

2 Getting Started with jGRASP 2.0

After you have successfully installed the Java JDK and jGRASP, you are ready to get started. For the examples in this section, Microsoft Windows and Java will be used. However, much of the information in sections 2.1 - 2.7, 2.11, and 2.13 - 2.16 applies to other operating systems and supported languages for which you have installed a compiler (e.g., Ada, C, C++, and Python) unless noted otherwise. For example, in the “Creating a New File” below, you may select C++ as the language instead of Java, and then enter a C++ example. Also note that for Ada, C, and C++, you’ll need to “compile and link” (rather than just “compile”) in order to run your program.

Objectives – When you have completed this tutorial, you should be comfortable with editing, compiling, and running Java programs in jGRASP. In addition, you should be familiar with the pedagogical features provided by jGRASP, including using interactions, generating the CSD, folding your source code, numbering the lines, stepping through the program in the integrated debugger, and using the dynamic viewers and canvas.

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

- 2.1 Starting jGRASP
- 2.2 Quick Start - Opening a Program, Compiling, and Running
- 2.3 Creating a New File
- 2.4 Saving a File
- 2.5 Building Java Programs - - Recap
- 2.6 Generating a Control Structure Diagram
- 2.7 Using Line Numbers
- 2.8 Using the Debugger (Java only)
- 2.9 Using Interactions (Java only)
- 2.10 Using Viewers and the Viewer Canvas (Java only)
- 2.11 Creating and Using Projects
- 2.12 Generating and Using a UML Class Diagram (Java only)
- 2.13 Opening a File – Additional Options
- 2.14 Closing a File
- 2.15 Exiting jGRASP
- 2.16 Review and Preview of What’s Ahead
- 2.17 Exercises

2.1 Starting jGRASP



jGRASP

If you are working in a Microsoft Windows environment, you can start jGRASP by double clicking its icon on your Windows desktop.

If you don't see the jGRASP icon on the desktop, try the following: click *Start > All Programs > jGRASP (folder) > jGRASP*.

Depending on the speed of your computer, jGRASP may take between 10 and 30 seconds to start up. The jGRASP virtual **Desktop**, shown below, is composed of a Control Panel with a menu and toolbar across the top and three resizable panes. The *left pane* has tabs for **Browse**, **Debug**, **Find**, and **Workbench**. The Browse tab, which is the default when jGRASP is started, lists the files in the current directory. The large *right pane* is for CSD, Canvas, and UML Windows. The *lower pane* has tabs for **jGRASP Messages**, **Compile Messages**, **Run I/O**, and **Interactions**. The panes can be resized by selecting the partition with the mouse (left-click and hold down) then dragging the partition. You can also click the arrowheads on the partition to open and close the pane.

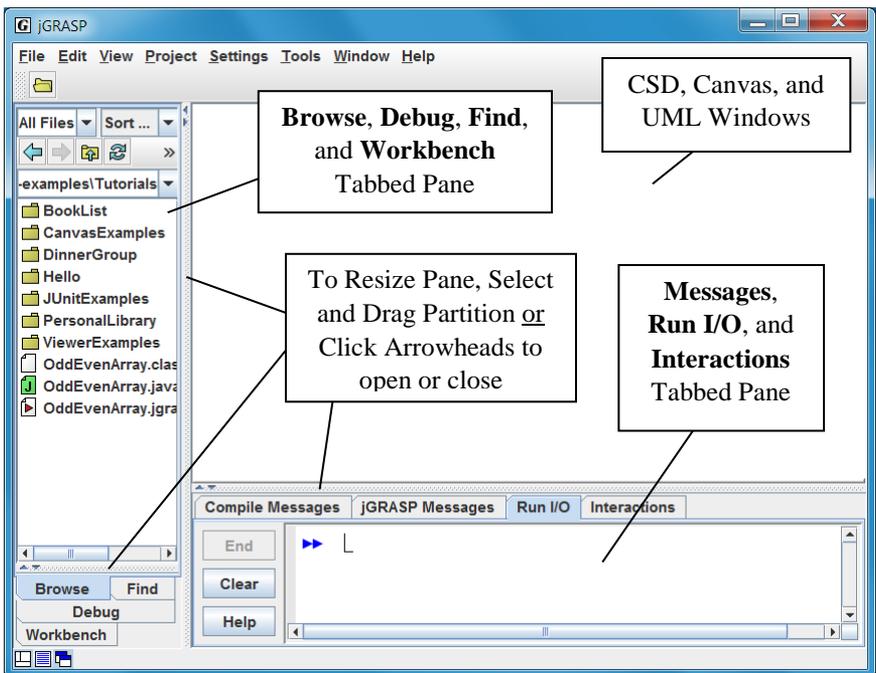


Figure 2-1. The jGRASP Virtual Desktop

2.2 Quick Start - Opening a Program, Compiling, and Running

Example programs are available in the jGRASP folder in the directory where it was installed (e.g., c:\Program Files\jgrasp\examples\Tutorials). Since the installation directory is read-only, you should copy the tutorial folder to one of your personal folders (e.g., in your *My Documents* folder) so that you will be able to edit, compile, etc. To copy the files, find the top menu, click **Tools > Copy Example Files**, then navigate to the folder where you want to save the jgrasp_examples folder.

Note: If you already have example programs with which you are familiar, you may prefer to use them rather than the ones included with jGRASP as you work through this first tutorial.

Clicking the Open File button  on the toolbar pops up the Open File dialog. However, the easiest way to open existing files is to use the **Browse** tab (below). The files shown initially in the Browse tab will most likely be in your home directory. You can navigate to the appropriate directory by double-clicking on a folder in the list of files or by clicking on  as indicated in the figure below. The refresh button  updates the Browse tab. Below, the Browse tab is displaying the contents of the Tutorials folder.

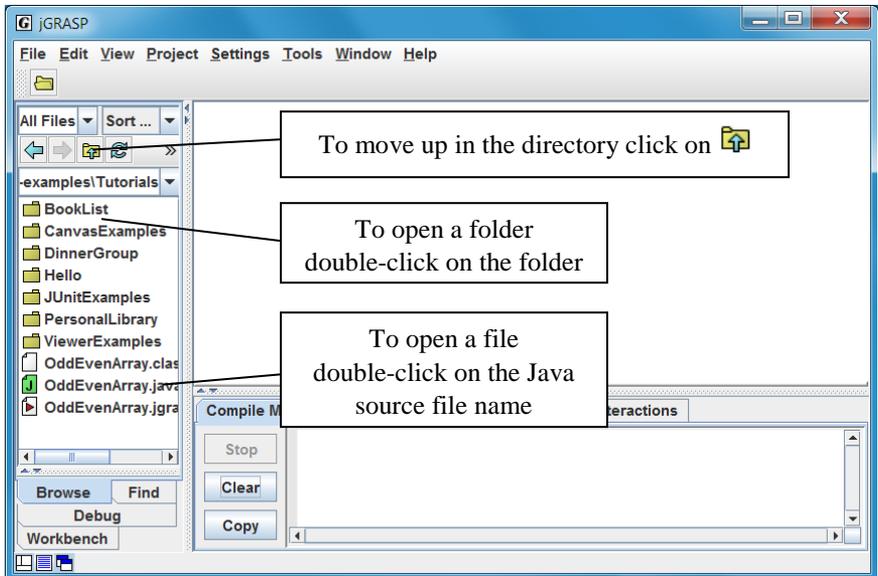


Figure 2-2. The jGRASP Virtual Desktop

Double-clicking on the Hello folder, then the Hello.java file, as shown in **Step 1** below, opens the program in a CSD window. The CSD window is a full-featured editor for entering and updating your programs. Notice that opening the CSD window places additional buttons on the toolbar. Once you have opened a program or entered a new program (**File > New File > Java**) and saved it, you are ready to compile the program and run it. To compile the program, click on the **Build** menu then select **Compile**. Alternatively, you can click on the Compile button **+** (or press **Ctrl-B**) indicated by **Step 2** below. After a successful compilation – no error messages in the Compile Messages tab (the lower pane), you are ready to run the program by clicking on the Run button **⚡** (or press **Ctrl-R**) as shown in **Step 3** below, or you can click the **Build** menu and select **Run**. The standard input and output for your program will be in the Run I/O tab pane. Short cuts for Compile and Run are Ctrl-B and Ctrl-R. If **Auto Compile** is ON (default), **Run** will also **Compile** the file if required.

