

12 Using the Viewer Canvas

In this tutorial we introduce the viewer canvas, which allows you to create dynamic visualizations of your programs using multiple viewers. The canvas is tightly integrated with the debugger, workbench, and interactions. It provides a conceptual visualization similar to what one might find in a textbook but with the added benefit of being dynamically updated as you step through the program. This allows you to explore the inner workings of your program regardless of its apparent complexity. The visualizations provided by the canvas are useful for general program understanding as well as traditional debugging.

Objectives – When you have completed this tutorial, you should be able to create a canvas by dragging and dropping objects and primitives displayed in the Debug or Workbench tabs, select among viewers for each item on the canvas, set individual viewer options, save the canvas, and run your program in the viewer canvas. The details of these objectives are captured in the hyperlinked topics listed below.

12.1 Introduction

12.2 Creating a Simple Canvas – AcmeDinnerTheater Program

12.2.1 Running in a New Canvas and Adding Variables

12.2.2 Running in a Saved Canvas

12.3 Viewing an Array on the Canvas – BinarySearchExample

12.3.1 Understanding the BinarySearch method

12.3.2 Compiling and Running the BinarySearchExample

12.3.3 Controlling the Viewer Canvas

12.4 Creating a Canvas with Array Viewer – BinarySearchExample

12.4.1 Opening a Second Canvas Window

12.4.2 Saving the Canvas File

12.4.3 Adding Variables to the New Canvas

12.4.4 Playing the New Canvas

12.4.5 Adding Index Expressions to the Array Viewer

12.4.6 Changing Rotation, Width, and Scale in the Presentation Viewers

12.4.7 Experimenting with Different Array Viewers

12.5 Sorting on the Canvas – NumberSelectionSort

12.5.1 Reviewing the Source Code






12.5.2 Running the NumberSelectionSort in the Canvas

12.6 Wrapping Up – Notes on Using the Canvas









12.7 Looking Ahead to Data Structures – Exercises


12.1 Introduction



After you successfully Compile  your program, you have three ways to run your program in jGRASP: Run , Debug , and Run in Viewer Canvas . In this tutorial, we focus on Run in Viewer Canvas , which opens a canvas window on a new or existing canvas file. When any primitive, object, or field of an object in the Debug or Workbench tabs is dragged onto the canvas, a viewer window is opened using a viewer associated with the variable type. Below is a summary of the basic steps for creating a canvas for your program. These will be explained in detail in the sections below using example programs.

General steps for creating and using a new viewer canvas with your compiled program:

- (1) On the desktop toolbar, click the Run in Canvas button .
- (2) Click the Step button  on the canvas window or debug tab until you see variables of interest in the Variables tab.
- (3) Drag one or more variables onto the canvas; a default viewer should open for each variable.
- (4) Save the canvas . [NOTE: To associate the canvas file with this program, the name must begin with the Java file name that contains main or with the name of the jGRASP project (e.g., program *MyProgram.java* has canvas *MyProgram.jgrasp_canvas.xml*). This naming convention enables “Run in Canvas”  to use the appropriate canvas file when launching the program.]
- (5) Step through the program and observe the object in the viewer.
- (6) On the canvas, click the Play button  (auto step-in) on the canvas to start the visualization. Use the Pause button  and Stop button  as needed. To regulate the speed of the program, decrease or increase the delay between steps using the Delay slider. 

You can also use the canvas via the Debug or Workbench tabs by clicking the Open New Viewer Canvas  button on the debug or workbench toolbar and then dragging one or more variables onto the canvas. Changes to these variables resulting from statements executing in the Interactions tab, from stepping the debugger, and/or from executing methods via the Invoke Method dialog will be reflected on the canvas.

We'll use three example programs (AcmeDinnerTheater, BinarySearchExample, NumberSelectionSortExample) to illustrate the basic steps above and to demonstrate the overall utility of the canvas. Exercises will then guide you through several data structure examples.

12.2 Creating a Simple Canvas – AcmeDinnerTheater Program

[NOTE: If you have not already done so, you should copy the jGRASP examples to a personal folder as follows. Start jGRASP, then on the main menu click **Tools > Copy Example Files**. In the dialog, select a folder as appropriate and click **Choose** to save the jGRASP examples. This will open the Browse tab on the jgrasp_examples folder containing the copied files.]

Let's begin by opening one of the programs in the examples folder 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. In the Tutorials folder, you should see a folder called CanvasExamples. Double-click on the folder to open it then locate the file *AcmeDinnerTheater.java*. Open this file by double-clicking the file name (Figure 12-1).

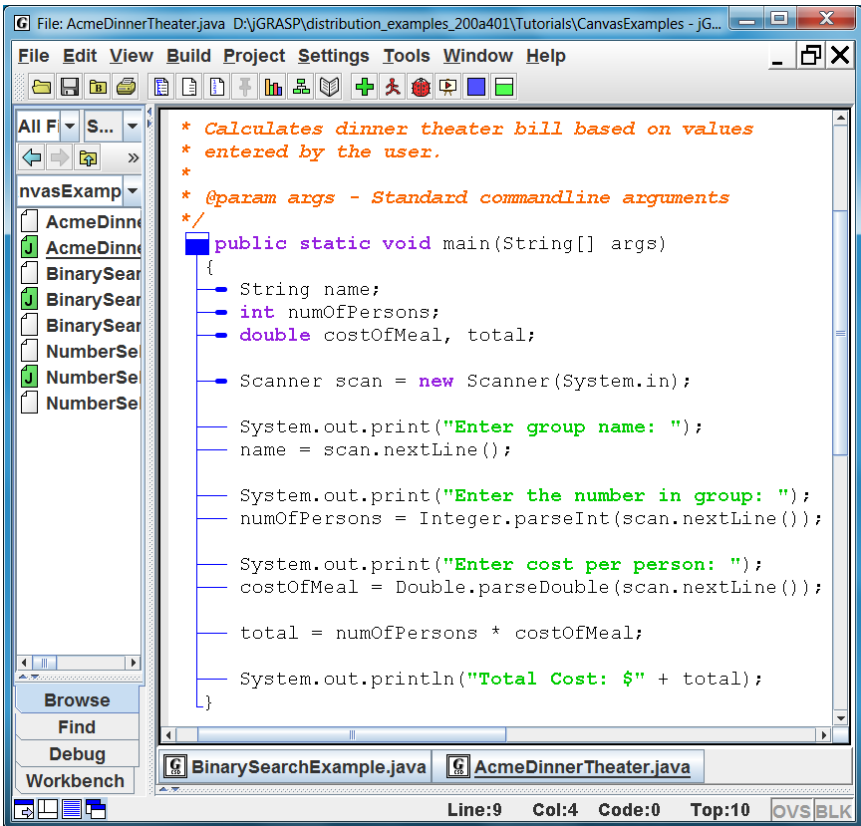


Figure 12-1. *AcmeDinnerTheater.java*

A quick review of the program shows that it creates a Scanner object and then reads in a String for the group name, an *int* for number in the group, and a *double* for the cost per person.

12.2.1 Running in a New Canvas and Adding Variables

Let's compile the program by clicking the green plus **+**. The Compile Messages tab pane should indicate that the Java compiler (javac) was launched and that the operation was completed. If no compile errors are indicated then compilation was successful and you are ready to run the program.

Click the Run button **R** to run the program. This program queries the user for each of the three input values: group name, number in group, and cost per person. For example, if you entered Pat Brown, 6, and 49.99, after the program ends, you should see the following in the Run I/O tab pane.

```

----jGRASP exec: java AcmeDinnerTheater

Enter group name: Pat Brown
Enter the number in group: 6
Enter cost per person: 49.99
Total Cost: $299.94

----jGRASP: operation complete.

```

Now we are ready to run the program in the viewer canvas. This will allow us to examine some of the details of the program while it is running. Click the Run in Viewer Canvas button **R** which is located on the desktop toolbar. This runs the program in debug mode and stops at the first executable statement. Since this program does not yet have a canvas file associated with it, an empty canvas window is opened (Figure 12-2).

