

10 Viewers for Data Structures

Viewers for objects and primitives are briefly introduced in *Getting Started with Objects*, *The Workbench*, and *The Integrated Debugger*. In this tutorial, we introduce a family of “Presentation” views for data structures. A presentation view is a conceptual view similar to what one might find in a textbook but with the added benefit of being dynamically updated as the user steps through the program.

Objectives – When you have completed this tutorial, you should be able to open a viewer for any data structure object displayed in the Debug or Workbench tabs, set the view options in the viewer window, and select among the views provided by the viewer.

The details of these objectives are captured in the hyperlinked topics listed below.

- 10.1 Introduction
- 10.2 Opening Viewers
- 10.3 Setting the View Options
- 10.4 Selecting Among Views
- 10.5 Presentation Views for LinkedList, HashMap, and TreeMap
- 10.6 Presentation Views for Code Understanding
 - 10.6.1 LinkedListExample.java
 - 10.6.2 BinaryTreeExample.java
 - 10.6.3 Configuring Views generated by the *Structure Identifier*
- 10.7 Using the Viewers from the Workbench
- 10.8 Summary of Views
- 10.9 Exercises

10.1 Introduction

jGRASP viewers are tightly integrated with the workbench and debugger. They can be opened for any primitive, object, or field of an object in the Debug or Workbench tabs. To use viewers with the debugger, (1) set a breakpoint in your program, (2) run Debug () (3) after a local variable has been created, drag it from the Debug tab, (4) step through program and observe the object in the viewer. To use viewers with the workbench, (1) create an instance from the UML window, the CSD window () or Interactions tab, (2) drag the instance from the Workbench tab, (3) invoke methods on the instance and observe the object in the viewer.

Note that once an instance is in the workbench tab or debug tab, its methods can be invoked via the Invoke Method dialog or by entering Java statements and/or expressions in the Interaction tab. When methods are invoked on the instance, any open viewers on it are updated as appropriate.

jGRASP includes a general view for data structures called *Presentation – Structure Identifier (SI)* which automatically detects linked lists, binary trees, and array wrappers (lists, stacks, queues, etc.) when a viewer is opened on one of these during debugging or workbench use. For linked structures, this is an animated view that shows nodes being added and deleted from the data structure. This view is also configurable with respect to the structure mappings and the fields to display. jGRASP also includes custom *Presentation* views for many of the classes in the Java Collections Framework (e.g., ArrayList, Stack, LinkedList, TreeMap, and HashMap). These non-animated views are optimized for large numbers of elements.

10.2 Opening Viewers

Let's begin by opening one of the example programs that comes with the jGRASP installation. After you have started jGRASP, use the Browse tab to navigate to the jGRASP\examples\Tutorials folder. If you have been working with the examples in the "Hello" or "PersonalLibrary" folders, you'll need to go up one level in the Browse tab by clicking the up arrow. [Note that you should copy this folder to a personal directory. This will need to be done using a file browser rather than jGRASP.] In the Tutorials folder you should find a folder called ViewerExamples. Open this folder by double-clicking on the folder name, and you should see a file called *ArrayListExample1.java*. Open this file by double-clicking the file name (Figure 10-1).

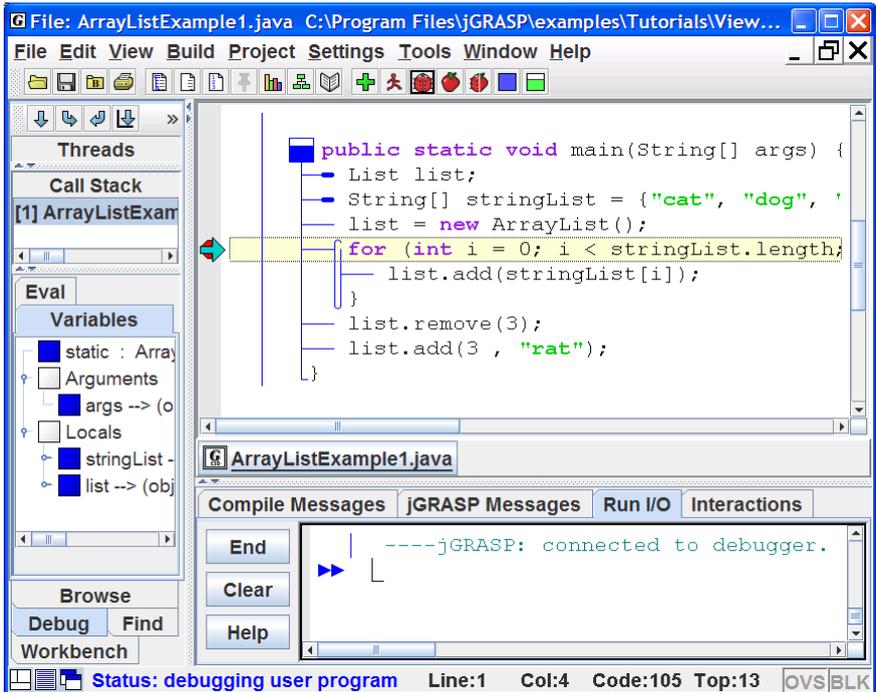


Figure 10-1. *ArrayListExample1.java*

A quick review of the program shows that it creates an `ArrayList` called `list` and adds strings to it from an array called `stringList`. Compile the program by clicking the green plus **+**. Since the viewers are for visualizing objects and primitives as the program executes, let's set a breakpoint on the first line of the `for` statement. To do this, move the mouse over the left margin next to the statement until you see the breakpoint symbol **●** and then left-click. You should see the **●** in the margin if you have successfully set the breakpoint. Now start the debugger by clicking **🐞** on the toolbar. Figure 10-1 shows the program stopped at the `for` statement. At this point in the program, the `list` object has been created, and it is shown in the Debug Variables tab. However, no elements have been added to `list`.

A separate **Viewer** window can be opened for any object (or field of an object) in the debug tab (or on the workbench). The easiest way to open a viewer is to left-click on an object and drag it from the debug tab (or workbench) to the location where you want the viewer to open. When you start to drag the object, a viewer symbol should appear to indicate a viewer is being opened. [Note: You can also open a viewer by right-clicking on the `list` object and selecting either

View by Name or View Value.] Let's left click on *list* and drag it from the Debug tab. When you release the left mouse button, the viewer should open.

Figure 10-2 shows a viewer opened on *list* before any elements have been added. Note that the default *View* for an instance of *ArrayList* is *Presentation – Structure Identifier*. This view shows the fields for *size* and *modCount* along with the underlying array with its default size of 10.

To add elements to *list*, step through the program by clicking the “Step” button  on the Debug tab. Since the viewer is updated on each step, you should see the elements being added to the list. Red text indicates a change or, in this case, a new element. Figure 10-3 shows the view of *list* after going through the loop three times. As you continue to step through the program, notice that when elements are removed, the value stays in the array but the *size* is decremented.

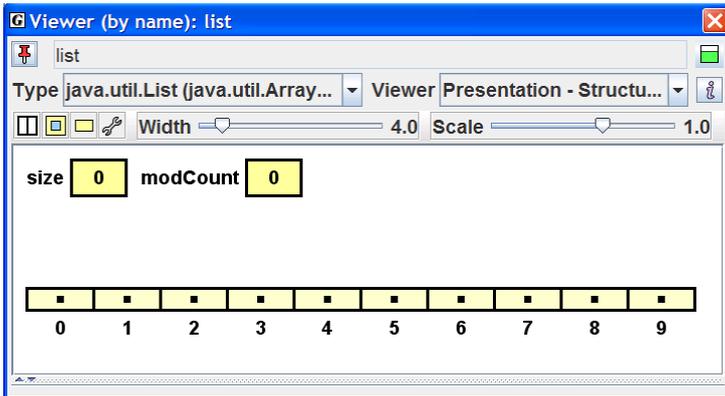


Figure 10-2. View of *list* with no elements

Each jGRASP Presentation viewer provides one or more subviews. When an element is selected as indicated by a red border, a view of the element itself is shown in the subview. Figure 10-3 shows “ant” selected in the *ArrayList* view. Since “ant” is a *String*, the subview is a *String* viewer for which the *formatted* view is the default.

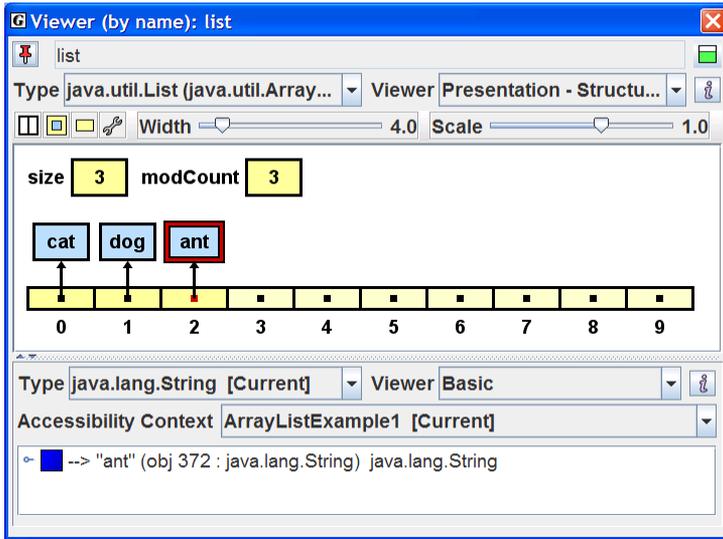


Figure 10-3. View of *list* with 3 elements

10.3 Setting the View Options

For most Presentation views in jGRASP, several *view* options are available which provide personal choices to users.

Horizontal vs. Vertical – sets the orientation of the display.

Non-Embedded/Embedded – shows the elements outside or inside the structure.

Normal vs. Simple – shows node pointer from inside or from edge of structure.

Configure View – opens dialog to configure the structure-to-view mapping as well as which fields to display in the viewer (discussed in Section 9.5).

Width of Elements (slider) – sets the width of the boxes containing the elements.

Scale of View (slider) – scales the entire view.

Figure 10-4 indicates the location of the buttons and sliders for each view option. Click on each of these and notice the change in the view. The `ArrayList` is shown vertically after the display orientation is changed. The location of the **View** drop down list and the Information button  is also indicated below.