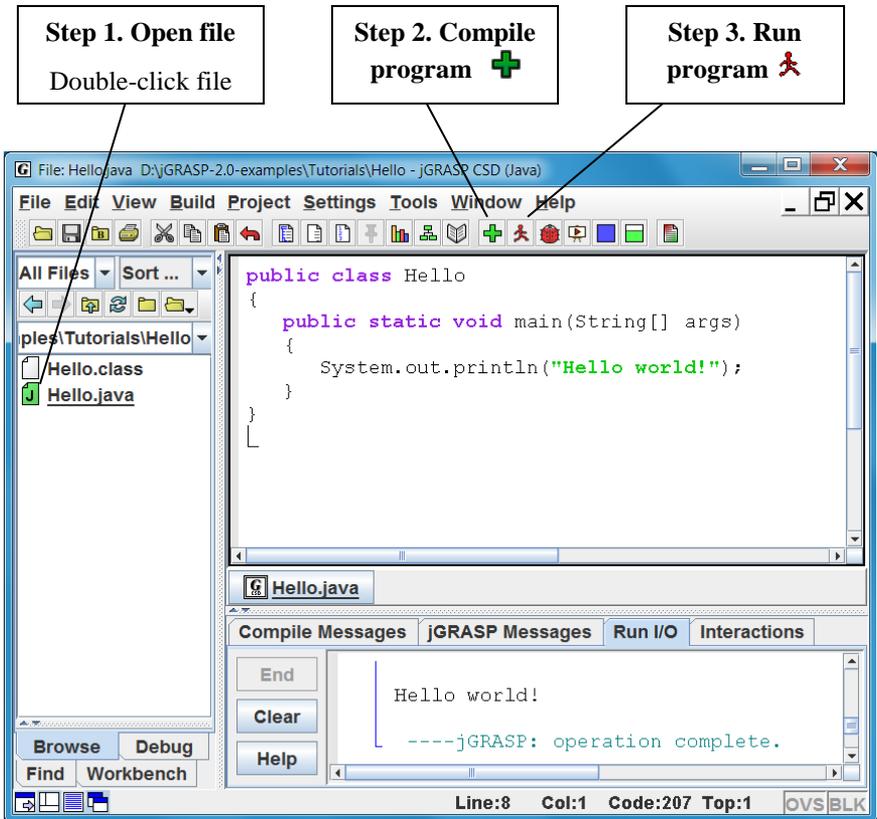


Figure 2-2. After loading file into CSD Window

2.3 Creating a New File

On the main menu, click **File > New File > Java** to create a new Java file. Note that the list of languages displayed by **File > New File** will vary with your use of jGRASP. If the language you want is not listed, click **Other** to see the additional available languages. The languages for the last 25 files opened will be displayed in the initial list; the remaining available languages will be under **Other**.

After you click on **File > New File > Java**, a CSD window is opened in the right pane of the Desktop as shown in Figure 2-4 below. Notice the title for the frame, jGRASP CSD (Java), which indicates that the CSD window is Java specific. If Java is not the language you intend to use, you should close the window and then open a CSD window for the correct language. Notice that a *button* for each open file appears below the CSD windows in an area called the windowbar (similar to a taskbar in the Windows OS environment). Later when you have multiple files open, the windowbar will be quite useful for popping a particular window to the top. The buttons can be reordered by dragging them around on the windowbar. Figure 2-4 shows the newly opened CSD window maximized in the desktop.

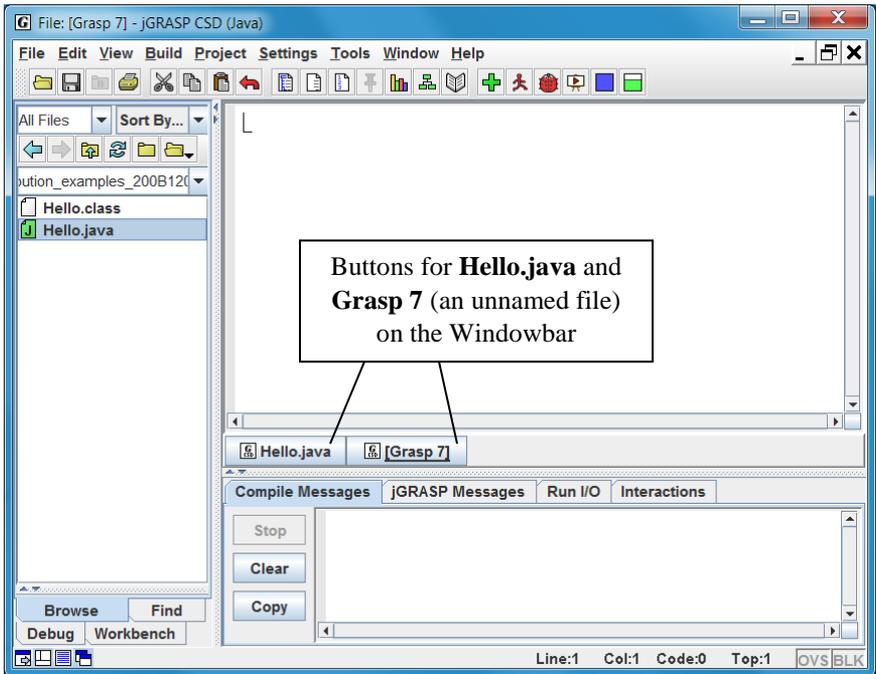


Figure 2-4. After opening a new CSD Window for Java

 When a CSD window is open, there are three buttons to the right of the menu that control its display. The first button minimizes the CSD window; the second button restores (unmaximizes) the CSD window. When the CSD window is not maximized you'll see   , and the second button will maximize the CSD window. The third button closes the CSD window. You may also make the Desktop itself full screen by clicking the appropriate button in the upper corner of it.

Figure 2-5 shows the CSD window unmaximized within the virtual Desktop. The flashing “L” shaped cursor in the upper left corner of the empty window indicates where text will be entered.

TIP: You may choose whether or not you want all of your CSD windows to be maximized automatically when you open them. This is the default for new installations of jGRASP. Click **Settings > Desktop**, and then click **Open Desktop Windows Maximized** to toggle this on/off (a check mark indicates that this option is turned ON). You can also click  (or ) in the lower left corner of the desktop to change this setting. jGRASP will remember this setting even when you update to a new version.