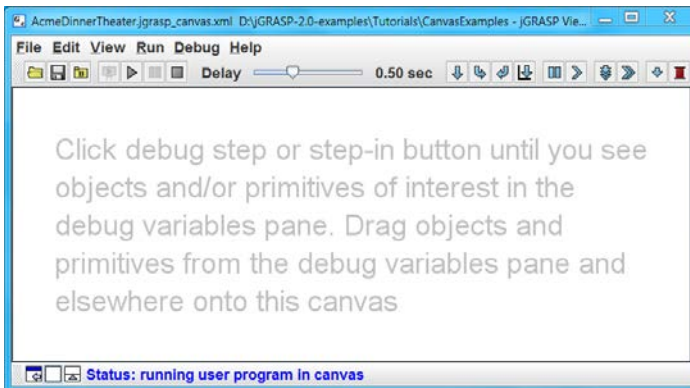


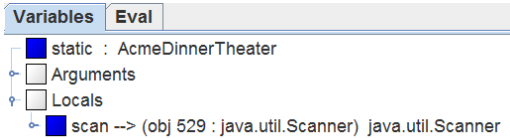


Figure 12-2. Empty canvas window

As the message in the canvas window indicates, we now need to click the Step button  on the canvas window or debug tab until we see variables of interest in the Variables tab. In the CSD window, you should see that the program is stopped on the line that creates the Scanner object. After you click the Step  button, a new Scanner object called *scan* should appear in the Variables tab of the Debug pane.



Using the mouse, drag the variable *scan* onto the canvas window and a Presentation viewer is opened for *scan* as shown below. Note that the Scanner buffer is currently empty (i.e., the pointer is at the 0 position).

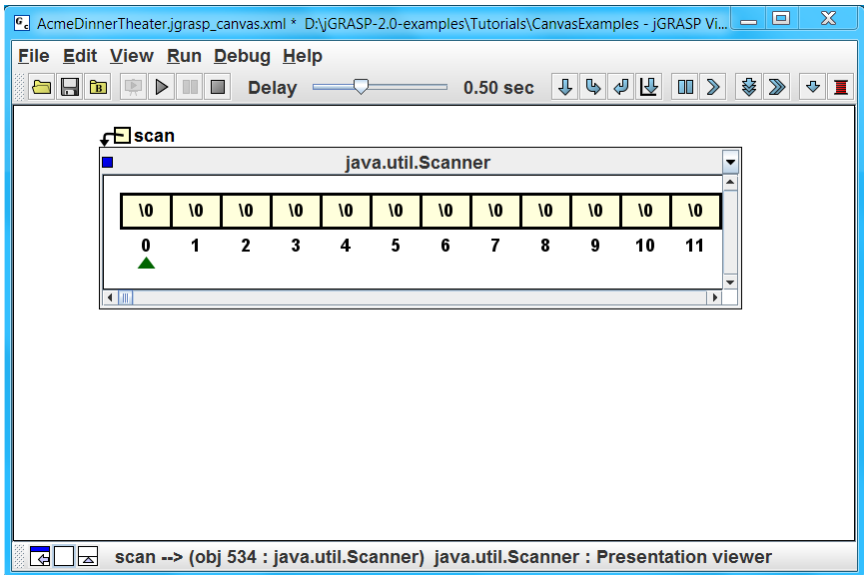






Figure 12-3. Canvas window after the variable scan has been added

With the viewer selected, as it is when initially dragged onto the canvas, you can move it around the canvas by moving the mouse to the top border of the viewer frame and then dragging the viewer. You can make a copy of the viewer by dragging the blue square  on the left end of the top border. To remove a viewer from the canvas, click the Menu button  on the right end of the top border, and select **Remove**. To deselect the viewer, simply click elsewhere in the canvas window and the viewer frame will be hidden.

Now let's continue stepping  in the program to enter the values for *name*, *numOfPersons*, and *costPerPerson*. After you step through a statement requesting input, "Status: waiting for input" will be indicated at the bottom of the canvas and jGRASP desktop. You need to enter the input in the Run I/O tab pane as you did earlier when you ran  the program in the traditional way. If you attempt to step without entering the requested input, jGRASP will pop up an error dialog (shown below) indicating that you "can't step". If this happens, click the OK button on the dialog and then enter the input in the Run I/O tab.

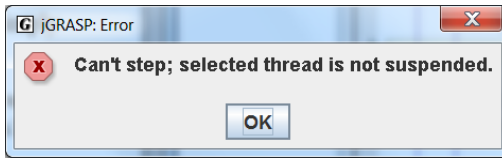

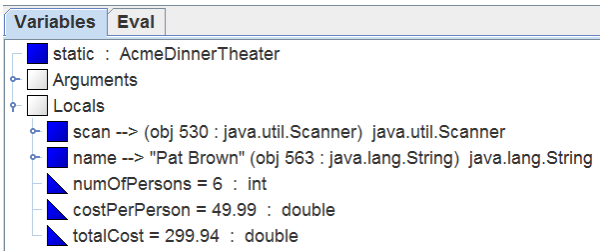


Figure 12-4. "Can't step" error when waiting for input

After entering the input and pressing the Enter key, you should see the text that you entered in the Scanner object on the canvas. If you select the Scanner viewer, you can scroll it to see all of the values entered thus far. Notice the '\r' and '\n' characters that follow each value you entered. These are the "return" and "line feed" control characters that resulted from pressing the Enter key. The line terminators on non-Windows systems may be different from \r\n. Since we are scanning an entire line using the *nextLine()* method, the token marker is pointing to the character after the '\r' and '\n' characters in the Scanner buffer.

Continue stepping  and entering input until you get to the last print statement in the program. You should now see all of the variables in the Debug tab pane.



Now drag all of the remaining variables (*name*, *numOfPersons*, *costPerPerson*, *totalCost*) onto the canvas. As each variable is added to the canvas, the default viewer for its type is opened. For *name*, which is a String, the Formatted viewer is opened. For the other variables, which are of type *int* and *double*, the Basic viewer is opened. The Basic viewer displays the result of *String.valueOf()* for primitives.

Now that all of the variables are on the canvas, this would be a good time to make any adjustments with respect to a variable's specific viewer or its position on the canvas. Figure 12-5 shows one way to arrange the four variables using their default viewers.

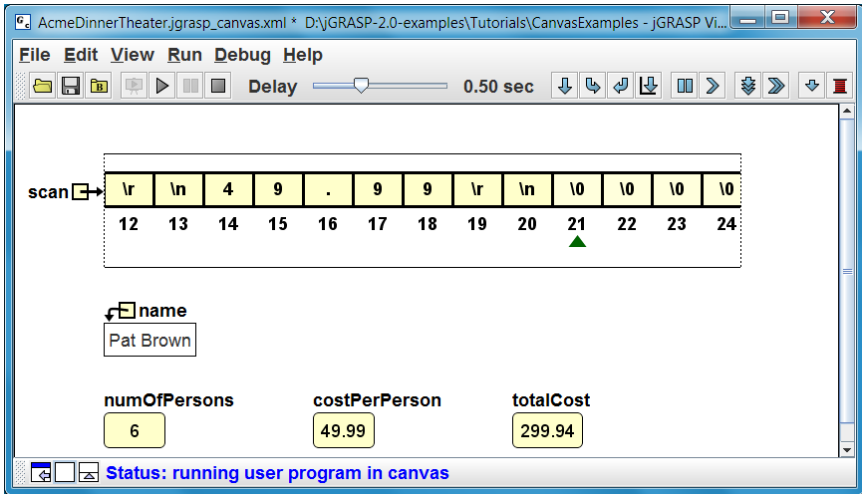


Figure 12-5. Canvas after all variable have been added

You can experiment with different viewers for each of the values. For example, instead of the Formatted viewer for *name*, you may want to use the Presentation String viewer. To change the viewer for *name*, select the viewer window, click the Menu button \blacktriangledown in the upper right corner of the viewer frame, select **Viewers**, then select **Presentation String** as shown below in Figure 12-6.

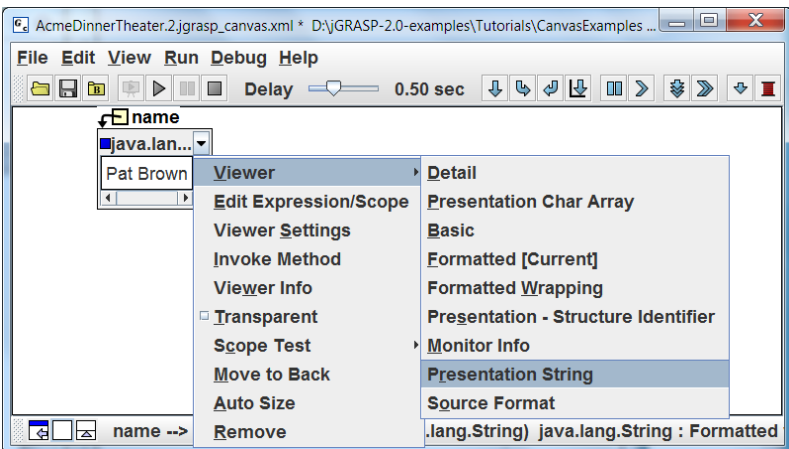
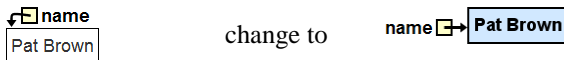



Figure 12-6. Changing the viewer for name to Presentation String

When the viewer for *name* is changed from the Formatted viewer to the String Presentation viewer, you should see:





After changing the viewer, if the label “name” is not fully visible on the canvas, you may need to reposition the viewer window on the canvas by selecting it and dragging the top border.