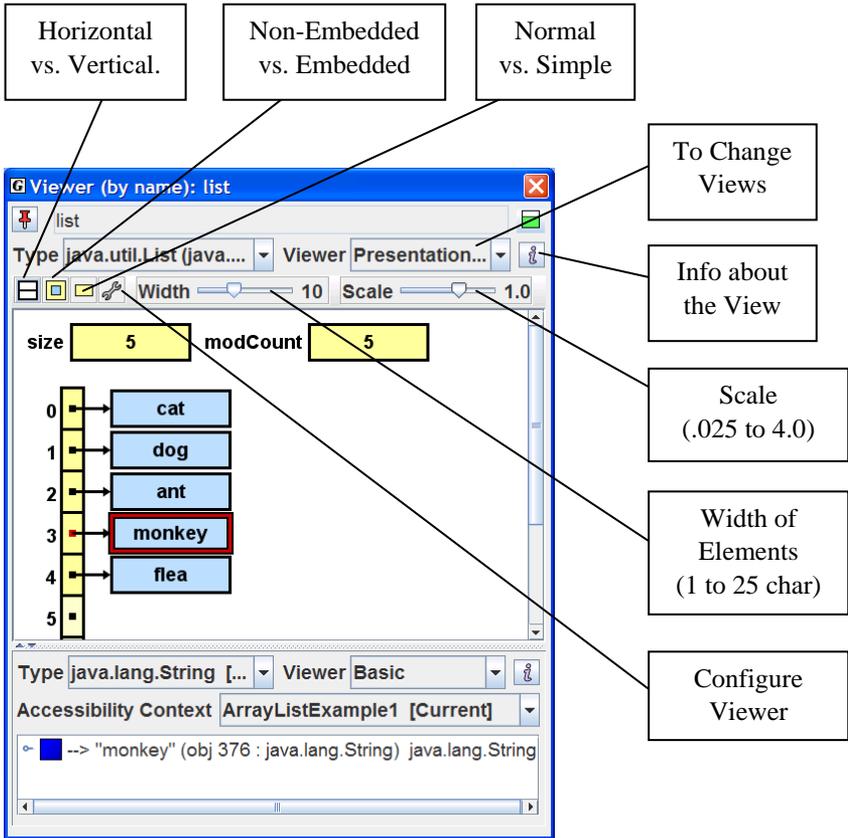


Figure 10-4. View of *list* in vertical mode, width set to 8, Scale set to 1.2, *monkey* selected and shown in subview

10.4 Selecting Among Views

Each viewer and subviewer provides one or more views among which you may select from the respective *View* drop down lists. Let's take a closer look at our ArrayList of Strings example. The *Presentation – Structure Identifier* view is the default for ArrayList and the other classes in the Java Collections Framework. Other views include *Basic*, *toString()*, *Presentation*, and *Collection Elements*.

Figure 10-5 shows the view options for ArrayList on the drop-down list (combo box) with *Presentation – Structure Identifier* view selected. When *monkey* is selected in the ArrayList, a String subview is opened. When this view is set to *Presentation* view, the character array for *monkey* is displayed. Selecting the first element in the array opens a subview for character *m* in monkey which is set to the default *Basic* view.

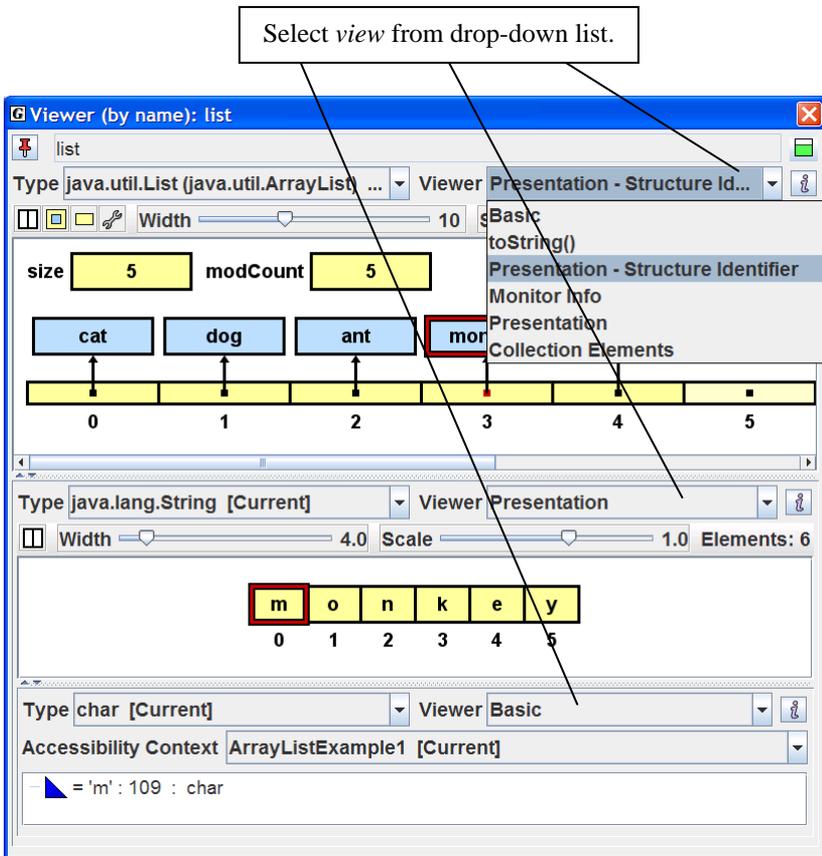


Figure 10-5. Selecting the *Collection Elements*

Figure 10-6 shows the viewer after the *Collection Elements* view is selected. If *list* has many elements, this may be a more appropriate view than the *Presentation* view. The *Collection Elements* view was specifically designed to handle larger numbers of elements efficiently. As the number of elements increases, additional navigational controls appear on the viewer for moving about in the *ArrayList*. Notice that two subviews are also shown in Figure 10-6. When element 0 (indicated by “<0> = **cat**”) is selected in the *Collection Elements* view, a subview for *String* opens below the main view. Notice that the view for *String* has been set to *Presentation* in the figure; the default for *String* is *Formatted*. When the ‘c’ in “cat” is selected, a second subview is opened for the primitive type *char*, for which *Basic* is the default view. However, in the figure, the view has been set to *Detail*, which displays additional information about ‘c’ including its value in hexadecimal, octal, and binary.

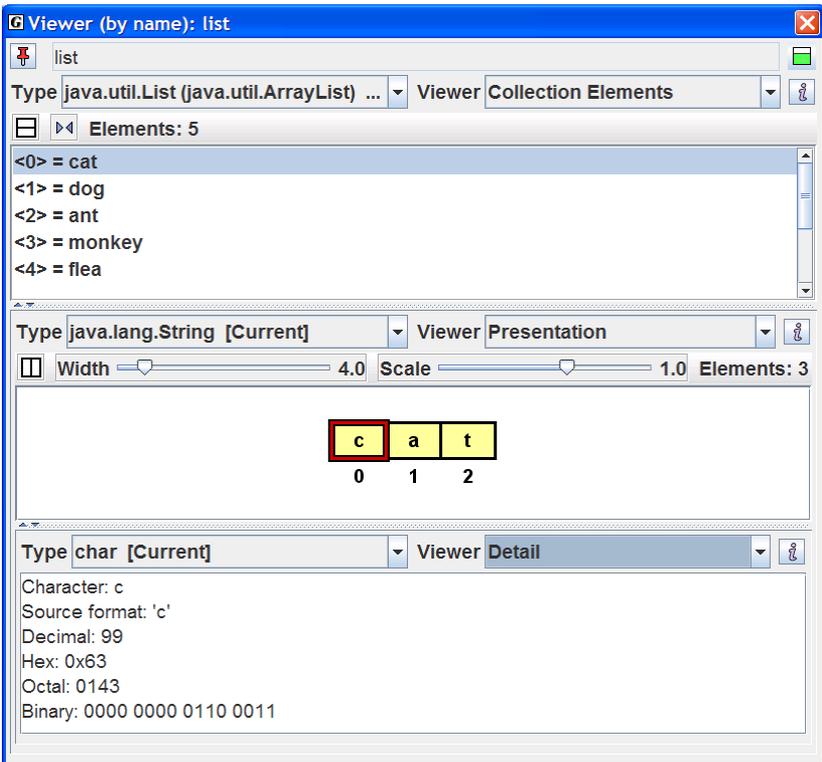


Figure 10-6. The *Collection Elements* view of *list* with two subviews: *Presentation* view for *String* “cat” and *Detail* view for *char* ‘c’

10.5 Presentation Views for LinkedList, HashMap, and TreeMap

The ViewerExamples folder contains a program, CollectionsExample.java, which creates instances of classes from the Java Collections Framework, including Vector, ArrayList, LinkedList, Stack, TreeMap, and HashMap. In this section, we'll take a look at *Presentation* views for several of these.

In the Browse tab, locate CollectionsExample.java, and double-click on it to open it in the CSD window. Compile the program by clicking the green plus . Set a breakpoint on any executable statement in the program. Now start the debugger by clicking . Figure 10-7 shows the program stopped at a breakpoint on the line in the inner loop that adds an element to myVector. Notice that prior to the breakpoint, the variables myVector, myArrayList, myLinkedList, myStack, myHashMap, and myTreeMap were declared and their respective instances were created. With the program stopped at the breakpoint, we can open viewers for each of the variables listed in the Debug tab.