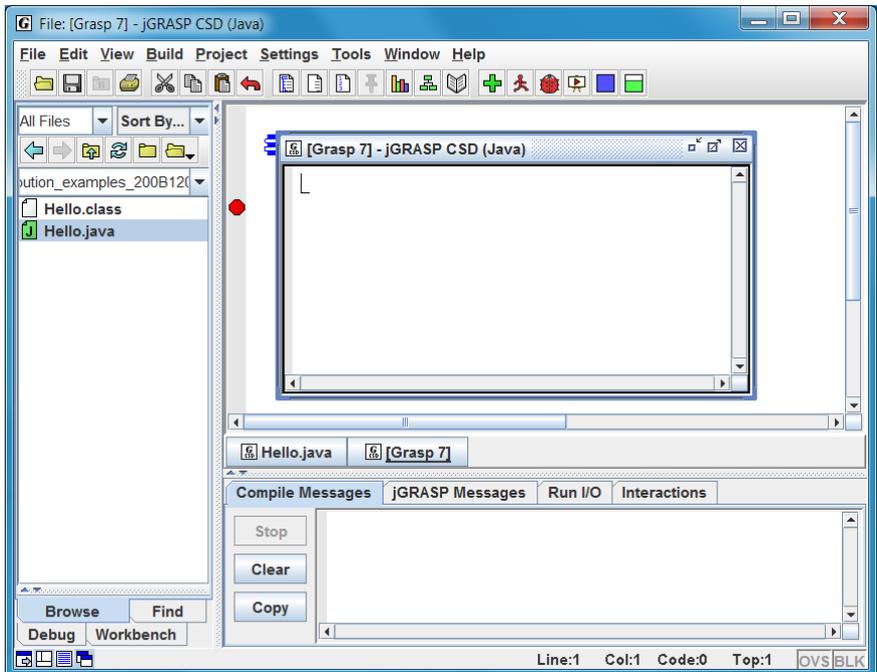


Figure 2-5. CSD Window maximized in Desktop

Type the following Java program in the CSD window, exactly as it appears. Remember, Java is case sensitive. Alternatively, you may copy/paste the Hello program into this window, then change the class name to Hello2 and add the “Welcome...” line.

```
public class Hello2
{
    public static void main(String[] args)
    {
        System.out.println ("Hello world!");
        System.out.println ("Welcome to jGRASP!");
    }
}
```

After you have entered the program, your CSD window should look similar to the program shown in Figure 2-6.

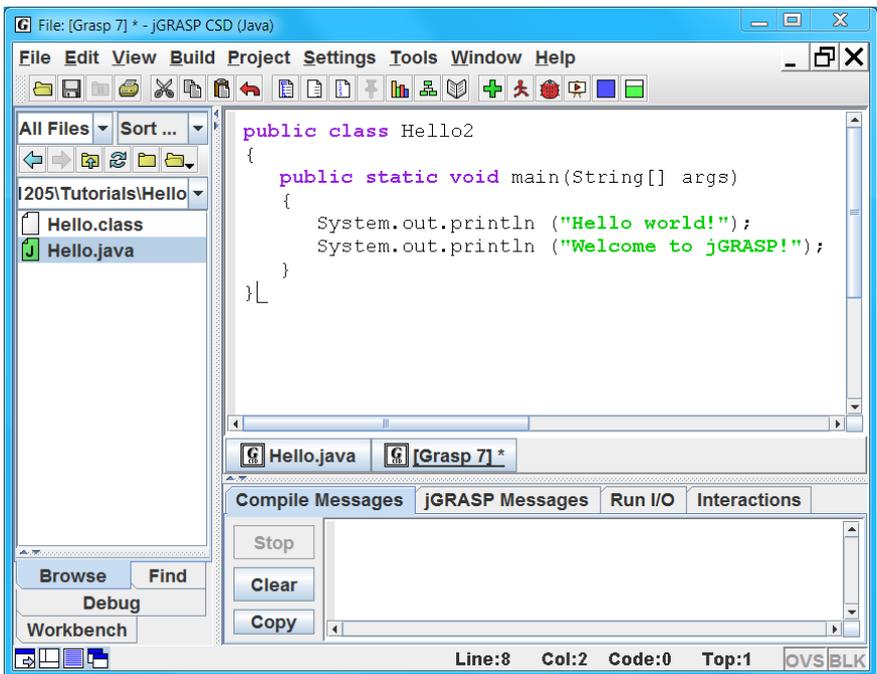


Figure 2-6. CSD Window with program entered

2.4 Saving a File

You can save the program as "Hello2.java" by doing any of the following:

- (1) Click the Save button  on the toolbar, or
- (2) Click **File > Save** on menu, or
- (3) Press **Ctrl-S** (i.e., while pressing the Ctrl key, press the "s" key).
- (4) With **Auto Compile** ON (the default), attempting to Compile or Run an unsaved file will also initiate a Save operation followed by the Compile and/or Run.

If the file has not been saved previously, the Save dialog (see Figure 2-7) pops up with the name of the file set to the name of the class file. Note, in Java, the file name must match the class name (i.e., class Hello2 must be saved as Hello2.java). Be sure you are in the correct directory. If you need to create a new directory, click the folder button  on the top row of the Save dialog. When you are in the proper directory and have the correct file name indicated, click the *Save* button on the dialog. After your program has been saved, it should be listed in the Browse tab (see Figure 2.10). If you do not see the program in the Browse tab, you may need to navigate to the directory where the file was saved or click  on the toolbar to change the **Browse** tab to the directory of the current file. Now you are ready to compile and run Hello2.

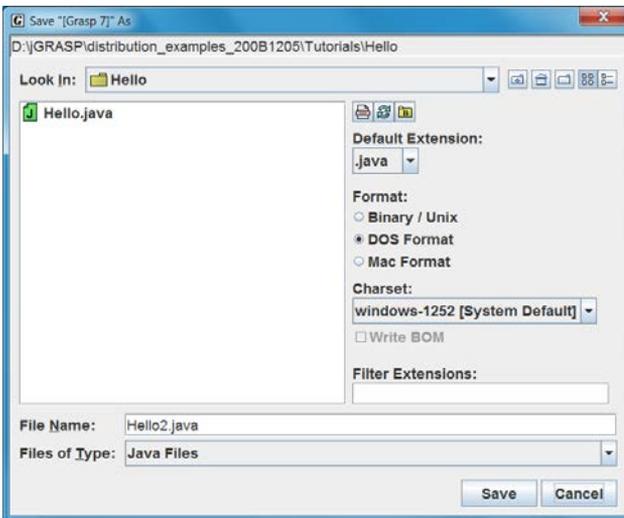


Figure 2-7. Save As dialog for previously unsaved file

2.5 Building Java Programs - - Recap

As seen in the previous sections, Java programs are written in an edit window, saved, compiled, and run. A somewhat more detailed description of steps for building software in Java is as follows.

- (1)  **Enter** and **Save** your source code in a CSD window. Be sure the file name matches the Java class name and has the “.java” extension (e.g., class MyProgram should be saved as MyProgram.java). You should try to enter your program in chunks so that it will always compile without errors.
- (2)  **Compile** the source program (e.g., MyProgram.java) to create the byte code file with a “.class” extension (e.g., MyProgram.class). After attempting to compile your program, you may need to make corrections via the edit window (step 1) based on the error messages provided by the compiler and then compile the program again. Note that the .class file is not created until your program compiles with no error messages. Keyboard shortcut: **Ctrl-B**
- (3)  **Run** your program (main method or applet). In this step, the byte code or .class file produced by the compiler is executed by the Java Virtual Machine. After you run your program, you should inspect the output (if any) to make sure the program did what you intended. At this point, you may need to find and correct logical errors (bugs). After making the corrections in the edit window (step 1), you will need to compile your program again (step 2). Later, we will use the debugger to step through a program so we can see what happens after each individual statement is executed. Keyboard shortcut: **Ctrl-R**