12.2.2 Running in a Saved Canvas

We are now ready to save the canvas by clicking the Save button  on the canvas toolbar. This saves the file as *AcmeDinnerTheater.jgrasp_canvas.xml* which you should see at the top of the canvas window. If you attempt to close a canvas window or exit jGRASP before saving the canvas after any changes have been made, a warning dialog will pop up and allow you to save and close the file, discard the edits, or cancel.

Continue stepping through the program until you reach the end of the main method. You should see the message “Status: run in canvas ended” at the bottom of the canvas window.

 Now that we have a canvas file for the program, let’s run the program again in the canvas. Click the Run in Viewer Canvas button  on the canvas window toolbar. As before, this runs the program in debug mode and stops at the first executable statement. Now we see “Status: running user program in canvas” at bottom of the canvas (Figure 12-7).

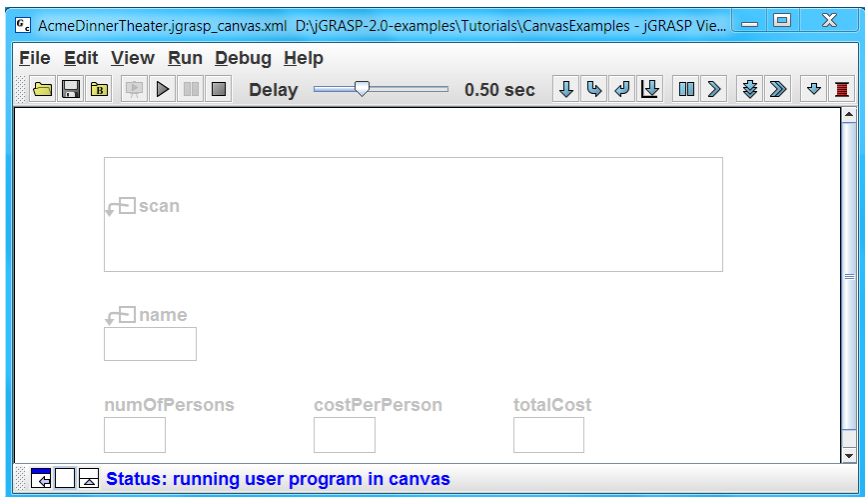








Figure 12-7. Running in canvas, stopped at first executable statement





 Previously, we used the Step button  to step through the program. This time let's use the Play button  since we have already created a canvas for our program. The Play button turns on Auto-Step in the debugger and then begins stepping. This means we do not have to click the Step button repeatedly. After clicking the Play button, each viewer on the canvas will become active when its associated variable is first assigned a value.

After you have entered the last value for *costPerPerson*, you should see the message “Status: run in canvas ended” at the bottom of the canvas window as shown in Figure 12-8.

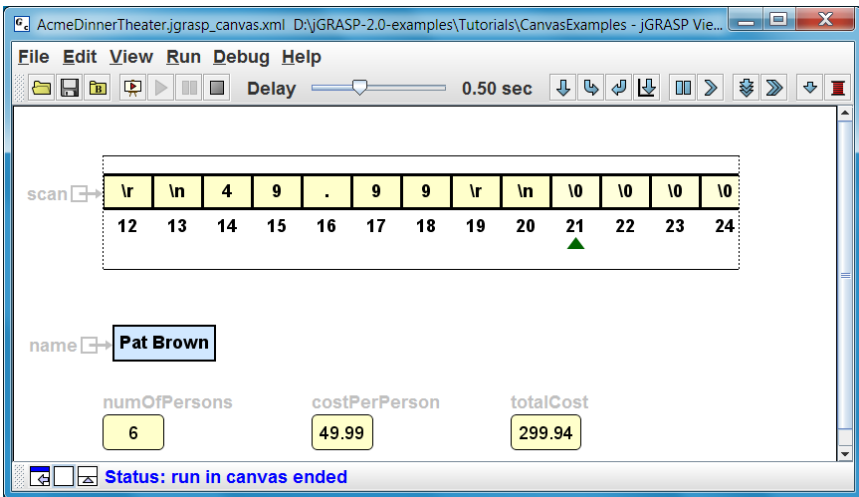




Figure 12-8. Canvas after all input has been entered and run has ended

As a final exercise before we leave this example, let's close the canvas window. Now click the Run in Viewer Canvas button  on the desktop toolbar. This opens the canvas window associated with the *AcmeDinnerTheater.java* and stops at the first executable statement. After clicking the Play button , the program begins auto-stepping, requesting input, and proceeding through the program as it did above.

This completes our first example. While this was a simple example, it served to illustrate the steps of creating a canvas by dragging variables onto the canvas and then stepping or playing the program. The next section will introduce additional features of the viewers and viewer canvas.

12.3 Viewing an Array on the Canvas – BinarySearchExample

Now let's look at a program that includes an array. Since our canvas for this program will be slightly more involved than the one for the AcmeDinnerTheater program, it will allow us to explore several important features that we didn't need in the first example. We'll begin by opening the BinarySearchExample program and discussing the details of the *main* method and *binarySearch* method. After compiling the program, we'll run it in the traditional way, and then we'll run it in an existing canvas. After this, we'll explore different ways to control the canvas and gain some experience using the canvas.

In the ViewerExamples folder, locate *BinarySearchExample.java* and double-click on it to open it in a CSD window. The BinarySearchExample class contains two static methods: *main* and *binarySearch*. Figure 12-9 shows the *main* method which creates and initializes *int* array *ia* with the values 12, 34, 56, 65, 73, 81, and 97. Then in successive print statements, the *main* method calls the *binarySearch* method to find the location or index of various values in the array. For example, in the first print statement, the *binarySearch* method is called to find the index of value 12 in array *ia*. Since 12 is the first value in the array, we expect our program to print the following.

```
Index of 12 is: 0
```

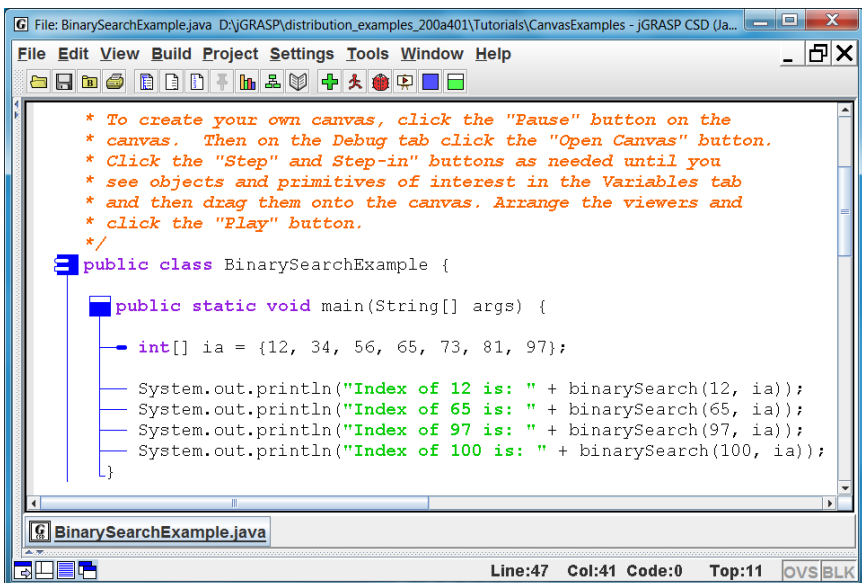


Figure 12-9. The main method in BinarySearchExample program

12.3.1 Understanding the `binarySearch` method

Now let's take a look at the details of our `binarySearch` method shown in Figure 12-10, which takes two parameters, `int key` and `int[] intArray`. It searches `intArray` for the value specified by `key` and returns the index of the value if found or -1 if not found. Note that a linear search works with an array of elements in an arbitrary order, whereas a binary search requires that the elements in the array be ordered or sorted. For a linear search, we would begin searching at the first location and continue searching until the `key` element is found or the end of the array is reached. However, the strategy in our binary search is to set `low`, `mid`, and `high` to the first index, middle index, and last index of the array respectively, and then compare the `key` with the value at `mid` in the array. If `key` is less than the value at `mid`, we know the key must be in the first half of the array so we can disregard the second half by setting `high` to `mid - 1`. If `key` is greater than the value at `mid`, we know the key must be in the second half of the array so we can disregard the first half by setting `low` to `mid + 1`. If the `key` matches the value at `mid`, we're done. We continue this process until `key` is found or until `low` is greater than `high`. If `key` matches the value at `mid`, then `mid` is returned; otherwise, -1 is returned to indicate that the value was not found.

```

public static int binarySearch(int key, int[] intArray) {
    int low = 0;
    int high = intArray.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (key < intArray[mid]) {
            high = mid - 1;
        }
        else if (key > intArray[mid]) {
            low = mid + 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}

```

The screenshot shows the code in a window titled "BinarySearchExample.java". The code is as follows:

```


public static int binarySearch(int key, int[] intArray) {
    int low = 0;
    int high = intArray.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (key < intArray[mid]) {
            high = mid - 1;
        }
        else if (key > intArray[mid]) {
            low = mid + 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}


```

The IDE interface includes a menu bar (File, Edit, View, Build, Project, Settings, Tools, Window, Help), a toolbar with various icons, and a status bar at the bottom showing "Line:19 Col:1 Code:0 Top:38 OVSBLK".