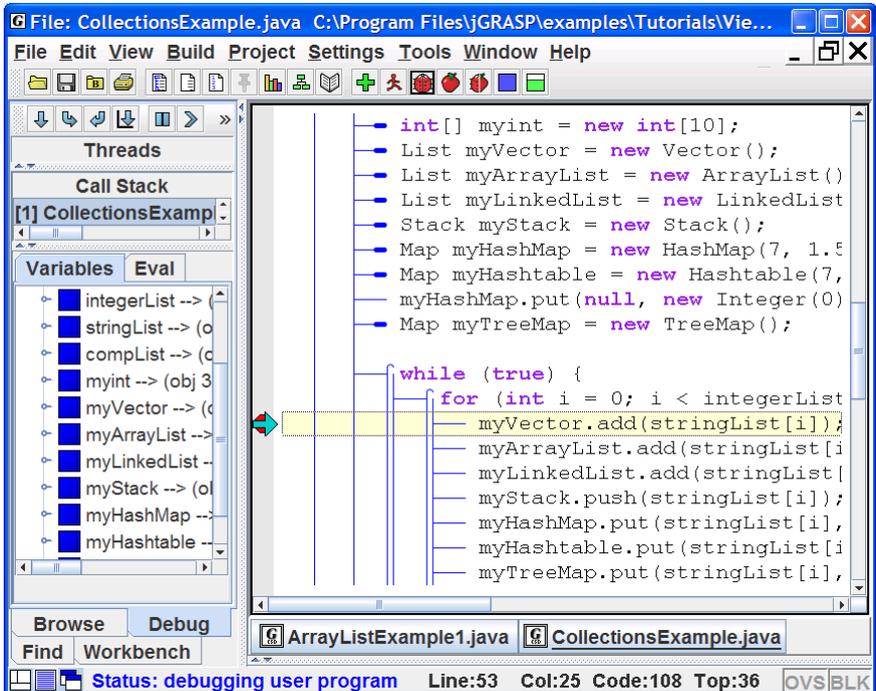


Figure 10-7. CollectionsExample.java stopped at a breakpoint

Figure 10-8 shows a viewer set to *Presentation - Structure Identifier* view opened on *myLinkedList* after three elements have been added to it. Notice that *myLinkedList* is a doubly-linked list with a header node. Width has been set to 8.0, and the element *mouse* is selected in the main view and shown in the subview in *Presentation* view for the String Class. In this view, the *m* in *mouse* is selected, and the character subview is shown set to the *Basic* view.

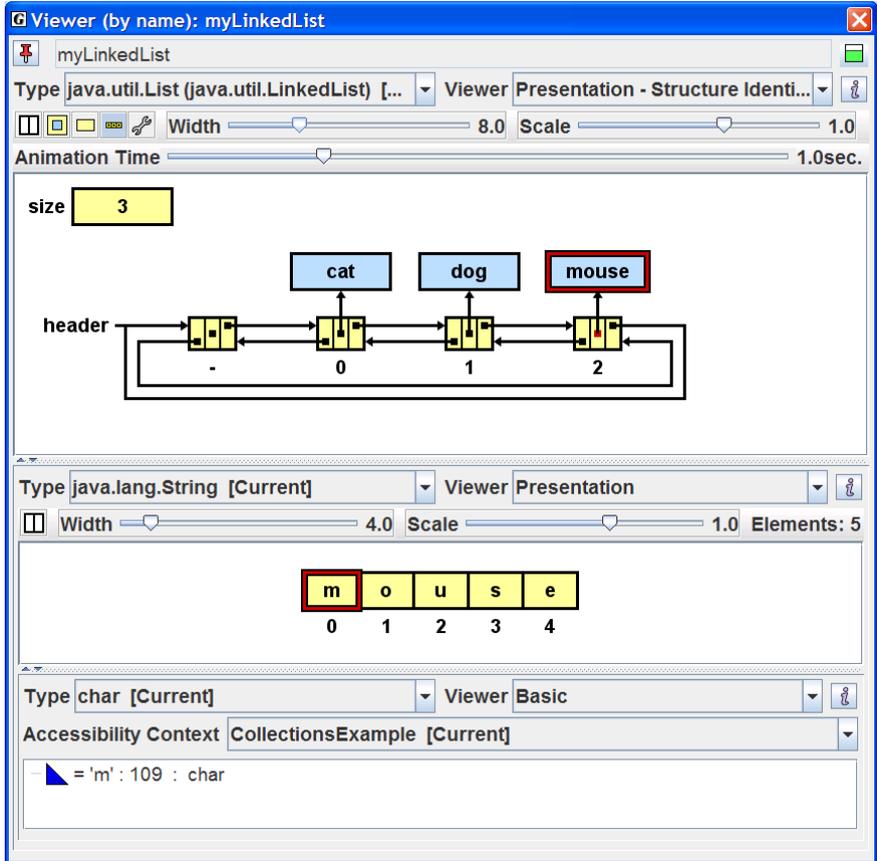


Figure 10-8. View of *myLinkedList* after three elements have been added

Figure 10-9 shows a viewer set to *Presentation - Structure Identifier* view opened on the variable `myHashMap` after three elements have been added. A hashmap entry is selected, as indicated by the red border, and its *Basic* view is shown in the subview with fields: *key*, *value*, *hash*, and *next*. As elements are added to the HashMap, it is useful to use the Scale slider to zoom in and out on the structure so that the “topology” of its elements can be seen.

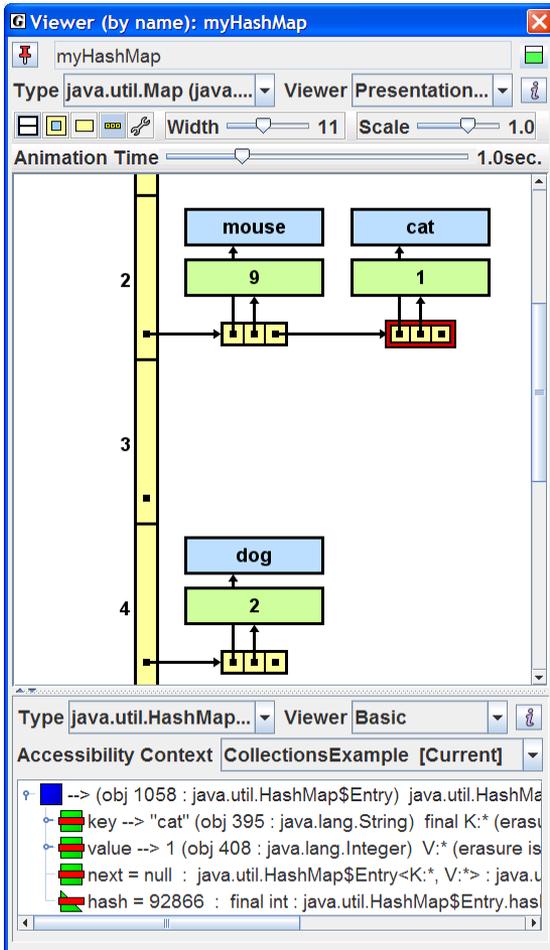


Figure 10-9. View of `myHashMap` after three elements have been added

Figure 10-10 shows a viewer opened on myTreeMap after seven elements have been added. TreeMap uses a *Red-Black* tree as its underlying storage structure, and the default *Presentation - Structure Identifier* view indicates the red and black nodes by coloring their borders light red and dark gray respectively. As you step through the program and put items in the TreeMap, you should see the red-black node rotations.

In the figure, width has been set to 11.0, and in the red node containing “ant”, the key field “ant” has been selected as indicated by an additional dark red border. The String subview for *ant* is set to *Presentation*.

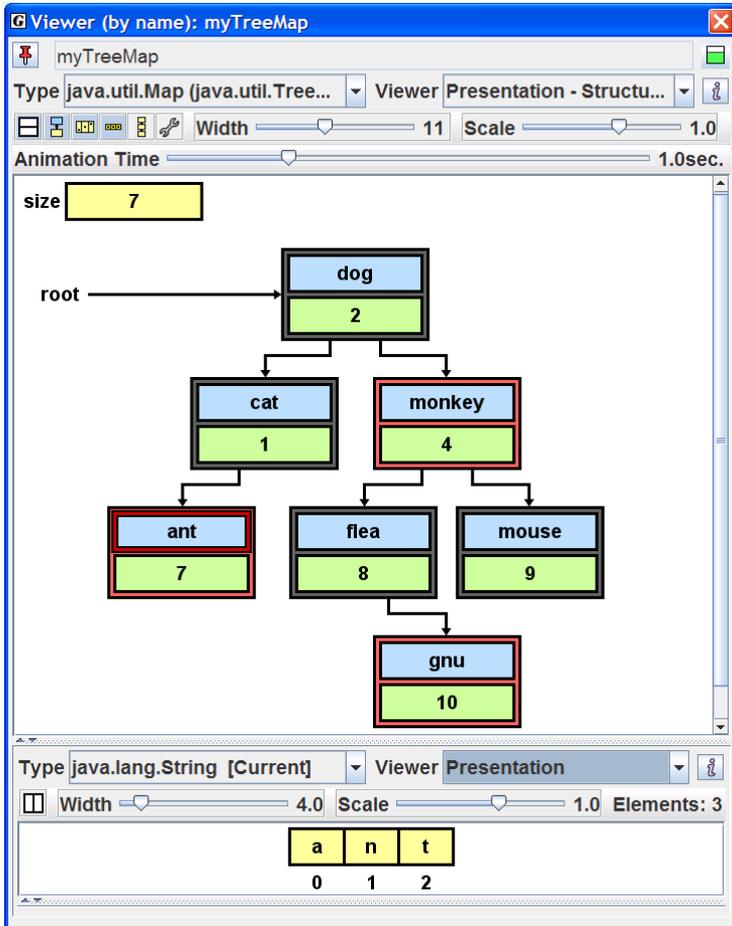


Figure 10-10. *Presentation - Structure Identifier* View of myTreeMap with six elements

Figure 10-11 shows a second viewer opened on `myTreeMap` with the view set to *Key/Value*. The node for *dog* has been selected and two subviews have been opened: one for the *key* and one for the *value*. In the figure below, the node with *key* = “dog” and *value* = 2 has been selected.

In the left subview for String *key*, the view is set to *Presentation* as it was in the previous figure. In the right subview, we have the *Basic* view of *value* which is an object; specifically, it is an instance of `java.lang.Integer`, the wrapper class for the Java primitive *int*.