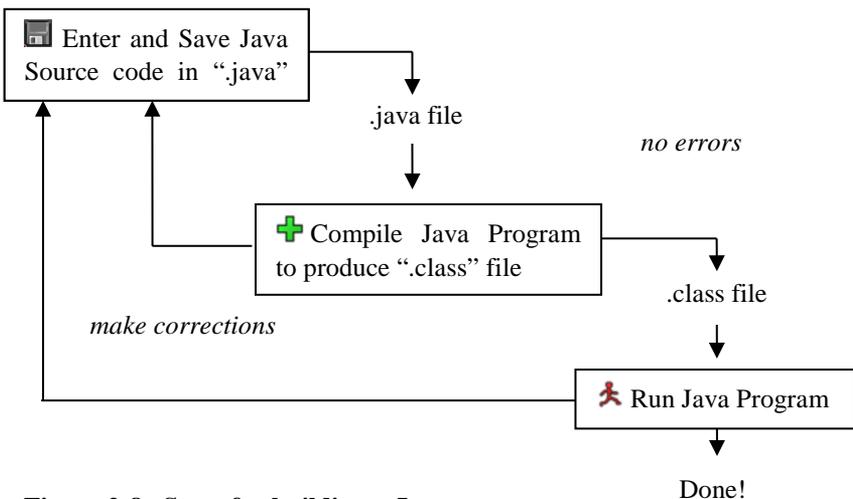


Figure 2-8. Steps for building a Java program

2.6 Generating a Control Structure Diagram

You can generate a Control Structure Diagram in the CSD window whenever you have a syntactically correct program, such as the Hello2.java program described above. Note that CSD generation checks only the structure of a program, so even though the CSD may generate successfully, the program may not compile. Generate the CSD for the program by doing any of the following:

- (1) Click the Generate CSD button , or
- (2) Click **View > Generate CSD** on the menu, or
- (3) Press the **F2** key.

If your program is syntactically correct, the CSD will be generated as shown for the Hello2.java program in Figure 2-10. After you are able to successfully generate a CSD, go on to the next section below.

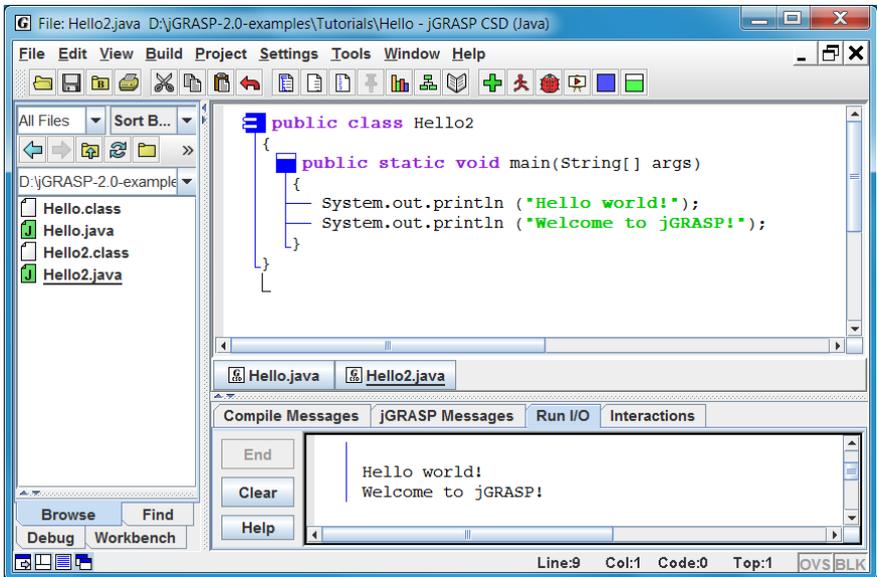


Figure 2-10. After CSD is generated

If a syntax error is detected during the CSD generation, jGRASP will highlight the vicinity of the error and describe it in the message window.

If you do not find an error in the highlighted line, be sure to look for the error in the line just above it. For example in Figure 2-11, the semi-colon was omitted at the end of the first println statement. As you gain experience, these errors will become easier to spot.

If you are unable find and correct the error, you should try compiling **+** the program since the compiler may provide a more detailed error message (e.g., `' ; ' expected`) as shown in Figure 2-11.

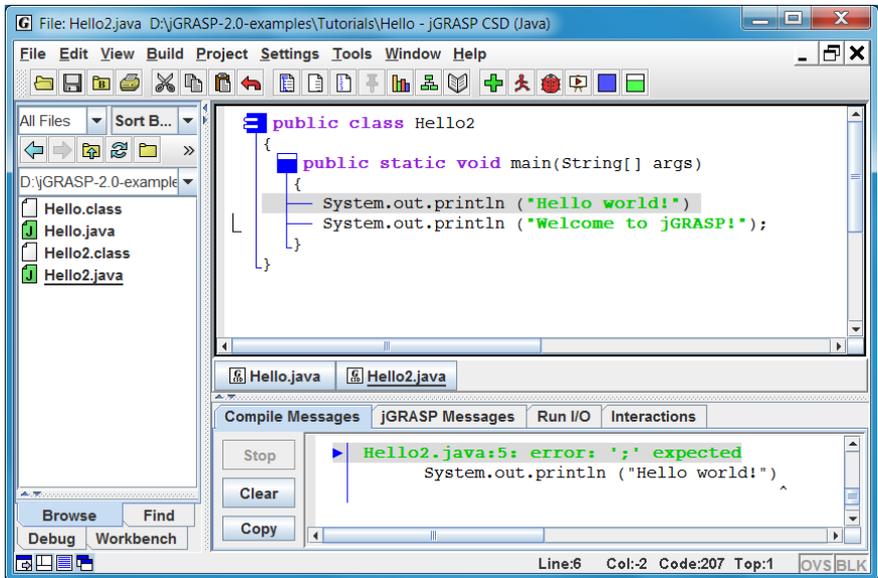


Figure 2-11. Syntax error detected

Removing the CSD leaves the code 3-space indented and removes the leading spaces that were occupied by the CSD. You can remove the CSD by doing any of the following:

- (1) Click the Remove CSD button , or
- (2) Click **View > Remove CSD** on the menu, or
- (3) Press **Shift-F2**.

Note that it is not necessary to ever remove the CSD since the CSD is never saved with your programs. Many users turn on Auto Generate (**View >** then check **ON Auto Generate CSD**) so that the CSD will be generated when a file is opened and again anytime the file is compiled.

Remember, the purpose of using the CSD is to improve the readability of your program. While this may not be obvious on a simple program like the example

above, it should become apparent as the size and complexity of your programs increase, especially when they contain nested control structures.

TIP: As you enter a program, try to enter it in “chunks” that are syntactically correct. For example, the following is sufficient to generate the CSD.

```
public class Hello
{
}
```

As soon as you think you have entered a syntactically correct chunk, you should generate the CSD. Not only does this update the diagram, it catches your syntax errors early. Alternatively, if you have Auto Generate (**View** > then check ON **Auto Generate CSD**) turned on, CSD will generated/updated each time you successfully compile your program.

Folding a CSD is a feature that becomes increasingly useful as programs get larger. After you have generated the CSD, you can fold your program based on its structure. For example, if you double-click on the class symbol  in the program, the entire program is folded (Figure 2-12). Double-clicking on the class symbol again will unfold the program completely.

If you double-click on the “plus” symbol , the first layer of the program will be unfolded. Large programs can be unfolded layer by layer as needed. Although the example program has no loops or conditional statements, these may be folded by double-clicking the corresponding CSD control constructs. For other folding options, see the **View** > **Fold** menu.