Figure 12-10. The static `binarySearch` method

12.3.2 Compiling and Running the BinarySearchExample

Let's compile the BinarySearchExample program by clicking the green plus . The Compile Messages tab pane should indicate that the Java compiler (javac) was launched and that the operation was completed. If no compile errors are indicated then compilation was successful and you are ready to run the program.

Click the Run button  to run the program and you should see the output below for the array created in *main* with the following values: 12, 34, 56, 65, 73, 81, and 97.


```

----jGRASP exec: java BinarySearchExample

Index of 12 is: 0
Index of 65 is: 3
Index of 97 is: 6
Index of 100 is: -1

----jGRASP: operation complete.

```

Now let's run the program in the canvas to see a visualization of exactly how the binary search works. We'll begin by using the canvas file that was provided with the program (BinarySearchExample.jgrasp_canvas.xml). Click the Run in Viewer Canvas button  on the desktop toolbar. This opens the canvas window, launches the debugger, and stops at the first executable statement. In the canvas, you should see the outlines of viewers for *key*, *low*, *mid*, *high*, and *intArray* as shown in Figure 12-11.

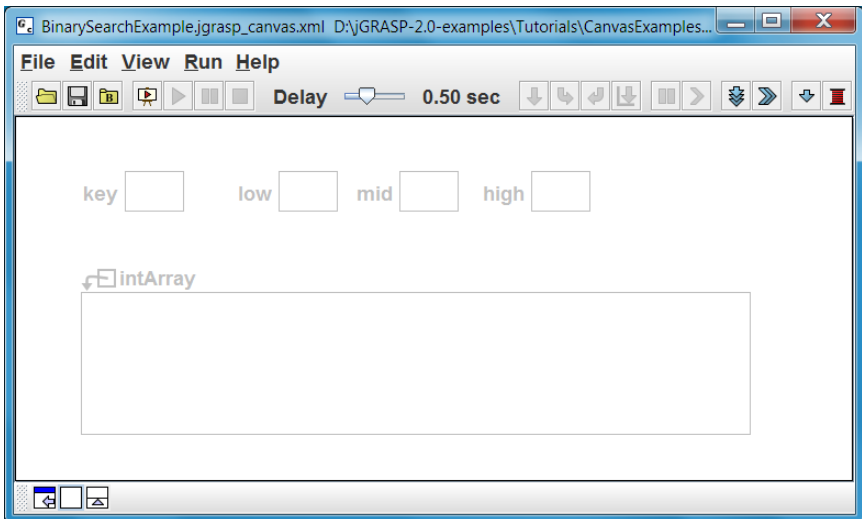



Figure 12-11. The canvas with outlines of viewers to be opened

After clicking the Play button , the canvas begins auto-stepping through the program, and when it steps-in at the first print statement where the `binarySearch` method is called, each viewer on the canvas should become active as the variables are created and initialized as shown in Figure 12-12. Depending on the resolution of your screen, you may need to resize the canvas window and/or the `intArray` viewer. This is done by selecting the item and then dragging a corner or border.

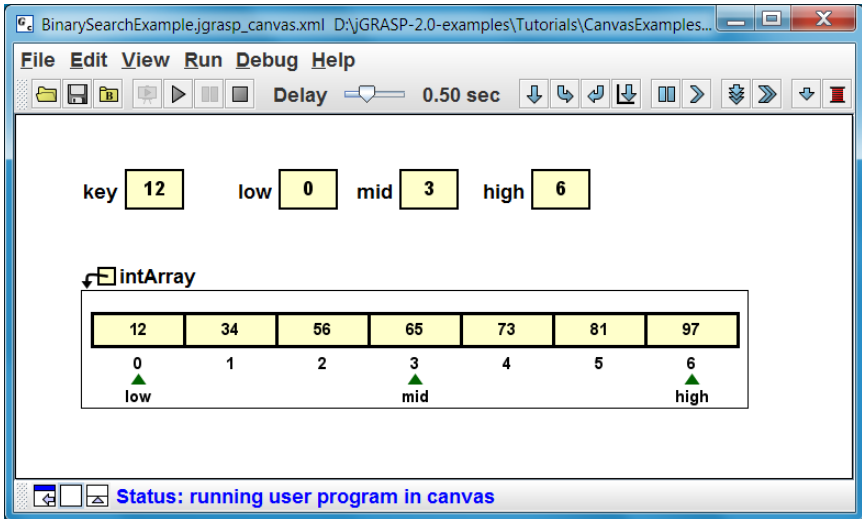

















Figure 12-12. The canvas paused after all variables have been initialized

12.3.3 Controlling the Viewer Canvas

   Any time you are playing  the canvas, you can pause  or stop  the canvas as needed. After clicking the Pause button , you can resume by clicking the Play button . If you click the Stop button , the program ends, and you can click the Run in Viewer Canvas button  and then the Play button  to see the visualization again. These buttons, which provide a convenient way to control the program when running in canvas, are essentially a combination of debug control operations.

-  Turn *Auto Step* on and *Step In*
-  Turn *Auto Step* off
-  Stop running and turn *Auto Step* off







Delay  0.30 sec You can control the speed of the steps in the

program with the *Delay* slider. The default delay between each step is 0.5 seconds. While the program is running in the canvas, try moving the slider to the left to speed up the program or to the right to slow it down.







Although the Play, Pause, and Stop buttons described above provide a general level of control over the canvas, the Debug control buttons provide an additional level of control. Often you will need this more fine-grained control while running a program in the canvas. The Debug control buttons, which are available on both the Debug tab pane and the Canvas window, are shown below.







These buttons can be used to control the stepping in the program when running in the debugger.


-  Step or execute the current statement.
-  Step-in to a method, if any, in the current statement; otherwise Step.
-  Step-out of a method, if any; otherwise run to completion.
-  Run or Step to cursor.
-  Suspend current thread (pause).
-  Resume to next breakpoint





The Debug buttons below toggle the indicated setting on/off.

-  Auto Step or step repeatedly when Step  is clicked.
-  Auto Resume or resume to next breakpoint repeatedly when Resume  is clicked.
-  Use bytecode size steps. [Rarely used with canvas.]
-  Suspend new threads. [Rarely used with canvas.]

You should experiment with the debug controls above. Since the `binarySearch` method is quite short, let's try the following. If the canvas is still running, end the program by clicking the Stop button . Click the Run in Viewer Canvas button  and then click the Step-in button  repeatedly until you have stepped-in to the `binarySearch` method. Now with each click of the Step button , try to relate the results of executing the statement to the program visualization in the canvas. If you are not quite sure how the `binarySearch` method works, this activity should help to clarify it. You may need to repeat this process in order to fully understand the program. If you feel that you already understand the `binarySearch` method, then you can use this activity to

confirm your understanding of details of the method. This is a form of visual verification or interactive code inspection that can be used along with testing to confirm that a program is working exactly as intended.

Let's now set a breakpoint inside the `binarySearch` method on the line where the *if* statement begins. To do this hover the mouse over the margin to the left of the statement until you see the Breakpoint symbol , then left-click the mouse. Alternatively, you can right-click on the line where the *if* statement begins and select "Toggle Breakpoint" to set the breakpoint.

If the program is currently running, end the program by clicking the Stop button . Now click the Run in Viewer Canvas button  then click the Play button . After the debugger stops at the first executable statement, click the Resume button . This tells the debugger to execute all the steps in the program until it reaches the breakpoint where it stops as shown in Figure 12-13.