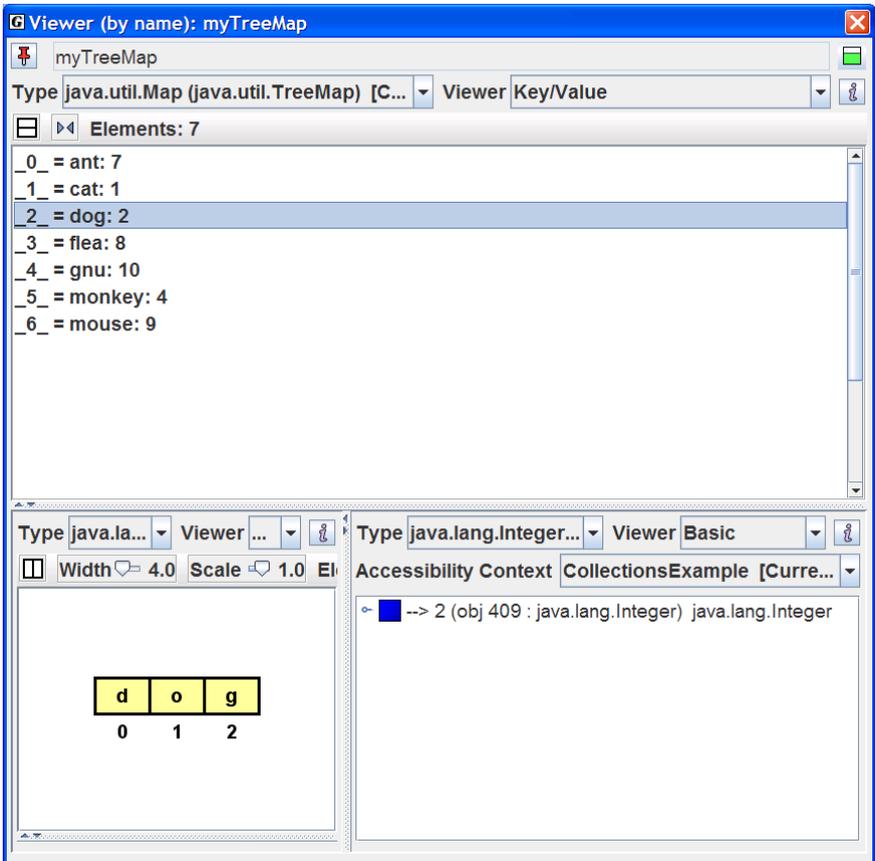


Figure 10-11. *Key/Value* view of `myTreeMap` with seven elements

10.6 Presentation Views for Code Understanding

Now we turn our attention to the details of the *Presentation - Structure Identifier* viewer when it is used in conjunction with user classes for data structures including most textbook examples. When this viewer is opened on an object, it automatically attempts to determine if the object represents a common data structure; if so, it verifies relevant links, displays nodes referenced by local variables, and provides animation for the insertion and deletion of nodes. The structure mappings that are determined by the viewer and the fields that are displayed in the view can be configured by the user while the viewer is open on the object.

Custom *Presentation* views (as opposed to the more general *Presentation - Structure Identifier* view) are available for many of the Java Collections Framework classes. Each of these views is generated by a non-verifying viewer implemented specifically for the respective class. Because these viewers assume that the JDK Java code for each data structure is correct, no verification is done. As a result, these viewers can efficiently display data structures with large numbers of elements. In contrast, the *Presentation - Structure Identifier* view is less efficient but provides link verification and animation. It is extremely useful when viewing a data structure with a relatively small number of elements (e.g., less than 100) while attempting to understand the source code itself. For example, when stepping through the insert method, this view shows links being set for a local node instance and then shows the node sliding up into the data structure. Seeing a link set as a result of a particular assignment statement helps the user make a mental connection between the source code and the actual behavior of the program during execution.

When a viewer is opened on an object, the *Structure Identifier* attempts to determine if the underlying structure of the object is a linked list, binary tree, or array wrapper (lists, stacks, queues, etc.). The object's fields and methods are examined for references to nodes that themselves reference the same type of node. If a positive identification is made, the data structure is displayed; otherwise, the user is given the opportunity to configure the view. The *Presentation - Structure Identifier* view works for all of the Collections Framework Classes used in the examples above, and it should work for most user classes that represent data structures. During the generation of the visualization, relevant *links* are verified and then displayed in a specific color to denote the following: **black** – part of structure; **green** – local reference or not part of the formal data structure; **red** – in transition or probably incorrect for specified structure. The most distinguishing aspect of this presentation view is the animation of node insertions and deletions. The control buttons and sliders

on the viewer are similar to ones discussed above with the addition of a slider to set the animation time.

Now let's look at several example programs that use non-JDK data structures similar to what you might find in a textbook. In the `Tutorials\ViewerExamples` directory, we have `LinkedListExample.java`, `DoublyLinkedListExample.java`, and `BinaryTreeExample`. The actual data structure classes used by these examples are in the folder `jgraspvex`, which is a Java package containing `LinkedList.java`, `DoublyLinkedList.java`, `BinaryTree.java`, `LinkedListNode.java`, and `BinaryTreeNode.java`.

10.6.1 LinkedListExample.java

In the Browse tab, navigate to the `ViewerExamples` directory and open the file `LinkedListExample.java` by double-clicking on it. Generate the CSD, and then compile the program by clicking  on the toolbar. Set a breakpoint  in the left margin on a line inside the inner loop (e.g., on the line where `list` is declared and a new `LinkedList` object is created). Now click the Debug button  on the toolbar. Figure 10-11 shows the program after it has stopped at the breakpoint prior to creating an instance of `LinkedList` called `list`. Click Step  on the controls at the top of the Debug tab. When `list` is created, you should see it in the Variables tab of the Debug window.

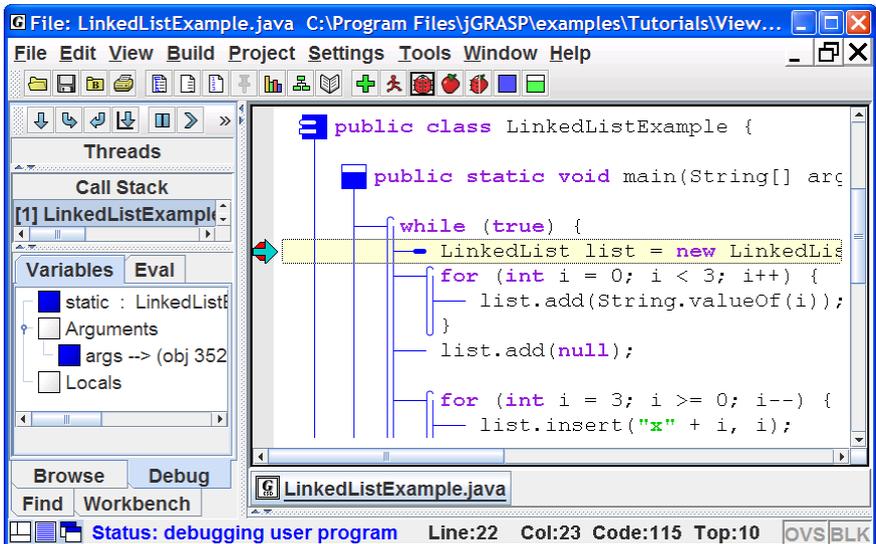


Figure 10-11. `LinkedListExample.java` stopped at a breakpoint

Now open a viewer on *list* by selecting and dragging *list* from the Debug window. Figure 10-12 shows a view of *list* before any elements have been added. Add two elements to the linked list by stepping (↓) through the inner loop twice. Figure 10-13 shows a view of *list* after two elements have been added. Note that the viewer is set to *Presentation – Structure Identifier* view, which is the default. *Basic* and *Monitor* views are also available. The latter view displays any Java owning or waiting threads for the monitor associated with the object. This is used for multi-threading and synchronization. After experimenting with the other views, change the *View* to *Presentation - Structure Identifier* by selecting this on the drop down list as shown in Figure 10-13.

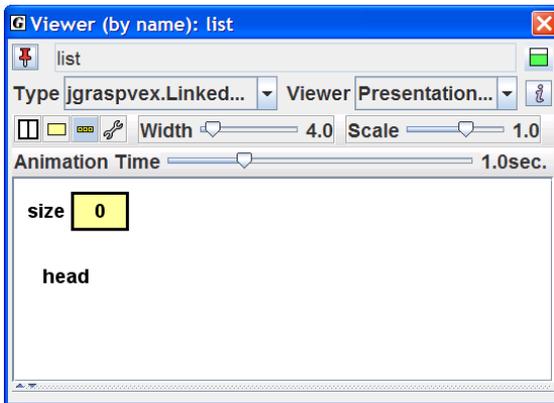


Figure 10-12. View of *list* with no elements added

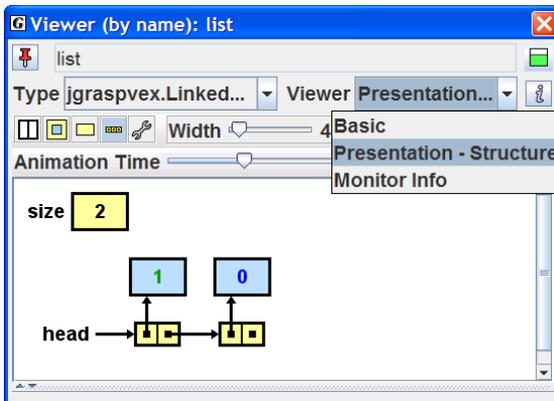


Figure 10-13. View of *list* with two elements

Now you are ready to see the animation of a local node being added to the linked list. You need to step into the `add()` method by clicking the *Step in* button  at the top of the debug tab. Each time you click , the program will either step into the method indicated or step to the next statement if there is no method call in the statement. Figure 10-14 shows `list` after `node.next` for the new node has been set to `head`. Figure 10-15a shows `list` after `head` has been set to `node`, and the new node begins to move into `list`. Figure 10-15b shows `list` after the new node has been inserted. As you repeatedly *step in*, you should see added and inserted nodes “slide” up into `list` and removed nodes slide out of `list`. Note that the Call Stack in the Debug tab indicates the methods into which you have stepped.