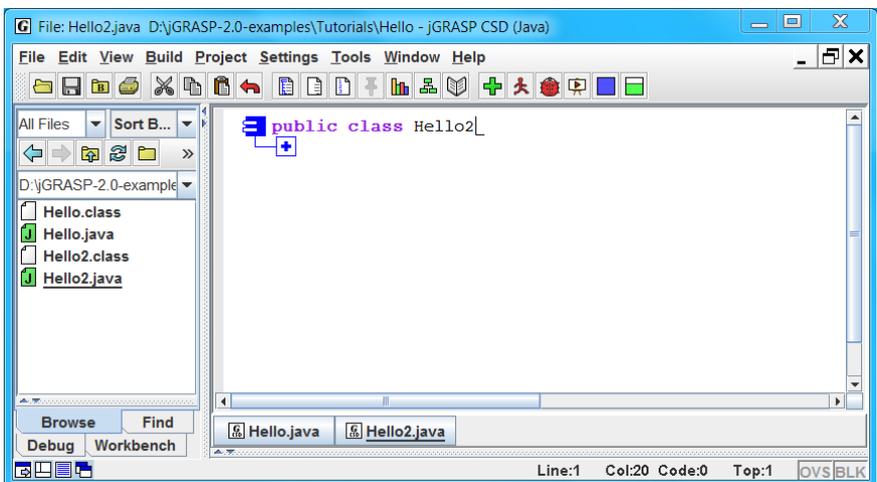


Figure 2-12. Folded CSD

2.7 Using Line Numbers

Line numbers can be very useful when referring to specific lines or regions of a program. Although not part of the actual program, they are displayed to the left of the source code as indicated in Figure 2-13.



Line numbers can be turned on and off by clicking the line numbers toggle button on the CSD window toolbar or via the View menu where you can check or uncheck the check box for Line Numbers. In addition, **Ctrl-L** toggles line numbers on and off.

With Line numbers turned on, if you insert a line in the code, all line numbers below the new line are incremented.



You may “freeze” and “unfreeze” the line numbers by clicking on the Freeze Line Numbers button. The default is for line numbers to increment when the ENTER key is pressed. When you freeze line numbers, they are not incremented when ENTER is pressed. This is useful if you are looking at several messages that reference line numbers in your program. As you make changes, you may not want the line numbers to change until you have made all of the changes. The line numbers will not be updated until you unfreeze the line numbers by clicking the button  again. This feature is also available on the View menu.

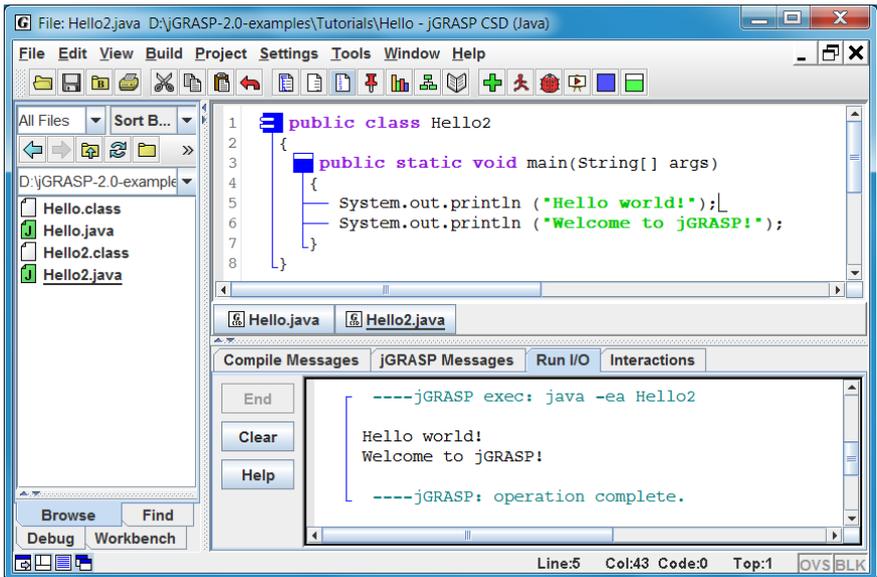


Figure 2-13. Line numbers in the CSD Window

2.8 Using the Debugger (Java only)

jGRASP provides an easy-to-use visual Debugger for Java that allows you to set one or more breakpoints in your program, run the debugger, then after the program reaches a breakpoint, step through your program statement by statement. First, let's check the **Build** menu to make sure that **Debug Mode** is checked ON. This should always be checked ON so your .class file will contain information for the debugger). Now, to set a breakpoint, move the mouse over the gray column to the left of the line where you want to set the breakpoint. When you see the breakpoint symbol, left-clicking the mouse will set the breakpoint. You can also set a breakpoint by left-clicking on the statement where you want your program to stop, then right-clicking to select **Toggle Breakpoint** (Figure 2-18). Alternatively, after left-clicking on the line where you want the breakpoint, click **View > Breakpoints > Toggle Breakpoint**. You should see the red octagonal breakpoint symbol  appear in the gray area to the left of the line. The statement you select must be an executable statement (i.e., one that causes the program to do something). In the Hello2 program below, a breakpoint has been set on the first of the two *System.out.println* statements, which are the only statements in this program that allow a breakpoint.

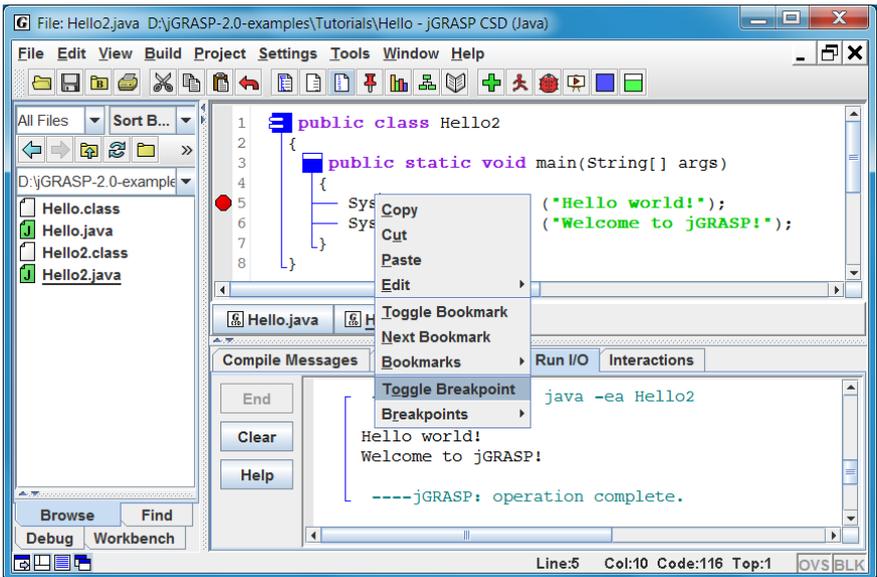


Figure 2-18. Setting a breakpoint

To start the debugger on an application, click the debug button  on the toolbar (or press **Ctrl-D**). Alternatively, you can click **Build > Debug**. When the debugger starts, the Debug tab with control buttons (Figure 2-19) should pop up in place of the Browse tab, and your program should stop at the breakpoint as shown in Figure 2-20 below.



Figure 2-19. Debugger control buttons

Only the “step” button  of the debugger control buttons, located at the top of the Debug tab, is needed in this section. Each time you click the “step” button , your program should advance to the next statement. After stepping all the way through your program, the Debug tab will go blank to signal that the debug session has ended. When a program contains variables, you will be able to view the values of the variables in the Debug tab as you step through the program.