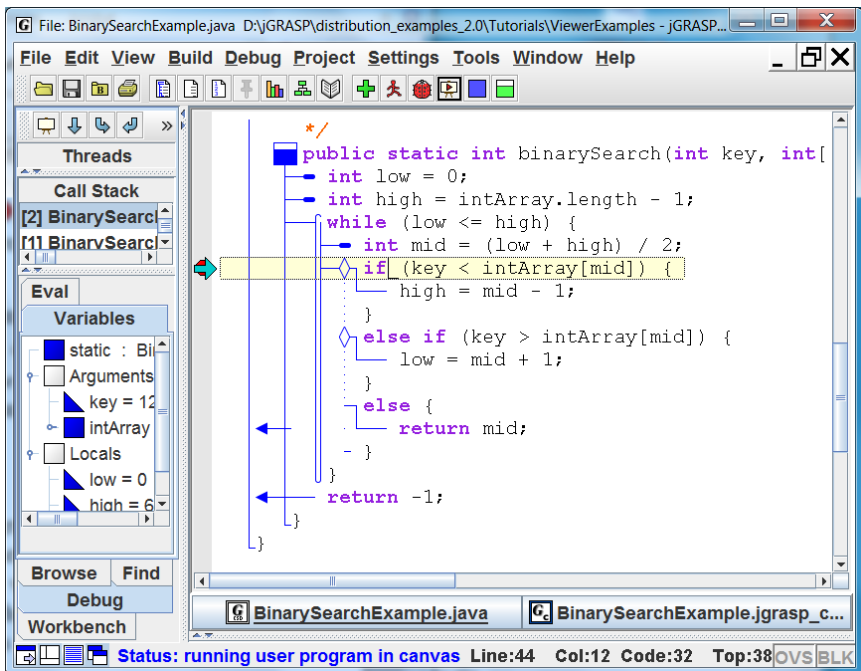



Figure 12-13. Stopped at breakpoint in the `binarySearch` method



Note that even though we have been running this program in the canvas using the debugger, we were not attempting to find bugs. Rather we were simply making sure that we understand the program. However, if we were working on a program that did have problems, using the canvas with the debug controls described above is an excellent way to diagnose the problems and isolate the

bugs causing those problems. That is, you would set a breakpoint on or before the statement where you suspect the problem, run the program in the canvas, and then Resume to the breakpoint. You could also use the debugger directly by clicking the Debug button , which would stop at the first breakpoint.

12.4 Creating a Canvas with Array Viewer – BinarySearchExample

Thus far, we have discussed how to run the BinarySearchExample program using the canvas file (BinarySearchExample.jgrasp_canvas.xml) that was provided with the program file in the ViewerExamples folder. Now comes the fun part – we’ll create a second canvas that is similar to the first. That is, we want to go through steps of how the first canvas file for BinarySearchExample was created, and in the process we’ll create a second canvas file. For a complex program, it is common to have multiple canvas files. This will allow you to gain some important experience in configuring a viewer for an array.

12.4.1 Opening a Second Canvas Window

The easiest way to open a second canvas window for the BinarySearchExample program is to do so while running the program in the first canvas window, so let’s begin by clicking the Run in Viewer Canvas button . After the program stops at the first executable statement, click **File > New Canvas on the canvas window menu**. Alternatively, you can click the Open New Canvas button  at the top of the Debug tab pane and select the “[New] . . .” entry in the Choose Canvas File dialog. Either of these options will open a second canvas window that’s ready for you to drag variables onto it.


12.4.2 Saving the Canvas File

Before we add the variables to our new canvas, let’s briefly discuss the naming convention required to associate the canvas file with the program. The default names for two canvas files associated with our program are as follows:

```
BinarySearchExample.jgrasp_canvas.xml
BinarySearchExample.2.jgrasp_canvas.xml
```

These file names begin with the name of the Java class, *BinarySearchExample*, that contains the *main* method, and they end with *jgrasp_canvas.xml* which identifies the file as a jGRASP canvas file. The “2” in the second file name, which is used to make the name unique, can be any sequence of characters that are legal in a file name. For example, the following file names can also be used to associate the canvas file with the BinarySearchExample program.

```
BinarySearchExample.my_second_canvas.jgrasp_canvas.xml
BinarySearchExample.temp.jgrasp_canvas.xml
```


Now let's save our new canvas file by clicking the Save Canvas button  on the canvas toolbar. This opens the Save Canvas File As dialog with the file name (BinarySearchExample.2.jgrasp_canvas.xml) already filled in as shown in Figure 12-14. Click the **Save** button on the dialog to save the file and close the dialog. BinarySearchExample.2.jgrasp_canvas.xml should be displayed at the top of the canvas window.

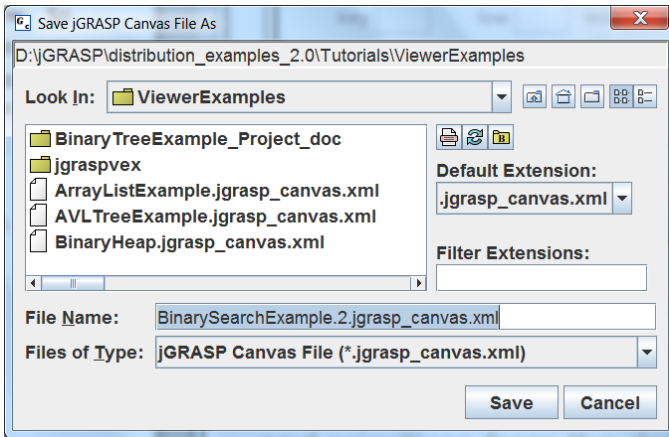





Figure 12-14. Save canvas dialog with default file name

12.4.3 Adding Variables to the New Canvas

In order to add variables to the canvas, we need to be able to see them in the Variables tab of the Debug tab pane. This means we need to let the program step to the point where the variables of interest have been created. Since this is a small program, simply click the Step-in button  twice, which should step-in to the `binarySearch` method. At this point you should be able to see the variables `key` and `intArray` in the Debug tab pane. Drag `key` onto the new canvas and drop it in the upper left of the canvas. This should open a viewer for `key` and show its initial value of 12. Now de-select the viewer by clicking elsewhere in the canvas and you should see the variable name `key` move from above the viewer to the left of the viewer. Remember that you can always select this viewer and reposition it on the canvas as needed. You should have the first canvas window open so that you can use it as a guide for laying out the new canvas.

Now drag the variable `intArray` from the Debug tab and drop it onto the lower left portion of the new canvas. We'll come back to this viewer and configure it later, but for now, let's get the other variables onto the canvas.

To create the remaining variables of interest, click the Step button  four times (i.e., down to the `if` statement). Now drag the variables `low`, `mid`, and `high` onto the canvas and position them approximately as they are in the first canvas. Your

new canvas should now have all of the variables of interest and appear similar to the canvas in Figure 12-15. Note that the default Basic viewer is used for *key*, *low*, *mid*, and *high* rather than the Presentation String viewer above. This would be a good time to save the canvas again by clicking the Save Canvas button .

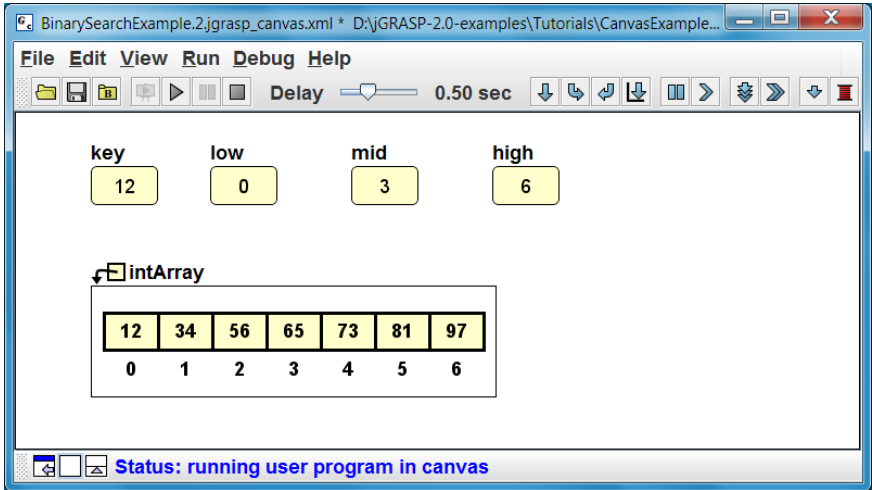




Figure 12-15. New canvas with variables added

12.4.4 Playing the New Canvas

Now click the Play button  at the top of the new canvas window. The program should begin stepping, and you should see both the original and new canvas windows being updated. If you move the Delay slider on one of the canvas windows, you should see the Delay slider on the other canvas window change as well. Since both canvas windows are associated with the same program, when you run the program in one canvas, it will also be running in the second canvas. Normally, if you have two canvas windows open for the same program, you would have different viewers and/or variables in the two canvas windows so that you could observe different aspects of the program as it runs.

12.4.5 Adding Index Expressions to the Array Viewer

By now, you have no doubt observed that the array viewer in the new canvas does not match the one in the original canvas. It does not display *low*, *mid*, and *high* with the array indexes, and the array cells are not as wide as the array cells in the original canvas window.