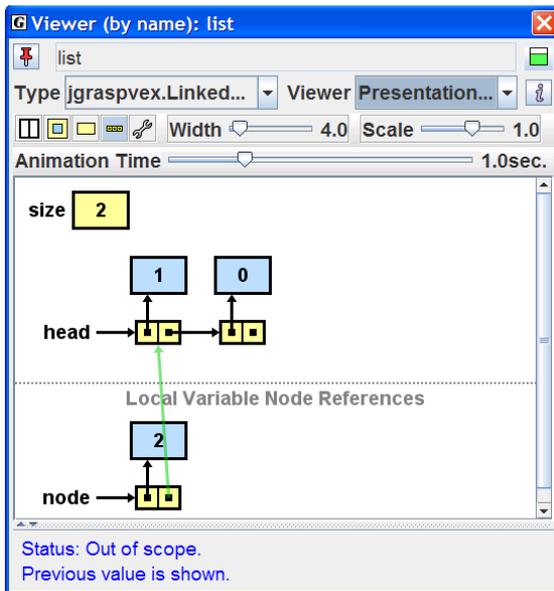


Figure 10-14. Node about to be added to `list`

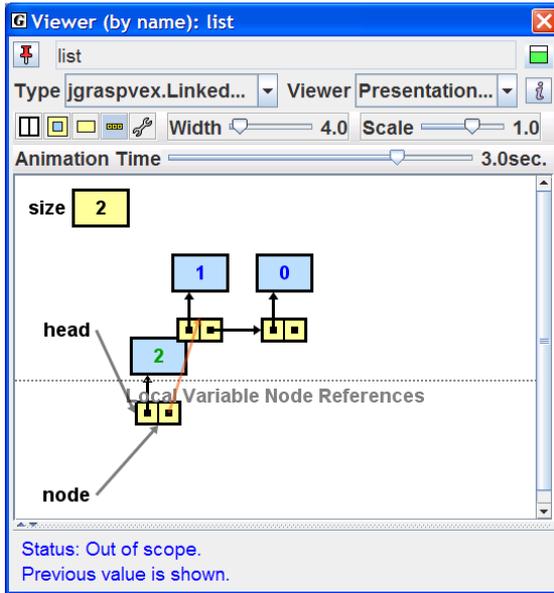


Figure 10-15a. As node is it is being added to *list*

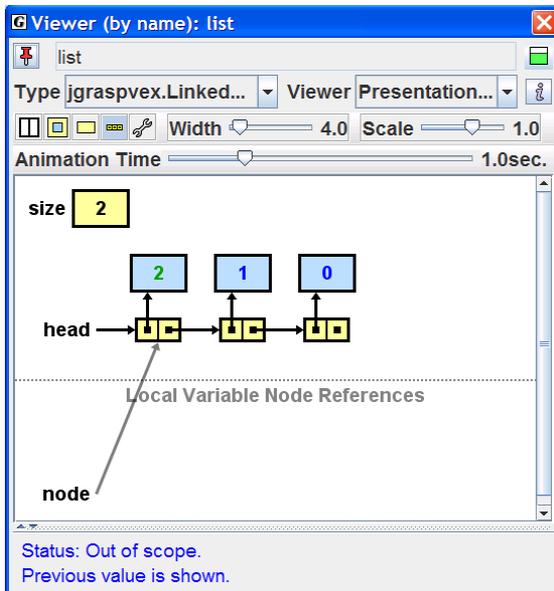


Figure 10-15b. After node has been added to *list*

10.6.2 BinaryTreeExample.java

Now let's take a look at an example of another common data structure, the *binary tree*. In the Browse tab, navigate to the *ViewerExamples* directory and open the file *BinaryTreeExample.java* by double-clicking on it. After compiling it, set a breakpoint  in the left margin on a line inside the inner loop (e.g., on the line where *bt.add(..)* is called). Now click the Debug button  on the toolbar. Figure 10-16 shows the program after it has stopped at the breakpoint prior to adding any nodes to *bt*. Now open a viewer on *bt* by selecting and dragging it from the Debug window. The *Structure Identifier* automatically determines that the object is a binary tree and provides an appropriate view for *bt*. Add two elements to *bt* by stepping () through the inner loop twice.

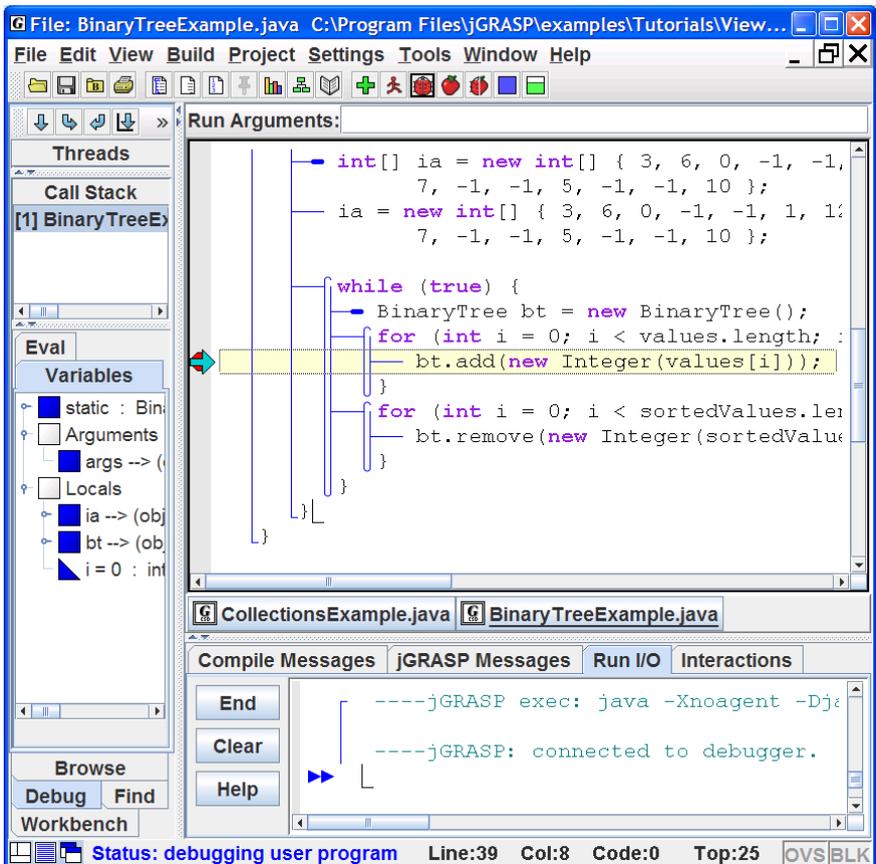


Figure 10-16. *BinaryTreeExample.java* stopped at a breakpoint

Now you are ready to see the animation of a local node being added to the binary tree. You need to step into the *add* method by clicking the *Step in* button  at the top of the debug tab. Each time you click , the program will either step into the method indicated or step to the next statement if there is no method call in the statement. The Call Stack in the Debug tab indicates the methods into which you have stepped. Figure 10-17 shows *bt* after *root* has been passed into the *add()* method as *branch*, and Figure 10-18 shows *bt* after *branch.left* has been set to *node*. As you repeatedly *step in*, you should see added and inserted nodes “slide” up into *bt* and removed nodes slide out of *bt*. Note that since *bt* is a local variable declared in the main method, when you step in to a method as we done in this example, *bt* is no longer in scope. This is indicated by the message at the bottom of the viewer. Because *bt* is a reference variable, the *previous value* still points to the instance of *BinaryTree* that we are viewing.

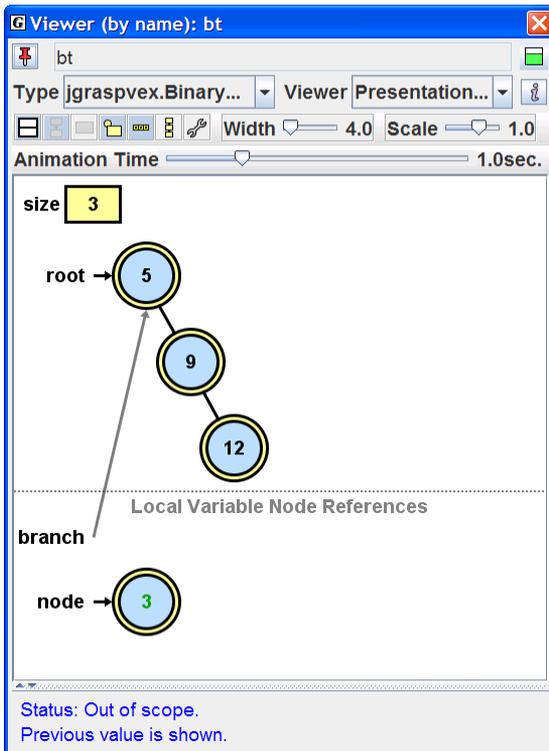


Figure 10-17. Binary tree example as node is about to be added to *bt*

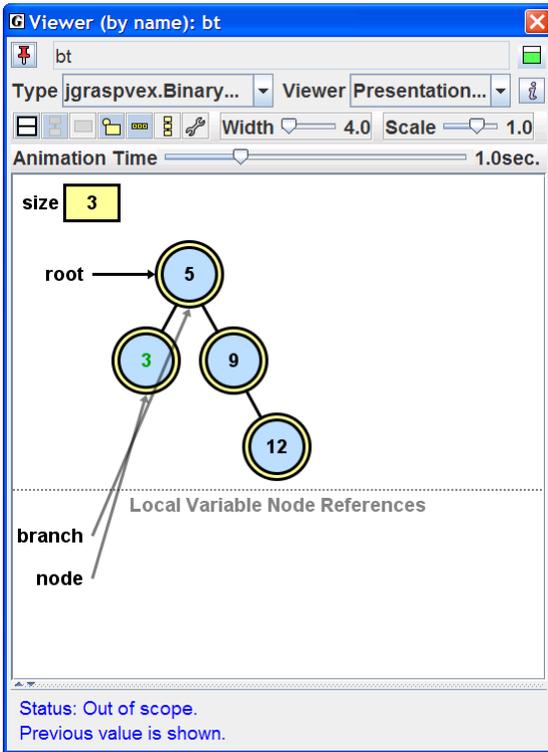


Figure 10-18. Binary tree example after node is added to *bt*

This is a good time to do some experimenting on your own with this example. For example, click the Debug button  to start the program. Click *step* () until *bt* is created, then open a viewer on it. Now, as you *step* () through the code, try to understand exactly what is happening in the program with respect to the diagram in the viewer.

Now repeat the process above, but this time click *step in* () repeatedly. The viewer will show the relationship between the data structure and local nodes in its methods, and the animation should help you understand the code in these methods.

10.6.3 Configuring Views generated by the *Structure Identifier*

The *Structure Identifier* uses a set of heuristics in its attempt to determine if the object for which a view is being opened is a linked list, binary tree, etc. Since the view it provides is only a best guess, some additional configuration may be needed in order to attain an appropriate *Presentation* view. Consider the viewer in Figure 10-19. Figure 10-20 shows the result of (1) clicking the *Configure* button  (located to the left of the *Width* slider). Figure 10-21 shows the dialog after modifying **Value Expression** by inserting "**Value:** " + (don't forget to enter the plus sign). Figure 10-22 shows the binary tree after clicking **OK** or **Apply** on the Configure dialog. Note that *Width* has been changed to 8.0 to accommodate the new node value.