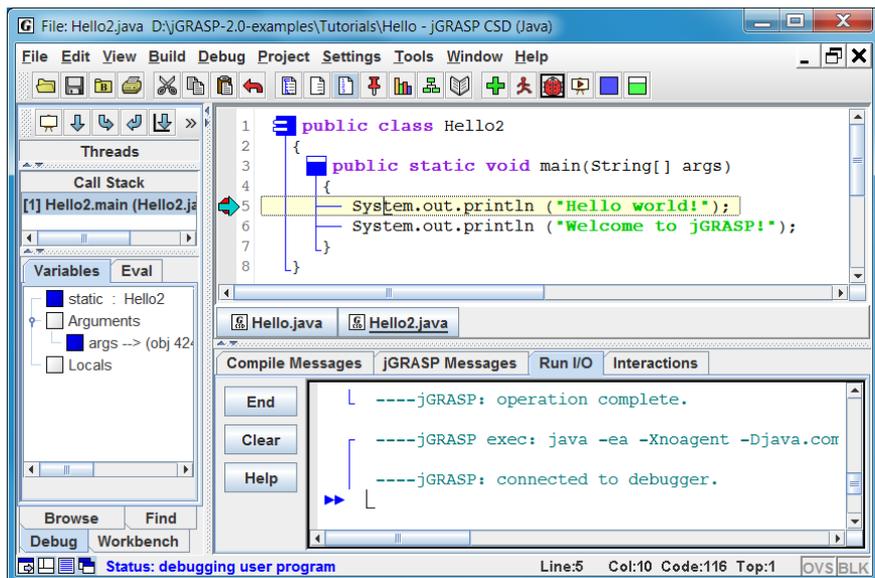


Figure 2-20. Stepping with the Debugger

In the example above, the program has stopped at the first output statement. When the step button  is clicked, this statement will be executed and “Hello world!” will be output to the Run I/O tab. Clicking the step button  again will output “Welcome to jGRASP!” on the next line. The third click on the step button will end the program, and the Debug tab should go blank as indicated above. When working with the debugger, remember that the highlighted statement with the blue arrow pointing to it will be the next statement to be executed. See the Debug menu for a list of keyboard shortcuts. For a complete description of the other debugger control buttons, see the tutorial on the *Integrated Debugger*.

2.9 Using Interactions (Java only)

While all of your Java programs will be built using the steps described above, or some variation of them, jGRASP provides an Interactions feature which can be very useful along the way. The **Interactions** tab, located next to the **Run I/O** tab in the lower window of the desktop, allows you to enter most Java statements and expressions and then execute or evaluate them immediately when you press ENTER. Interactions can be especially helpful when learning and experimenting with new elements in the Java language. You can also create individual objects and invoke their methods as you would in a program.

Consider the following statements that declare and assign the variables `i`, `j`, and `sum` of type `int`.

```
int i = 10;
int j = 25;
int sum = i + j;
```

Type these statements into the Interactions tab and see the results for each statement as soon as you press ENTER. As each variable is declared and assigned, it appears in the Workbench tab on the left. If you enter an expression rather than a statement, the expression will be evaluated and the results will appear directly below the expression. Try entering: `i * j` followed by ENTER without the semi-colon. Figure 2-8 shows the Workbench and Interactions tabs after the three statements and one expression have been entered. If you later assign a new value to an existing variable (e.g., `j = 99;`), the variable in the Workbench tab will be updated immediately.

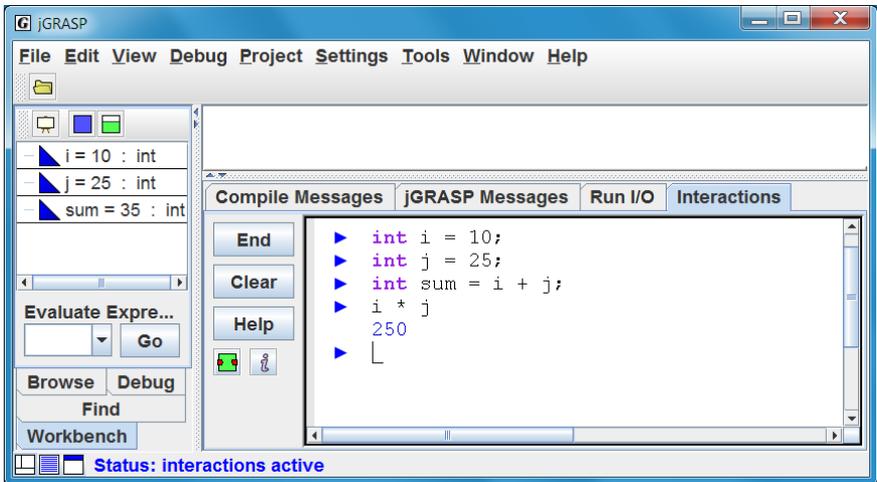


Figure 2-8. Using Interactions

Consider the following expression for a String that includes escape sequences for newline (`\n`) and tab (`\t`).

```
"Hi \n\tfrom \n\t\tInteractions!"
```

Type this expression into the Interactions tab and press ENTER to see the results as shown below in Figure 2-9.

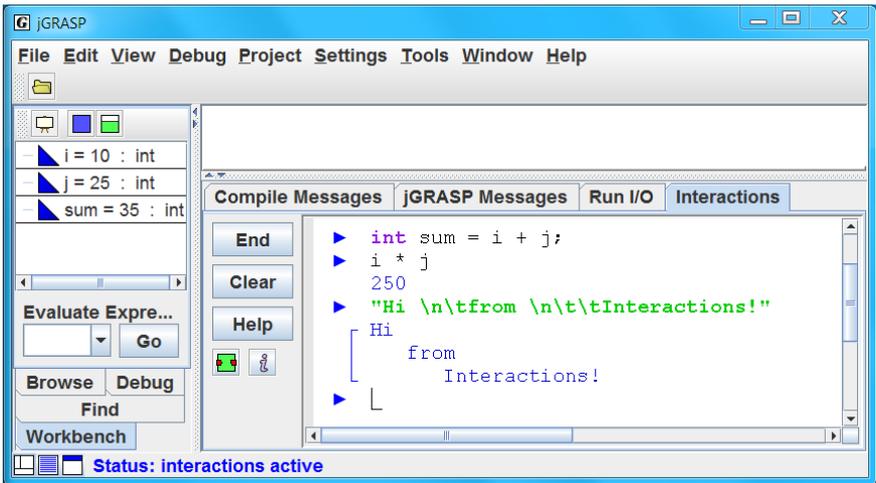


Figure 2-9. Using Interactions – a String expression

To find a statement you have already entered, press the UP and DOWN arrow keys to scroll through the previous statements (history) one by one until you reach the statement. Then use the LEFT and RIGHT arrow keys or mouse to move around within the statement in order to make the desired changes. Press ENTER to execute the statement again.

Clear

You can “clear” the interactions tab by pressing the **Clear** button. The items on the workbench are still available and text you entered is still available via the up and down keys.

End

You can “end” an interactions session by pressing the **End** button. This clears the Workbench but does not clear the interactions tab. The text you entered is still available via the up and down keys.

Now let’s declare and assign two String variables in the Interactions tab. Key in each line below and be sure and press ENTER at the end of each line. You should see the results as shown below in Figure 2-10.

```
String s1 = "Hey";
String s2 = "There";
s1 + " " + s2
```

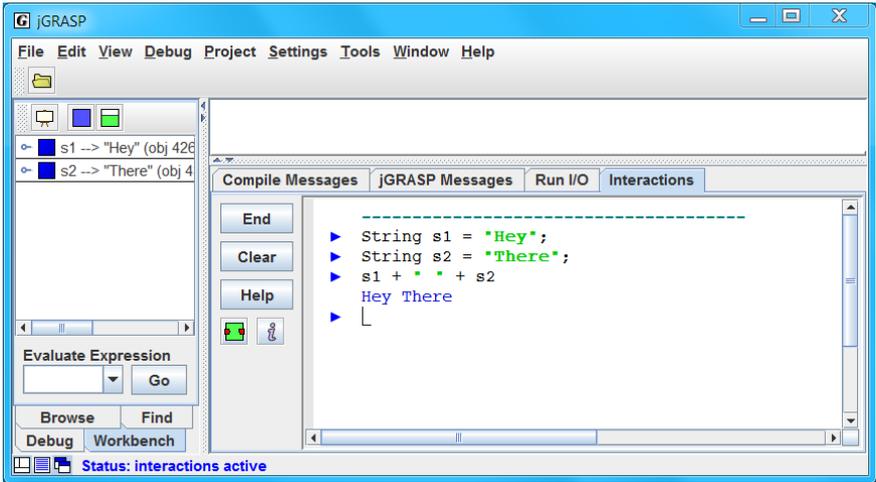


Figure 2-10. Declaring and assigning String variables

When you want to continue a statement on the next line, you can delay execution by pressing Shift-ENTER rather than ENTER. For example, you would need to press Shift-ENTER after the first line below and ENTER after the second line.

```
System.out.println
```

Shift-ENTER

```
("Hello\n\tfrom\n\t\tInteractions");
```

ENTER

If you simply press ENTER at the end of the first line, Interactions will attempt to execute the incomplete statement and you will get an error message. Figure 2-11 shows the statements above with delayed execution.