Let begin by adding the variables *low*, *mid*, and *high* as the index expressions for the array. Click the Menu button  in the upper right corner of the viewer frame and select the **Add Index Expressions** option. In the dialog, shown below in Figure 12-16, enter *low#mid#high* as index expressions and click OK to

close the dialog and apply the settings. When you play the canvas, you should now be able to see the variables *low*, *mid*, and *high* move and update when they are in scope.

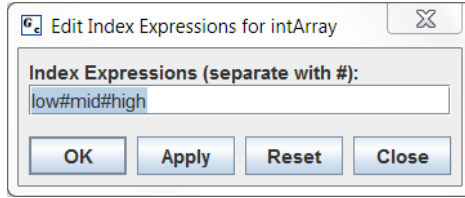


Figure 12-16. Adding index expressions for variables *low*, *mid*, and *high*

12.4.6 Changing Rotation, Width, and Scale in the Presentation Viewers

Select the viewer for *intArray*, to make the viewer frame visible and then click the Menu button ▼ in the upper right corner of the viewer frame and select the **Viewer Settings** option. This opens the Settings for *intArray* dialog as shown in Figure 12-17.

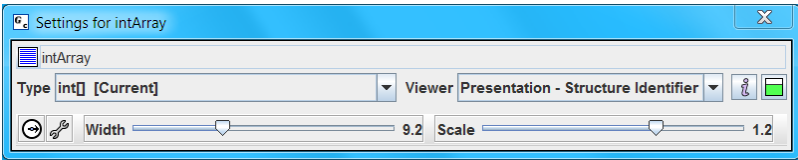



Figure 12-17. Viewer Settings dialog for *intArray*

First let's rotate the array viewer by clicking on the Rotate View button . Successive clicks rotates the array viewer down (elements top to bottom), left (elements right to left), up (elements bottom to top), and right (elements left to right). The latter is the default rotation.

Now let's make the array cells wider in the new canvas. Locate the Width slider bar and increase the width of the array cells by dragging the slider to the right. If the array contained String objects, this would allow you to adjust the width so that more of each string is visible. Now drag the *Scale* slider to adjust the scale of the entire array. Depending on the width and scale that you choose, you may need to resize the viewer frame by dragging the right border of the viewer to the right far enough to ensure that all the array elements are visible. Note that if you have arrays in your programs with many elements you will likely need to scroll the viewer rather than resize it. When a viewer is initially opened is set to Auto Size. If you manually resize the viewer, you can reset it to Auto Size by clicking the Menu button ▼ in the upper right corner of the viewer and then

selecting **Auto Size**. In Figure 12-18, *intArray* has been rotated the “Up” position, which shows the elements from bottom to top, the width has been set to 9.2, and the scale has been set to 1.2 as indicated in the Viewer Settings dialog (Figure 12-17). This rotation would be quite appropriate for an array that represented a stack.

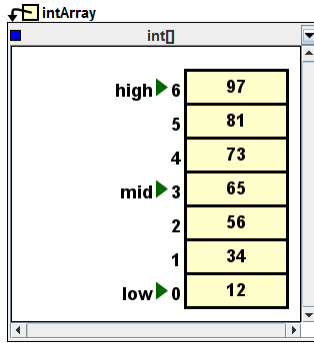







Figure 12-18. *intArray* after rotating “Up” and adjusting Width and Scale using the Viewer Settings dialog.

Now that you have “adjusted” the settings for the new viewer, close the Viewer Settings dialog, and then click the Save Canvas button  to save all the viewers and settings for the canvas. Click the Run in Viewer Canvas button  and then the Play button  to see the program visualizations in both canvas windows. Your new canvas window should now be displaying the variables *low*, *mid*, and *high* beneath the array indexes and they should be moving as their values change.

Click the Stop button  to end the program; then close both canvas windows. Now click the Run in Viewer Canvas button  on the desktop. This opens the Choose Canvas dialog (Figure 12-19) which allows you to choose between the two canvas files associated with the program. Double-clicking on one of the file names or selecting a file name and clicking OK opens the canvas window for the indicated file.

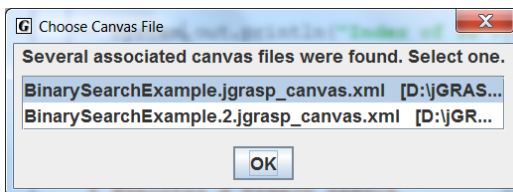





Figure 12-19. Choose Canvas File dialog


To open another canvas window for a program, find the canvas file in the Browse tab and double-click on it. Alternatively, you can click the Open New Canvas button  at the top of the Debug tab pane. Then load the existing file by clicking the Open File button  on the canvas window toolbar (or by clicking **File > Open** on the menu) and selecting the desired canvas file.

For most programs, you'll find that a single canvas file will meet most of your needs. However, multiple canvas windows can be extremely useful when debugging complex programs since the windows all play at the same time.

12.4.7 Experimenting with Different Array Viewers

Since we have *intArray* on the canvas, let's take a look at the different array viewers that are available. The first time you open a viewer on an array, the default will be the Presentation – Structure Identifier viewer. To change viewers, click the Menu button  in the upper right corner of the viewer and hover over **Viewers** to see the list of available viewers. For *intArray*, the list of viewers includes Array Elements, Bar Graph, Basic, Detail, Monitor Info, Presentation, Presentation – Structure Identifier, and Presentation String.

The “Presentation” viewer and the “Presentation – Structure Identifier” viewer provide textbook-like visualizations of arrays and data structures. The primary differences between these two viewers are as follows. The Presentation viewer can handle large data sets efficiently because it expects no errors and processes only the part of the structure that is visible on the screen. Therefore, it is suitable for large arrays as well as the Java library classes in `java.util` package such as `ArrayList`, `LinkedList`, `TreeMap`, and `HashMap`. In contrast, the Presentation – Structure Identifier viewer includes significant error handling capabilities and processes the entire structure. Thus, this viewer is suitable for structures with a smaller number of elements (e.g., < 500) as well as for objects of data structure classes that are under development since they may contain errors.

Now let's change the current viewer for *intArray* by clicking the Menu button  in the upper right corner of the viewer and hovering over **Viewers** to see the list of available viewers. The result of selecting each of the Array Elements, Bar Graph, Basic viewer, and String Presentation viewers for *intArray* are shown in Figures 12-20, 12-21, 12-22, and 12-23 respectively.

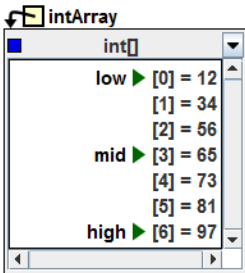


Figure 12-20. Array Elements viewer for *intArray*

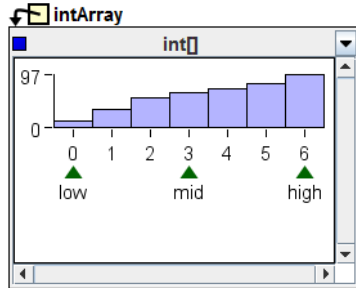


Figure 12-21. Bar Graph viewer for *intArray*

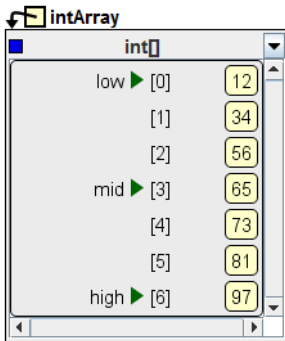


Figure 12-22. Basic viewer for *intArray*

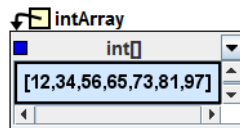


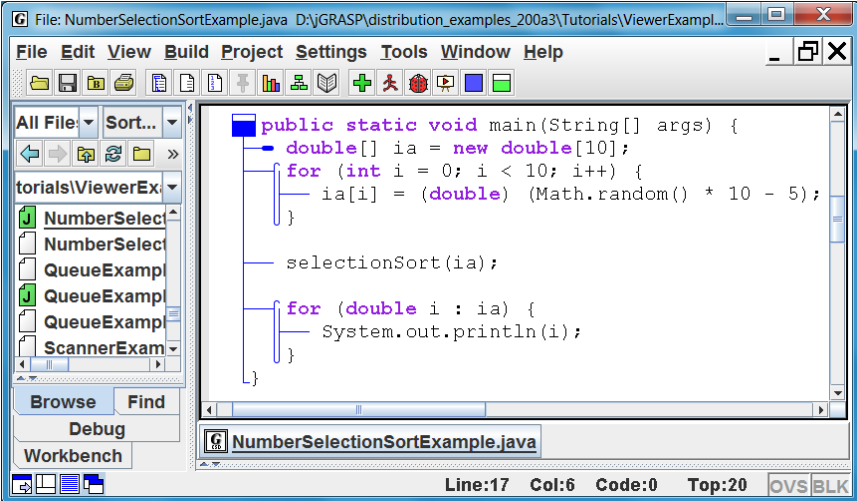
Figure 12-23. Presentation String viewer for *intArray*

12.5 Sorting on the Canvas – NumberSelectionSort

In this last example, we'll use the canvas to visualize a simple selection sort method for an array of *double* values. We'll see how using two separate viewers on the canvas for the same array can enhance the visualization.