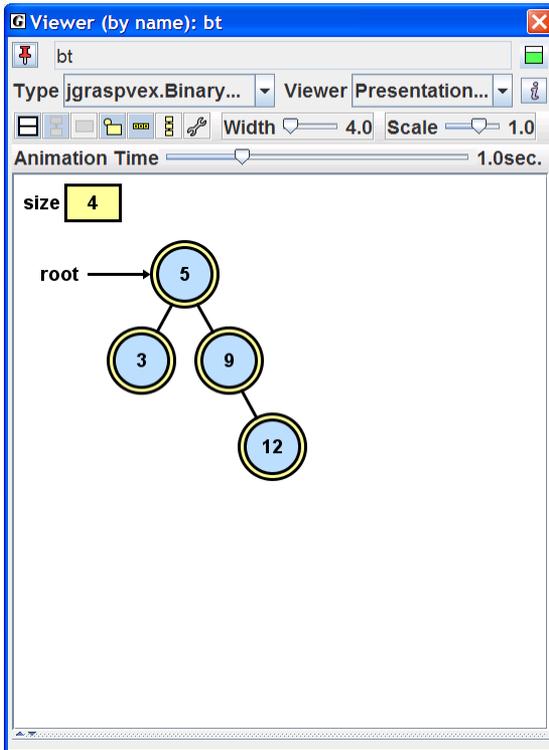


Figure 10-19. Binary tree example

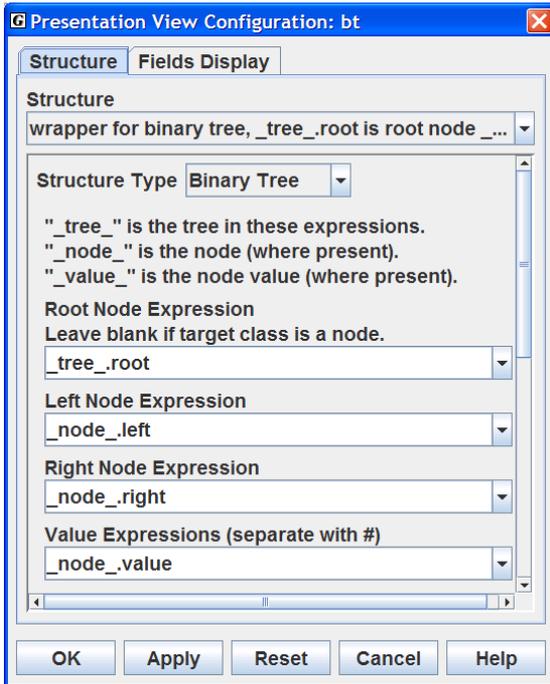


Figure 10-20. Configuration dialog (✎)

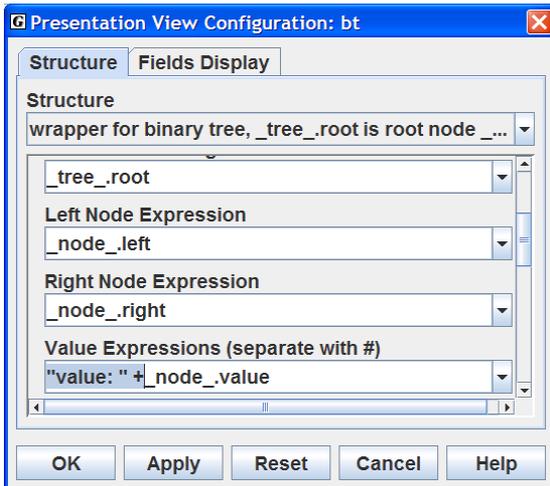


Figure 10-21. Configuration dialog with Value Expression modified

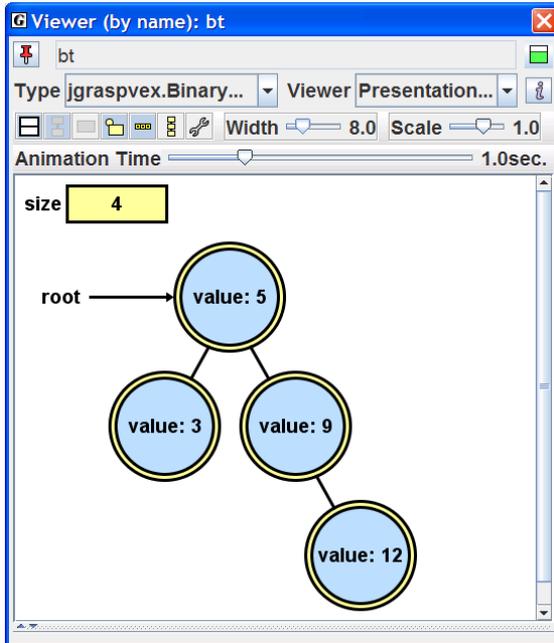


Figure 10-22. Binary tree example after OK (or Apply) on Configuration dialog

The *Structure* tab in the Configuration dialog includes: (1) *Structure* with a drop down list for the possible structure mappings identified by the *Structure Identifier*, (2) *Structure Type* with a drop down list containing *Binary Tree*, *Linked List*, *Hashtable*, and *Array Wrapper*, and (3) entries describing the structure itself. Currently, modifications made via the Configuration dialog are not saved from one jGRASP session to another.

Continuing with the binary tree example, Figure 10-23 shows the *Structure Type* for *bt* after it has been changed from *Binary Tree* to *Linked List*. Figure 10-24 shows the data structure after the configuration change has been applied (i.e., OK or Apply clicked). Notice that transparent red arrows represent links that are not correct for a linked list.

The *Structure* tab is intended primarily for advanced users, and structure changes are rarely needed to view most common data structures. After experimenting with these settings, be sure to set the configuration back to its defaults by clicking the Reset button, then Apply or OK.

The *Fields Display* tab provides some options with respect to which of the object's fields should be displayed. This is the most common configuration operation to perform on the view provided by the *Structure Identifier*. For some data structures one (or more) of the fields is treated as a formal part of the conceptual diagram itself. For example, the binary tree example has two fields, *size* and *root*, and the viewer treats *root* as part of the diagram, but considers *size* to be optional (however, it is included by default). Only the fields that are not part of the diagram are listed on the *Fields Display* tab.

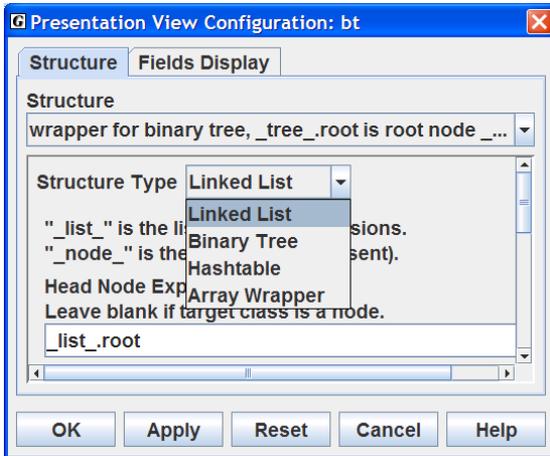


Figure 10-23. Changing structure type of *bt* from Binary Tree to Linked List

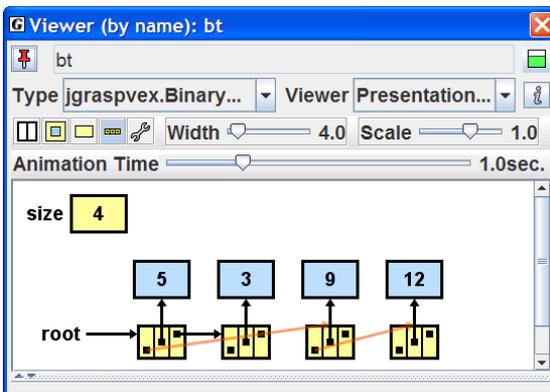


Figure 10-24. *bt* shown as a linked list with red translucent links indicating it is not a linked list

10.7 Using the Viewers from the Workbench

Thus far, we have concentrated on opening viewers from the Debug tab while a program is being run in debug mode. In this section, we'll see how to use viewers from the Workbench tab. Objects can be created and placed on the workbench from the CSD window, the UML window, and/or by entering appropriate source code in the Interactions tab. After an object is placed on the workbench tab, a viewer can be opened by selecting the object and dragging it from the Workbench tab.

Let's begin by opening the project for the BinaryTreeExample we used in the previous section. In the Browse tab, navigate to the *ViewerExamples* directory and open the file *BinaryTreeExample_Project.gpj* by double-clicking on it. After this file is opened, you should see the project listed in the "Open Projects" section of the Browse tab. If the UML diagram is not displayed, double-click on the UML diagram symbol ( <UML>) which should be the first entry under the project in the Browse tab. Figure 10-25 shows the UML diagram with three classes: *BinaryTreeExample*, *BinaryTree*, and *BinaryTreeNode*. Notice that the labels for *BinaryTree* and *BinaryTreeNode* indicate they are contained in package *jgraspvex* (see the *jgraspvex* folder in the current directory).

If you are still running a program in jGRASP (e.g., in debug mode from the previous section), you should end it before you start the workbench.

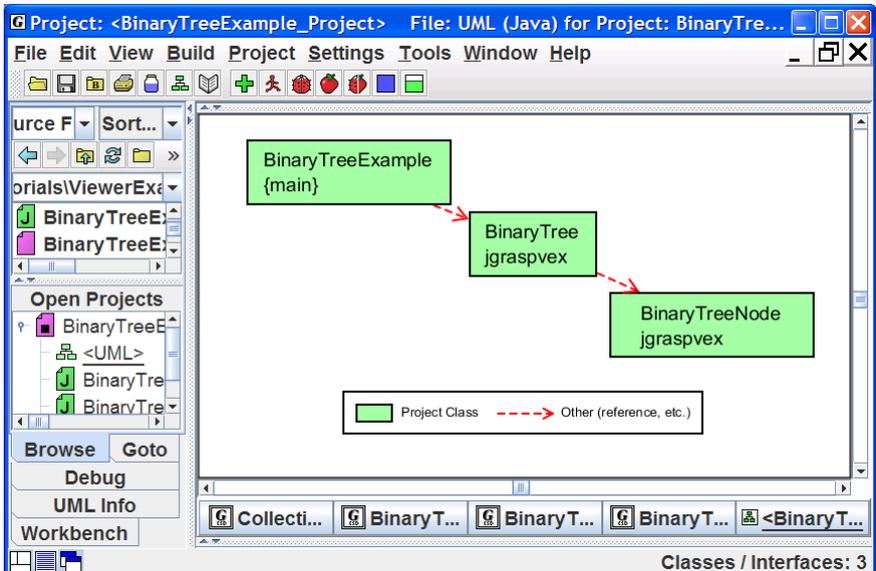


Figure 10-25. UML Class Diagram for BinaryTreeExample_Project

For more information on creating projects and generating UML class diagrams, see *Getting Started with Objects, Projects, and/or UML Class Diagrams*.