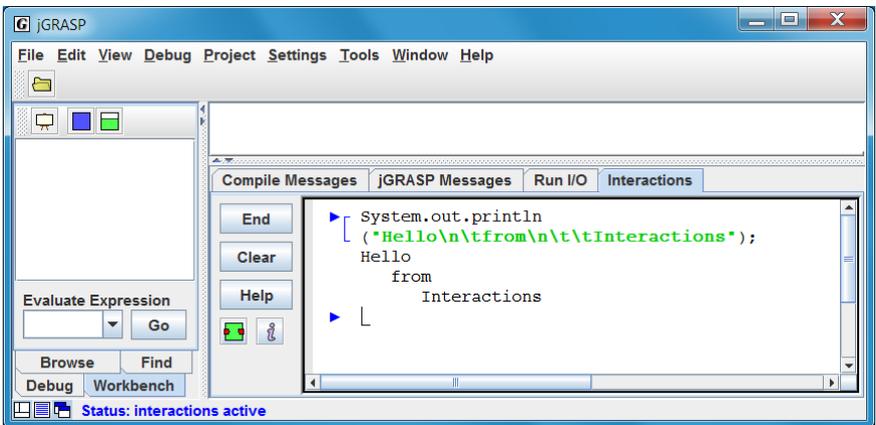


Figure 2-11. Multiple line statement with delayed execution

2.10 Using Viewers and the Viewer Canvas (Java only)



In this section, we'll look two simple programs with variables and run them in the **viewer canvas**. For a more in-depth description of the viewer canvas, see the separate tutorial entitled *Using the Viewer Canvas*.

You can open dynamic viewers on any objects or primitives in the Debug tab or the Workbench tab. If the viewers are placed on the viewer canvas, the saved canvas file can be used to run the program in the canvas which will automatically reopen the viewers when you "play" the canvas and/or step through the program using the debug controls.

In the Browse tab, navigate to the folder in the examples called DinnerGroup and double-click to open it. Then double-click on DinnerGroup.java to open it. Figure 2-12 shows the desktop after the program DinnerGroup.java has been opened, compiled, and run in the usual way.

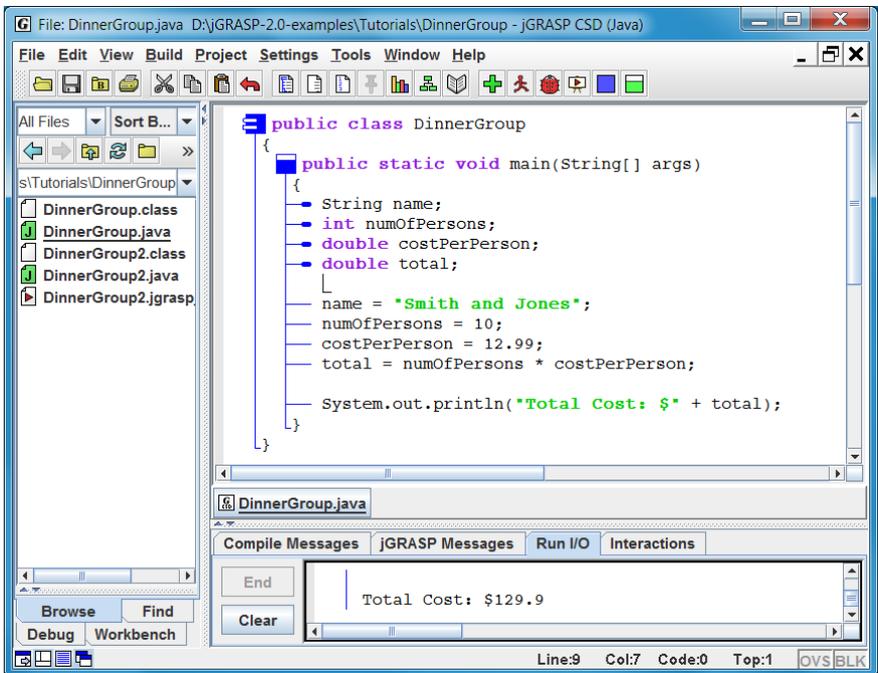


Figure 2-12. Desktop after the DinnerGroup.java has been run

Now click the Run in Canvas button  on the toolbar. You should see an empty viewer canvas window as shown in Figure 2-13 below. In the CSD window, you should see that the program is stopped at the first executable statement (`name = "Smith and Jones";`).

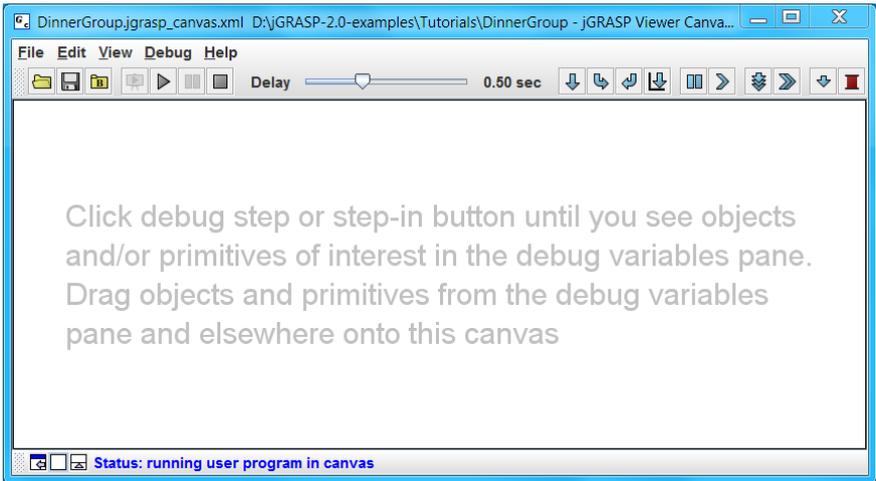


Figure 2-13. Empty viewer canvas window

As indicated by the message in the window, click the debug step button  to execute the statement. You should now see the variable name in the Debug tab. Now select and drag name onto the canvas (i.e., left-click the mouse on name and hold to drag name onto the canvas). Now click the step button  three more times so that you see numOfPersons, costPerPerson, and total in the Debug tab. Finally, drag each of these variables into the canvas window and arrange them as you like. Figure 2-14 shows one particular arrangement.