12.5.1 Reviewing the Source Code

In the ViewerExamples folder, locate *NumberSelectionSortExample.java* and double-click on it to open it a CSD window. The *NumberSelectionSortExample* class contains two static methods: *main* and *selectionSort*. Figure 12-24 shows the main method which creates *double* array *ia* of size 10, initializes it with 10 random *double* values between -5 and 5, calls *selectionSort* with *ia*, and then prints the values of *ia*.



```

File: NumberSelectionSortExample.java  D:\GRASP\distribution_examples_200a3\tutorials\ViewerExamp...
File Edit View Build Project Settings Tools Window Help
All File Sort...
torials\ViewerEx
NumberSelect
NumberSelect
QueueExamp
QueueExamp
QueueExamp
ScannerExam
Browse Find
Debug
Workbench
NumberSelectionSortExample.java
Line:17 Col:6 Code:0 Top:20 OVSBLK

```

```

public static void main(String[] args) {
    double[] ia = new double[10];
    for (int i = 0; i < 10; i++) {
        ia[i] = (double) (Math.random() * 10 - 5);
    }

    selectionSort(ia);

    for (double i : ia) {
        System.out.println(i);
    }
}

```

Figure 12-24. The *main* method in *NumberSelectionSort*

A selection sort works essentially as follows: find the smallest value in the list and swap it with the first item in the list; find the next smallest item in the list and swap it with the second item in the list; continue until the list is sorted.

The *selectionSort* method (Figure 12-25) has a single parameter *list* of type *double* array which is sorted in ascending order using two nested *for* loops. In the outer loop, *index*, which ranges from 0 to *list.length-1*, is the location in *list* where we want to place the smallest element found each time the inner loop completes. In the inner loop, *scan* ranges from *index+1* to *list.length* and an *if* statement checks for a smaller element on each iteration, and when found, records its location in *min*. After the inner loop completes, the element at *min* will be the smallest element in the range of the inner loop. The element at *min* is swapped with the element at *index*. Then *index* is incremented in the outer loop

and the inner loop finds the next smallest value in *list*. As the program continues, all of the elements to the left to index will have been sorted. The outer loop completes when index reaches the end of its range. At this time, *list* is sorted and the method ends.

```

File: NumberSelectionSortExample.java D:\JGRASP\distribution_examples_200a3\Tutorials\ViewerExamp...
File Edit View Build Project Settings Tools Window Help
[Icons]
static void selectionSort(double[] list) {
    int min;
    double temp;



    for (int index = 0; index < list.length - 1; index++) {
        min = index;
        for (int scan = index + 1; scan < list.length; scan++) {
            if (list[scan] < list[min]) {
                min = scan;
            }
        }

        // Swap the values.
        temp = list[min];
        list[min] = list[index];
        list[index] = temp;
    }
}
NumberSelectionSortExample.java
Line:50 Col:29 Code:0 Top:38 OVSBLK

```

Figure 12-25. The *selectionSort* method in *NumberSelectionSort*

12.5.2 Running the *NumberSelectionSort* in the Canvas

Let's compile the program by clicking the green plus **+**. Now click the *Run in Viewer Canvas* button  and then click the Play button  to start the program visualization.


Notice that the canvas has two viewers for *ia*, the *double* array created and initialized in *main*. The first viewer for *ia* is set to the **Bar Graph** viewer. Viewing the values in a bar graph enhances the visualization by making it easier to see the values as they are swapped during the sort. The second viewer for *ia* is set to the **Presentation - Structure Identifier** viewer which is the default viewer for arrays and structures such as linked lists and trees. Notice that this viewer has been configured to associate the variables *index*, *min*, and *scan* with the array's index values. Recall that this was done for *low*, *mid*, and *high* in the *BinarySearchExample* by clicking the Menu button  in the upper right corner of the viewer, selecting the **Add Index Expressions** option, then entering the expressions in the dialog (see Figure 12-16).

Figure 12-26 shows the canvas for `NumericSelectionSortExample` during the first iteration of the inner loop. When you run the program, the array values will be different from those shown in the figure since the values are randomly generated.

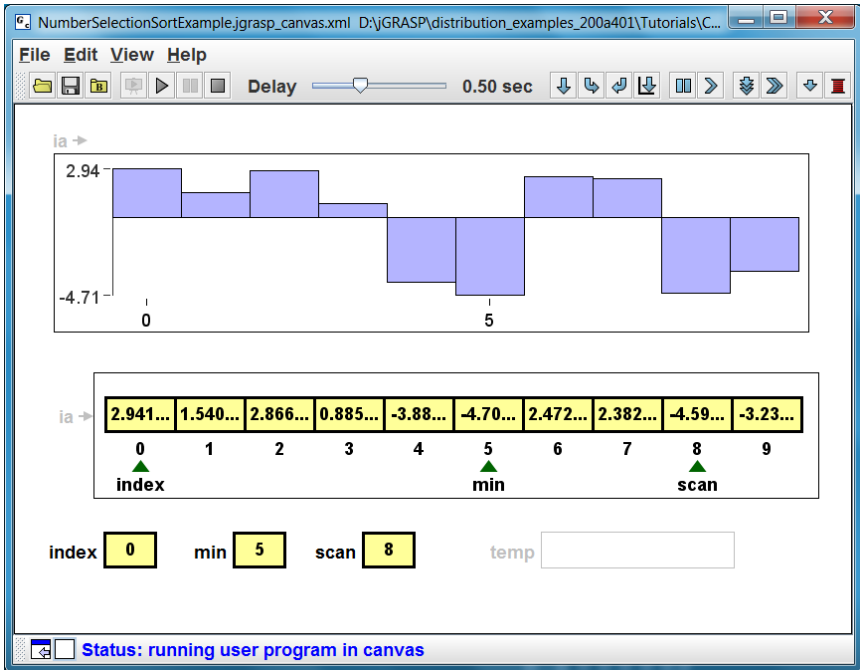





Figure 12-26. The viewer canvas for `NumericSelectionSort` during the sort

As you play or step the canvas, you should see that when *scan* reaches the end of the array, the element at location *min* is the smallest unsorted value. Then you should see this element swapped with the element at *index*. As *index* moves to the right, the elements in the array to the left of *index* should be sorted. When *index* reaches the end of its range, *ia* is sorted and the method ends as shown in Figure 12-27.

Now let's explore several other aspects of running this program in the canvas. Click the *Run in Viewer Canvas* button . Notice that before you click the Play button  or begin stepping in the program, the variable names and viewers are grayed out in the canvas window. After you click the Play button , you should see the variable names and viewers become active as they are created and/or assigned during the stepping. Note that sometimes the viewer is active but the variable name is grayed out. This means the variable is out of scope.

When a variable is no longer in scope, the viewer displays its last known value. In the case of a reference variable for an object (e.g., an array), the value may be updated via another variable that references the object (i.e., an alias). For example, the *double* array *ia* is declared in *main* and then passed to *selectionSort* as the parameter *list*. Since *ia* and *list* refer to the same object, the viewer open on *ia* is updated anytime *list* is updated. Hence, the viewer for the array is updated and active even though its variable *ia* is grayed out as shown above in Figure 12-26.