Now we are ready to create an instance of BinaryTree. Right-click on the BinaryTree class in the UML diagram as shown in Figure 10-26, then select the second entry on the pop-up list, *Create New Instance*. This brings up the Create New Instance dialog which lists the available constructors for BinaryTree. Figure 10-27 indicates that we are about to create an instance called “jgraspvex_BinaryTree_1” using BinaryTree’s only constructor. When the *Create* button is clicked, the new object is placed on the workbench and listed in the Workbench tab as shown in Figure 10-28. Now let’s open a viewer, as we’ve done before, by selecting and dragging the object from the Workbench tab. Figure 10-29 shows the BinaryTree object in the viewer with size 0. To add elements to the instance, we need to invoke its public *add()* method. Clicking on the Invoke Method button  located in the upper right corner of the viewer brings up the dialog shown in Figure 10-30. To make the dialog stay up so that we can add multiple objects, click on the stick pin  in the upper left corner.

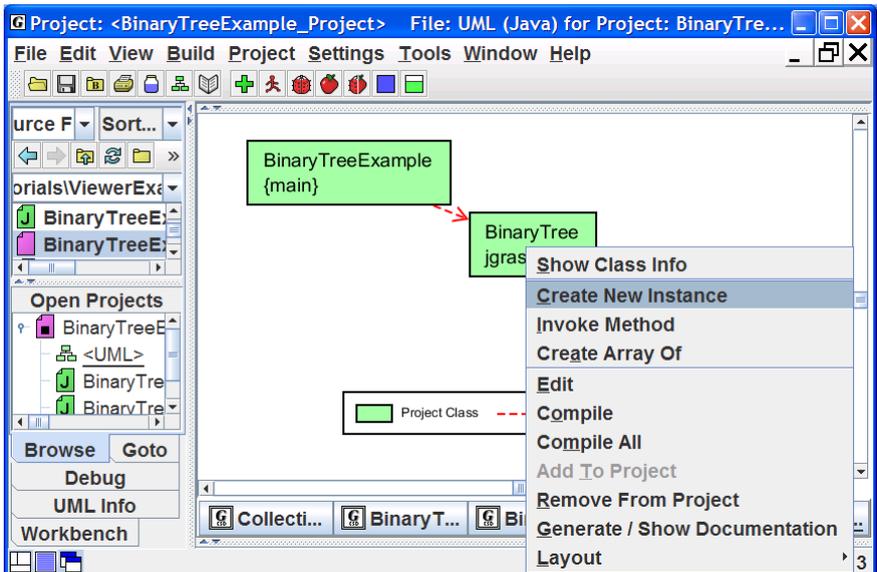


Figure 10-26. BinaryTree class selected to create new instance for workbench

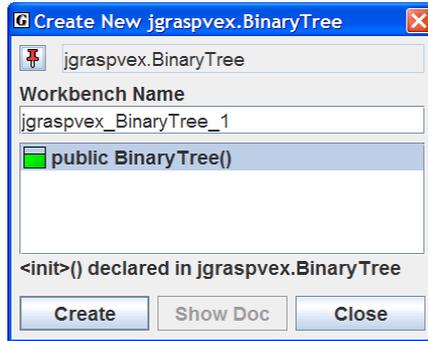


Figure 10-27. Create New Instance dialog for the BinaryTree class

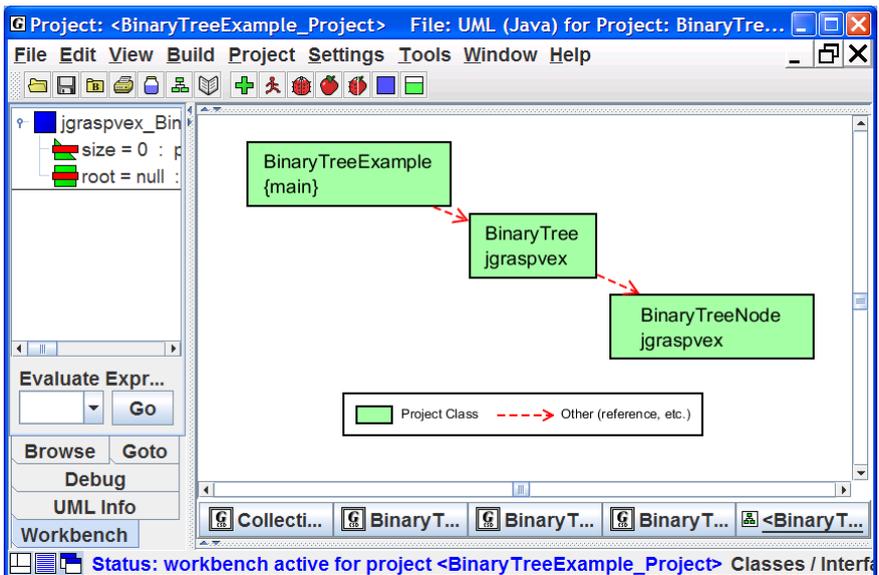


Figure 10-28. BinaryTree object on the workbench (unfolded to show fields)

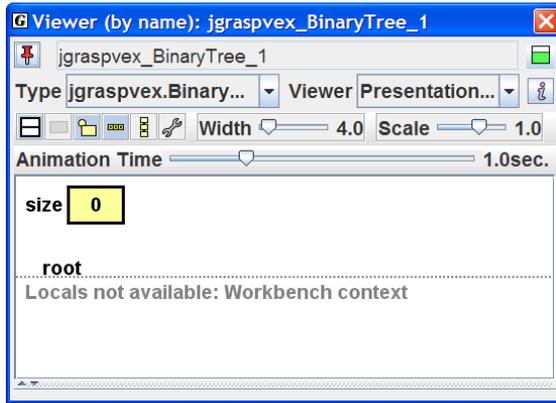


Figure 10-29. Viewer opened on the object `jgraspvex_BinaryTree_1` with 0 elements

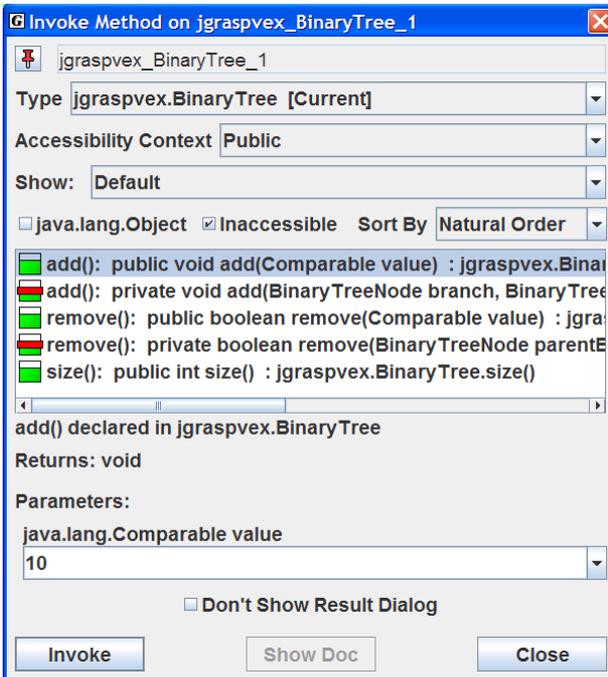


Figure 10-30. Invoke Method dialog for `jgraspvex_BinaryTree_1` to add element 10

Let's add the value 10 to the binary tree by selecting the public `add()` method. If you are using Java 1.5 or higher, you can enter 10 (without quotes) in the parameter box labeled `java.lang.Comparable` value as shown in Figure 10-30. Java's autoboxing feature will convert this to an Integer object. Otherwise enter "10" (with quotes) to make the value a string. Clicking the *Invoke* button will cause the object to be inserted into the binary tree. Notice that the *Result* dialog pops up indicating the invocation was successful.

*To prevent the Result dialog from popping up after each invocation, you can check the **Don't Show Result Dialog** option located above the Invoke button.*

Now let's add each of the following elements using the same steps we used above to add the element **10** to the tree: **8, 12, 6, 9**

As you add each element, you should see the tree adjust to accept the new node. You can increase or decrease the animation time using the slider provided on the viewer. Decreasing the animation time speeds up the movement of the nodes. After adding these elements, your viewer should look similar to the one in Figure 10-31 with five elements.

Now let's remove the node containing 8. On the Invoke Method dialog for *bt*, select the public `remove()` method and enter 8 as the parameter then click the *Invoke* button (see Figure 10-32). The node with value 8 is removed and the tree is adjusted accordingly. Now try adding 8 back to the tree and notice where it ends up.

In workbench mode, local nodes are not available, as indicated by the message in the viewer. However, if you set a breakpoint in the `add()` method and then invoke it, the desktop switches to debug mode and allows you to step through the method, at which time local nodes are displayed as appropriate. As soon as you step to the end of the method, the desktop returns to workbench mode. If you do set a breakpoint in a method that you are invoking from the workbench, remember to remove the breakpoint when you are done. Otherwise, each time you invoke the method in the future, you will have to step through it in debug mode.

In the example above, we created the instance of `BinaryTree` by right-clicking on a class in the UML diagram. This approach assumes that the classes are in a jGRASP project and that a UML class diagram has been generated for it. Since most users spend much of their time reading and writing code in the CSD window, jGRASP provides a convenient way to create instances of a class for the workbench from the CSD window. The section concludes with an example using this method.

