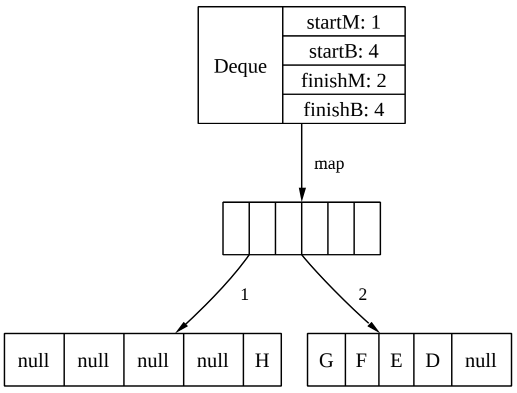


Figure 7: Visualising a Deque

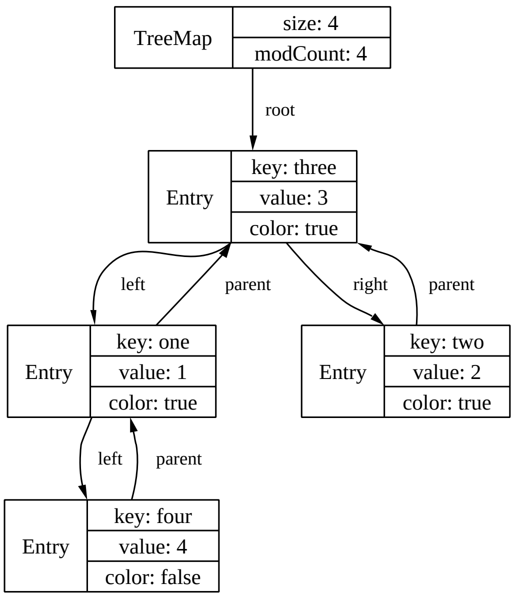


Figure 8: Visualising a red-black tree

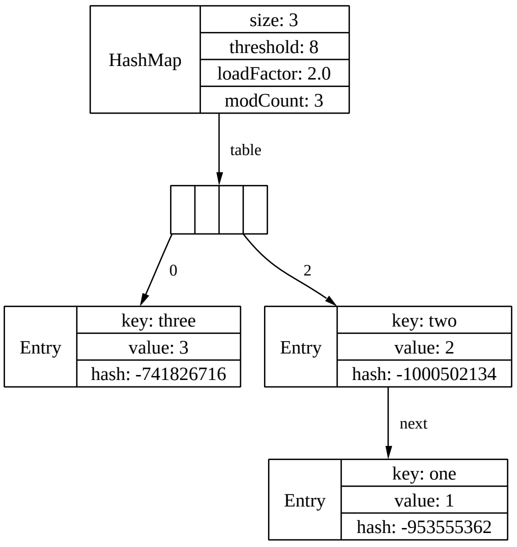


Figure 9: Visualising a Hash table

## 6 Related and future work

GraphViz is just one of many many free and commercial graph drawing tools currently available (e.g., see (Google.com 2003)). GraphViz was chosen because of its ease of installation, simple interface, and the aesthetic quality of the generated graphs. DaVinci (Fröhlich & Werner 1994), another popular graph drawing tool, can be driven through an API, making it better suited to displaying graphs while the program is running. Unfortunately, this would necessarily require setting up a non-standard Java environment, thus violating one of our stated aims.

North & Koutsofios (1994) presents a survey of graph visualisation applications, which includes mention of a prototype visual debugger, `vdbx`. `vdbx` sits on top of the widely used `dbx` debugger, and contains a parser that translates dbx-syntax C "structs" (i.e., records) into graph notation, allowing pointer structures to be browsed graphically while debugging. This is an appealing idea, although the tool itself appears to have fallen into disuse.

Recent releases of Java include a "platform debugger interface" (Sun Microsystems 2003) that is used by a wide variety of debugging tools. We are currently investigating adapting one of these tools to display graph visualisations along the lines of `vdbx`.

Gestwicki & Jayaraman (2002) present JIVE, an interactive Java object "inspector." JIVE incorporates some novel ideas for representing inheritance relations, and for displaying the currently executing source line next to the displayed objects. The current implementation of JIVE requires a source code transformation step, which the authors hope to replace with direct calls to the Java platform debugger.

Pedagogically, Naps's `Visualiser` 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 visualiser is more general, does not require any special configuration to handle new data structures, and is slightly easier to use. Naps's approach allows greater freedom in the presentation of data (e.g., bar charts for integer arrays) and supports arbitrary customisation.

## 7 Availability

The visualisation tool, comprising a single Java source file, can be requested by emailing the author.

The GraphViz utility can be downloaded for free from *http://www.graphviz.org*.

## References

Fröhlich, M. & Werner, M. (1994), Demonstration of the interactive graph visualization system daVinci, *in* R. Tamassia & I. Tollis, eds, 'Proceedings of DIMACS Workshop on Graph Drawing '94', Vol. 894, Springer Verlag, Princeton (USA).
**URL:** *ftp://ftp.Uni-Bremen.DE/pub/graphics/daVinci/papers/lncs.ps.gz*

Gestwicki, P. & Jayaraman, B. (2002), Interactive visualization of Java programs, *in* 'Symposia on Human Centric Computing Languages and Environments (HCC'02)', IEEE, Arlington, Virginia, USA, pp. 226–235.

Google.com (2003).
**URL:** *http://directory.google.com/Top/Science/Math/Combinatorics/Software/Graph˙Drawing/*

Hundhausen, C. & Douglas, S. (2000), Using visualizations to learn algorithms: Should students construct their own, or view an expert's?, *in* 'IEEE International Symposium on Visual Languages', pp. 21–30.

Naps, T. (1998), A Java visualizer class: Incorporating algorithm visualizations into students' programs, *in* 'Proceedings of ITiCSE'98', Dublin, Ireland, pp. 181–184.

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Velźquez-Iturbide, J. A. (2003), 'Exploring the role of visualization and engagement in computer science education', *ACM SIGCSE Bulletin* **35**(2), 131–152.

North, S. C. & Koutsofios, E. (1994), Application of graph visualization, *in* 'Proceedings of Graphics Interface '94', Banff, Alberta, Canada, pp. 235–245.
**URL:** *citeseer.nj.nec.com/221206.html*

Sun Microsystems (2003).
**URL:** *http://java.sun.com/products/jpda/*