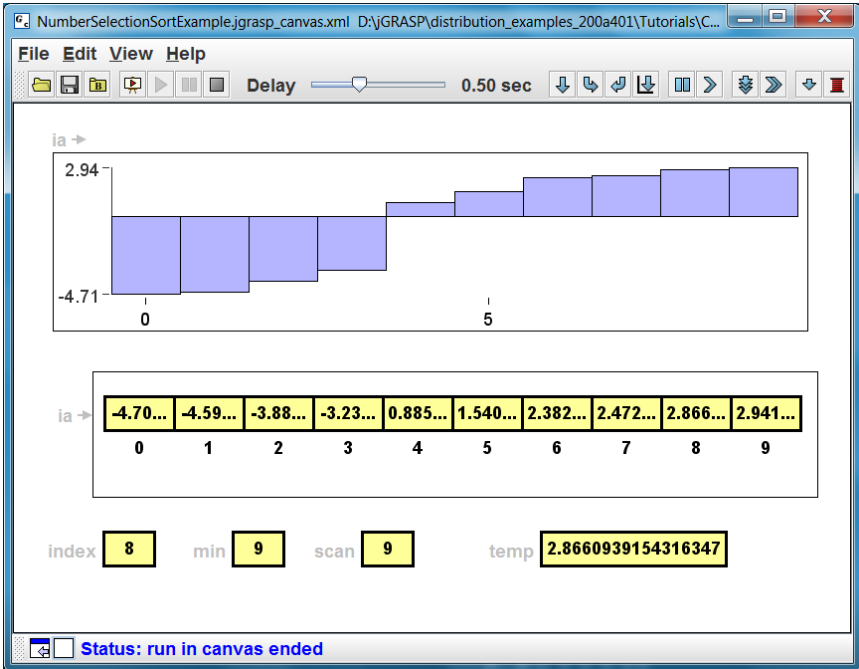














Figure 12-27. The viewer canvas for NumberSelectionSort after the sort

If a viewer never comes into scope directly or via an alias, it will remain grayed out even though the program completes normally. For example, if you click the *Run in Viewer Canvas* button  and then click the *Step* button  repeatedly without stepping-in to the *selectionSort* method, you will see that the viewers for *index*, *min*, *scan*, and *temp* remain grayed out since they never came into scope, even though the array *ia* has been sorted.

12.6 Wrapping Up – Notes on Using the Canvas

Running in the Canvas – After a program has been compiled, run it in the canvas as follows:


- (1) On the desktop toolbar, click the Run in Canvas button . This launches the program in the debugger and either opens a new canvas window or one with a previously saved canvas file for the program.
- (2) For a new canvas, Click the Step  and/or Step-in  on the canvas window or debug tab until you see variables of interest in the Variables tab. You may also set one or more breakpoints  and click the Resume button  to run to a breakpoint and stop. Drag the variables of interest onto the canvas to open the respective viewers, and then save the canvas.
- (3) Control the canvas with the Play button , Pause button  and Stop button  as well as the debug buttons. To regulate the speed of the program, decrease or increase the delay between steps using the Delay slider  0.30 sec.
- (4) Viewers on the canvas are updated automatically by stepping at the statements that modify the variables, fields, and expressions represented in viewers. Most viewers are state-based so updates only occur when the debugger has just completed a step or when it is stopped at a breakpoint. If you set a breakpoint and resume to it, the debugger runs to the breakpoint and stops. Some viewers for variables on the path to the breakpoint may not be updated, especially if the variables are not in scope (grayed-out) at the breakpoint.
- (5) Observe the behavior of the program, and re-run the program in the canvas as appropriate to understand and/or debug the program.

Viewer Menu – With an active viewer selected on the canvas, click the Viewer Menu  button in the upper right corner of the viewer frame. All settings and preferences below are saved with the canvas file.

- (1) **Viewers** –selects among the available viewers for an active object.
- (2) **Edit Expression/Scope** – allows you change the expression for the viewer and if applicable, its class, method, and call depth as applicable.
- (3) **Viewer Settings** – opens the viewer settings dialog.
- (4) **Invoke Method** – opens the Invoke Method dialog for an object viewer.
- (5) **Viewer Info** – displays a short description of the selected viewer.
- (6) **Transparent** – toggles viewer borders off/on and makes the viewer background transparent for some viewers.
- (7) **Scope Test** – sets the scope to **full**, **ignore depth** (recursive), or **none**.

- (8) **Move To Back** – moves viewer to back or behind an overlapping viewer.
- (9) **Auto Size** – allows the viewer to resize automatically within limits. If you manually resize a viewer, Auto Size is disabled until you reselect it.
- (10) **Remove** – removes a viewer from the canvas.

Canvas File (.jgrasp_canvas.xml) – Naming Convention and Contents

- (1) **Naming convention** – To associate the canvas file with a program, the name must begin with the Java file name that contains main or with the jGRASP project file name for the program. For example, program MyProgram.java has canvas *MyProgram.jgrasp_canvas.xml*. If MyProgram.java was in a jGRASP project MyProject.gpj, the canvas file could be *MyProject.jgrasp_canvas.xml*. This naming convention enables “Run in Canvas”  to open the appropriate canvas file when launching the program.
- (2) **XML File Contents** – Each viewer on the canvas is saved in the *.jgrasp_canvas.xml* file along with its settings and its scope and call stack depth. This has important implications regarding program changes after variables have been dragged on the canvas. For example, changing the names of classes and methods may make the recorded scope information incorrect and thus make the viewer unresponsive. See canvas **Edit** menu below to make “global changes to the canvas. Otherwise, if you have only a few variables you can remove the viewer and then drag the variable onto the canvas again. The recorded scope information also prevents you from simply renaming a canvas file after renaming your program. If you rename your program, see the “Change Class and Method Name option in the **Edit** menu below.






Canvas Edit Menu











- (1) **Add Text Box** – allows you to add a text box to the canvas (e.g., as a title or description of the canvas). The entered text may be plain text or HTML.
- (2) **Add Expression Viewer** – allows you to add a viewer and an expression. Note that the expression will be evaluated at each step whenever any involved variables/methods are in scope.
- (3) **Change Class or Method Names** – allows you to make “global” changes to the canvas for classes, method, and call depth. Set call depth to zero to allow recursive scope within the method.
- (4) **Exclusions** – allows you to indicate classes and/or methods to exclude from “Play” and “Step In” when Auto Step is on.








For addition details click **Help > Viewers and Canvas** on the canvas menu.

12.7 Looking Ahead to Data Structures – Exercises



The files for these exercises come with the jGRASP distribution and are found in the **Examples\Tutorials\ViewerExamples**.

- (1) Use the viewer canvas to experiment with the Java Collection classes.
 - a. Open *CollectionsExample.java* by double-clicking on the file in the Browse tab.
 - b. On the desktop toolbar click the Run in canvas  button. This opens a new (empty) canvas window if no canvas file has been previously created for the program.
 - c. Click the *Step*  button on the canvas window or debug tab until you see variables of interest in the Variables tab.
 - d. Drag one or more variables onto the canvas; a default viewer should open for each variable. For example, drag the following onto the canvas.
 - i. `stringList` for an array of Strings
 - ii. `myArrayList`
 - e. Save the canvas.
 - f. On the canvas, click the Play  button (auto step-in) to start the visualization. Use the Pause  button and Stop  button as needed. You can regulate the speed with the "Delay" slider.

Delay  0.30 sec
 - g. Since the *CollectionsExample* has an infinite loop, you will need to click the Stop  button on the canvas or the End button on the Run I/O tab to end the program.
 - h. On the toolbar click the Run in canvas  button. The canvas you created above should open, and the program should be stopped at the first executable statement. You are now ready to Play , Step , Step-in , set breakpoints , Resume , etc.
 - i. Now let's open a second canvas:
 - i. Set a breakpoint  in the program, and run the debugger .

- ii. After the program stops the breakpoint, open a canvas by clicking the Open Canvas  button on the Debug toolbar.
- iii. Click the *Step*  button on the canvas window or debug tab until you see variables of interest in the Variables tab.
- iv. Drag one or more variables onto the canvas; a default viewer should open for each variable. For example, drag one of the following onto the canvas.
 1. myLinkedList
 2. myTreeMap
 3. myHashMap
- v. Play , Step , Step-in , set breakpoints , Resume , etc.









In the exercises below, a canvas file has already been created for each of the example programs indicated below. See the notes on the last page for details on using the Viewer menu.

- (2) BinaryTreeExample – In the Browse tab, double-click on the file to open it. On the desktop toolbar click the Run in canvas  button. On the canvas, click the Play  button.

For fun let's try some **Interactions** with the running canvas.

- a. Pause the Viewer Canvas, then click on *BinaryTreeExample.main* in the call stack pane of the Debug tab. This is to ensure that *bt* is in scope.
- b. Click on the Interaction tab (lower part of desktop). As the statements below are entered in the Interactions tab, you should see the elements (in the viewer on the canvas) added to *bt* and then removed as the methods are invoked.

```
bt.add(455);  
bt.add(350);  
bt.remove(455);
```

- (3) QueueExample – In the Browse tab, double-click on the file to open it. On the desktop toolbar click the Run in canvas  button. On the canvas, click the Play  button.
- (4) LinkedListExample – In the Browse tab, double-click on the file to open it. On the desktop toolbar click the Run in canvas  button. On the canvas, click the Play  button.
- (5) DoublyLinkedListExample – In the Browse tab, double-click on the file to open it. On the desktop toolbar click the Run in canvas  button. On the canvas, click the Play  button.
- (6) ScannerExample – In the Browse tab, double-click on the file to open it. On the desktop toolbar click the Run in canvas  button. On the canvas, click the Play  button.