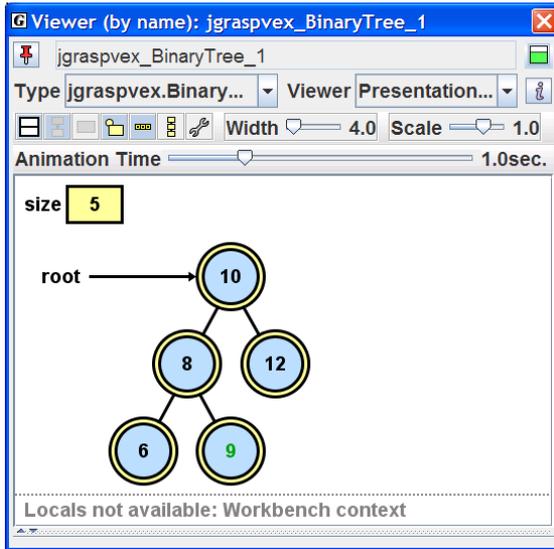


Figure 10-31. Viewer opened on the object `jgraspvex_BinaryTree_1` with 5 elements

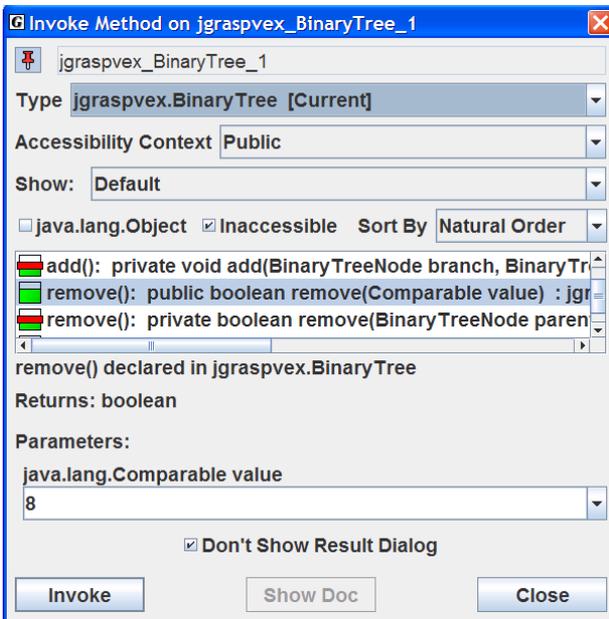


Figure 10-32. Removing element 8

In the Browse tab, navigate to the *ViewerExamples* directory if you are not already there. In this directory, you should see the directory *graspvex* which contains the data structure classes for this tutorial. Find *BinaryTree.java* and double-click on it to open it in a CSD window. Figure 10-33 indicates the location of the *Create Instance*  button on the CSD window tool bar. Clicking this opens the Create New Instance dialog which lists the available constructors for *BinaryTree* as shown above in Figure 10-27.

You can find also create an instance from the menu by clicking **Build > Java Workbench > Create New Instance**. This is illustrated in Figure 10-34.

Regardless of the way you choose to create instances, the workbench provides a convenient way to test a class and its methods without the necessity of a driver program. When a viewer is opened for an instance of a data structure on the workbench, the opportunity for understanding the software is even greater.

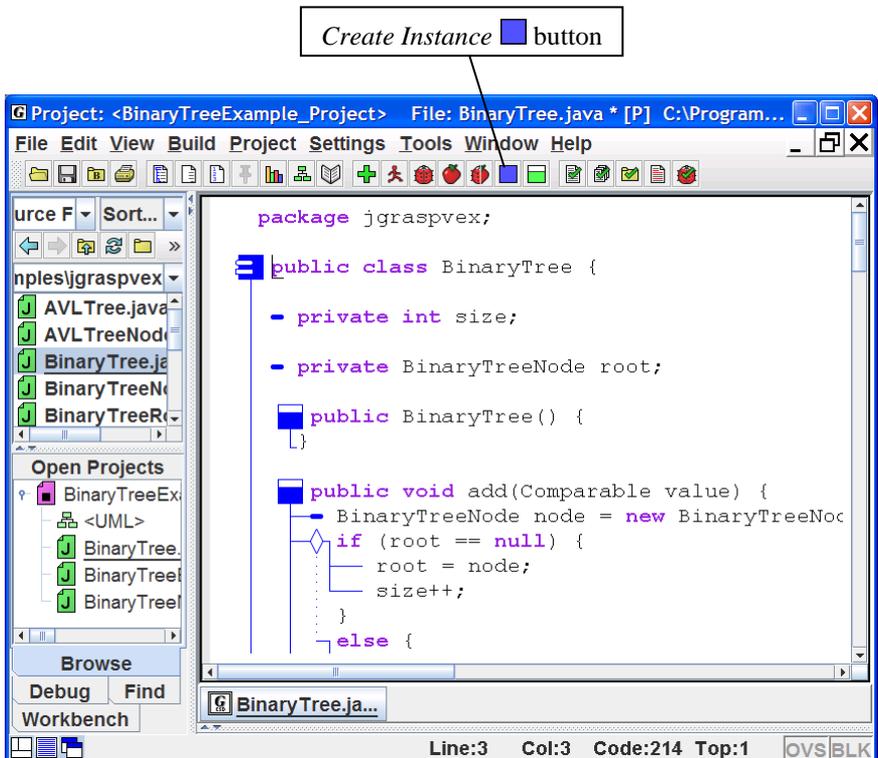


Figure 10-33. CSD window with *BinaryTree.java*

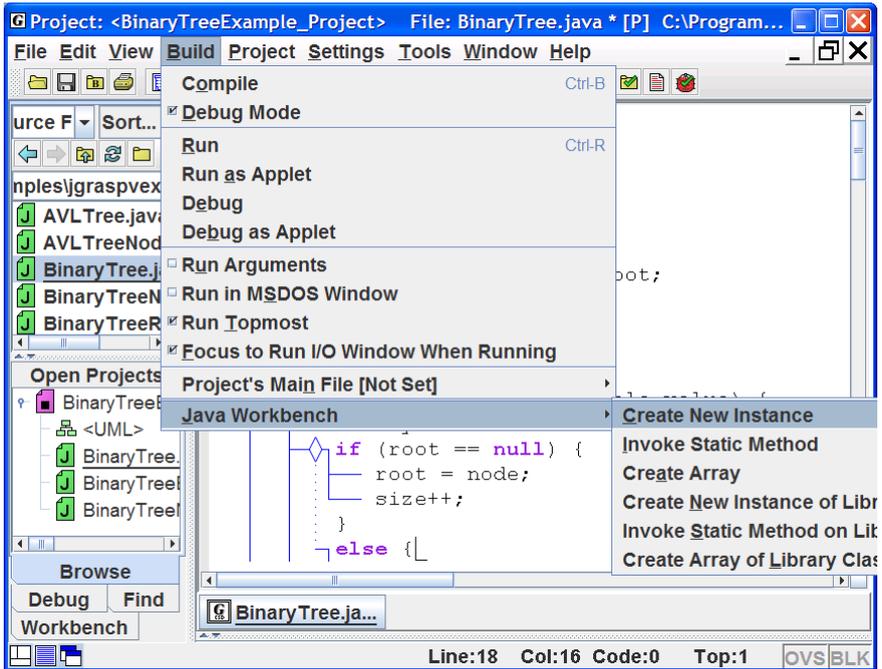


Figure 10-34. Using the Build menu to an create instance

10.8 Summary of Views

During execution, Java programs usually create a variety of objects from both user and library classes. Since these objects only exist during execution, being able to visualize them in a meaningful way can be an important element of program comprehension. Although this can be done mentally for simple objects, most programmers can benefit from seeing visual representations of complex objects while the program is running. The purpose of a viewer is to provide one or more views of a particular instance of an object during execution, and multiple viewers can be opened on the same object to observe different structural properties of the object. These viewers are tightly integrated with the workbench and debugger and can be opened for any primitive, object, or field of an object in the Debug or Workbench tabs. Below is a summary of current views.

General Description of Views

Basic – An object can be unfolded to reveal its fields; if a field is an object, it too can be unfolded to see its fields. This view is used in the debug and workbench tabs, and it is available for all classes.

Detail – For integer (*byte, short, int, long*) and character (*char*) types, the value in decimal, hexadecimal, octal, and binary is displayed. For floating point (*float, double*), the value is represented using the IEEE standard for mantissa and exponent. The *detail* view also works for each associated wrapper class.

Presentation – A conceptual view similar to what one might find in a textbook is provided by a viewer written for a specific class; typically handles very large number of elements efficiently. Currently supported classes include:

array, String, ArrayList, Vector, Stack, LinkedList

Presentation - Structure Identifier – A conceptual view is provided when a structure is automatically detected; typically handles a moderate number of elements efficiently. This view is listed on the *View* drop down list for many objects and if selected, the user has the opportunity to configure the viewer for a linked list or binary tree even if neither was automatically identified. The following structures are currently supported in jGRASP 1.8.7:

linked lists, binary trees (including binary heap, red black trees, AVL trees), hashtables, and array wrappers (lists, stacks, queues, etc.)

10.9 Exercises

- (1) Open *CollectionsExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops at the breakpoint, open a viewer on instances of one or more of the following, then step through the program:
 - a. array
 - b. ArrayList
 - c. LinkedList
 - d. TreeMap
 - e. HashMap
- (2) Continuing with the program from above, let's use the Auto-Step feature of the jGRASP Debugger. With the program stopped at a breakpoint and one or more viewers open, select *Auto Step*  on the debug control panel and click the *Step* . You can control the speed of the steps with the slider bar beneath the step controls.
- (3) Open *QueueExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops the breakpoint, open a viewer on queue. Select *Auto Step*  on the debug control panel. Now click the *Step* button . You can control the speed of the steps with the slider bar beneath the step controls on the debug control panel. You can control the speed of the animation with the slider bar on the viewer. By watching the queue in the viewer as the program executes, what can you learn about the implementation of the queue?
- (4) Open *LinkedListExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops at the breakpoint, open a viewer on list. Select *Auto Step*  on the debug control panel. Now click the *Step in* button .
- (5) Open *BinaryTreeExample.java* and repeat the task described in (4).
- (6) Although *float* and *double* are primitive data types rather than data structures, the IEEE standard representation for floating point types is quite interesting.

Create floating point variable in your program by adding the statement:

```
double myDouble = 4096.0;
```

After compiling the program, set an appropriate breakpoint, and run the program in Debug mode. Open a viewer on `myDouble` and set the view to

Detail. The *Detail* view for *float* and *double* values shows the exponent and mantissa representation used for floating point numbers and how these are calculated.

Change the value of `myDouble` by right-clicking on it in the Debug tab and selecting “Change Value” from the list of options.

Notes