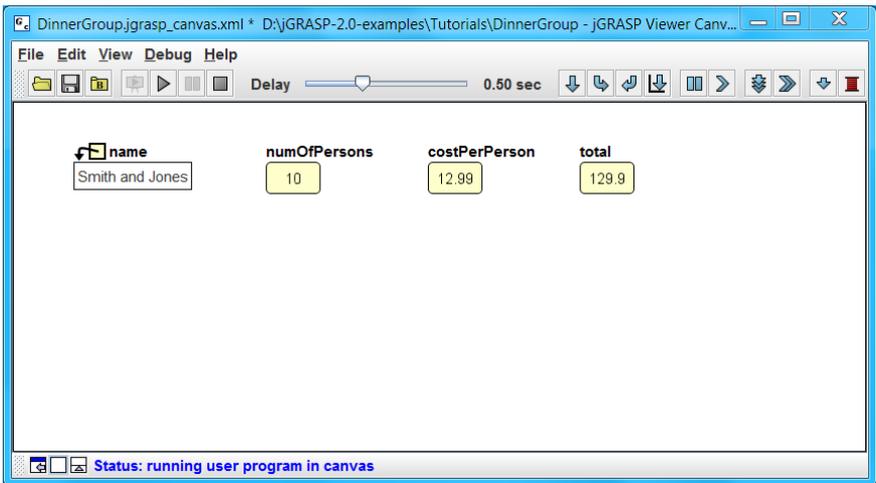


Figure 2-14. Viewer canvas window with variables from DinnerGroup

Click the Save button  on the canvas toolbar. You have now constructed and saved your first canvas. If the program is still running, click the **End** button. Now let's run the canvas from the beginning. Click the Run in Canvas button  on the toolbar, as you did before. Note that jGRASP will open your saved canvas if it is not already opened. Your program should now be running and stopped at the first executable statement. This time, instead of the clicking the step button, 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. Delay  0.30 sec. While the canvas can provide a useful program visualization, to fully understand a program, you should try to relate each "step" in the program with what is happening on the canvas.

Now let's take look at the other program in the DinnerGroup folder. Double-click on DinnerGoup2.java to open it. DinnerGoup2 is similar to DinnerGroup except that it reads in the values for each variable using a Scanner object on System.in (the keyboard). Run the DinnerGroup2 program . Figure 2-15 shows the program running and waiting for user to enter the name of the group. Enter the value for each variable so that program completes.

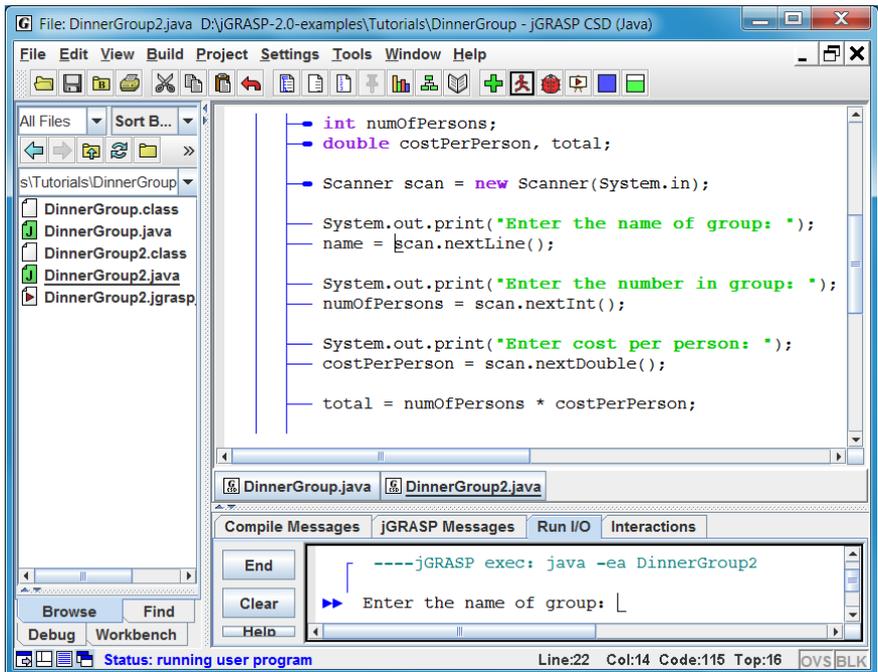


Figure 2-15. DinnerGroup2.java running and waiting for input

Now let's run DinnerGroup2 in the canvas using the existing canvas file. Click the Run in Canvas button  on the toolbar and canvas window opens with variables for `scan`, `name`, `numOfPersons`, `costPerPerson`, and `total` grayed out. The viewer for the variable `scan` depicts the input buffer of the Scanner object so that we can see the input from the keyboard as it is scanned. The small green triangle indicates the next character that will be consumed. Click the “play” button and the program should step until it requires input. As you enter the input for each variable, the Scanner viewer will advance to the next token and the variable associated with the input will be updated as shown in Figures 2-16 and 2-17 below. An astute observer will note that the `\r\n` for the last line entered by the user has not yet been consumed by the Scanner object.

You should now be ready to run your own programs in the canvas! For more information, see the separate tutorial entitled *Using the Viewer Canvas*.

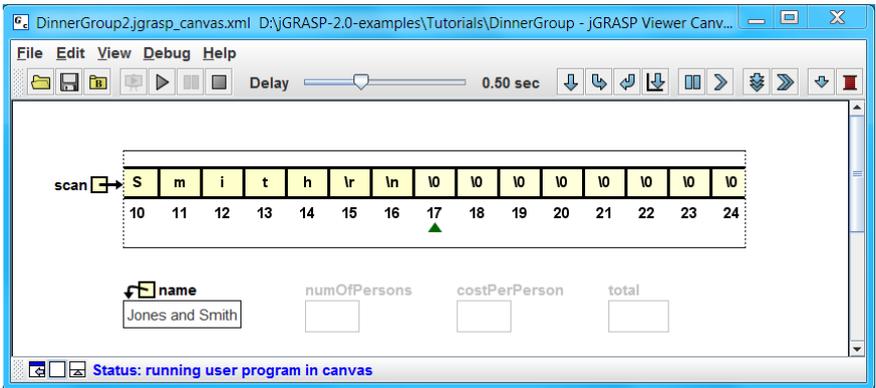


Figure 2-16. Canvas after reading/assigning name in DinnerGroup2

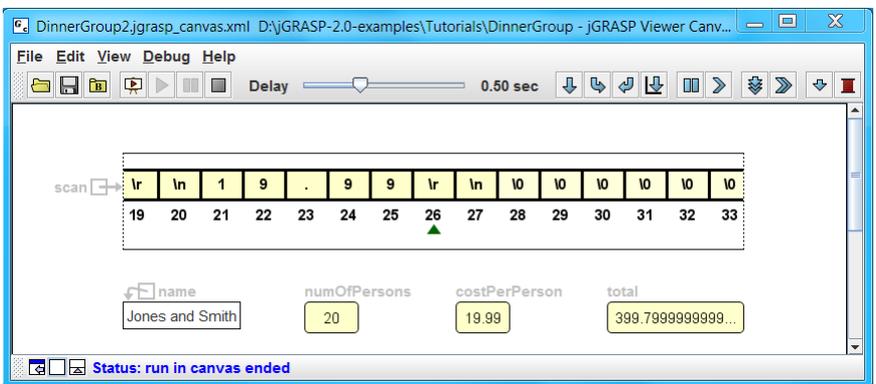


Figure 2-17. Canvas after a “run in canvas” for DinnerGroup2 has ended

2.11 Creating and Using Projects

When your program consists of two or more files, you should strongly consider creating a jGRASP project (.gpj) and then adding your program files to the project. For C, C++, Objective-C, and Ada, having all source files for your program in a project allows multiple files to be compiled and linked, and allows you to have separate source and executable directories if desired. For Java, there are many additional advantages: (1) allows jGRASP to make sure all files in the project have been compiled, (2) allows you to generate Javadoc documentation for your project, (3) allows you to generate a UML class diagram, and (4) when using plug-ins such as Checkstyle and JUnit, allows you see the respective status for each file in the project.

In the Browse tab, navigate to the folder in the examples called BookList and double-click to open it. Then double-click on BookList.java to open it. As you can see, the main method creates three Book objects and then prints them out.

Let's create a project for the BookList program. On the top menu, click **Project** > **New** to open the Create Project dialog. Now enter a project name (e.g., BookList) as shown in Figure 2-18.

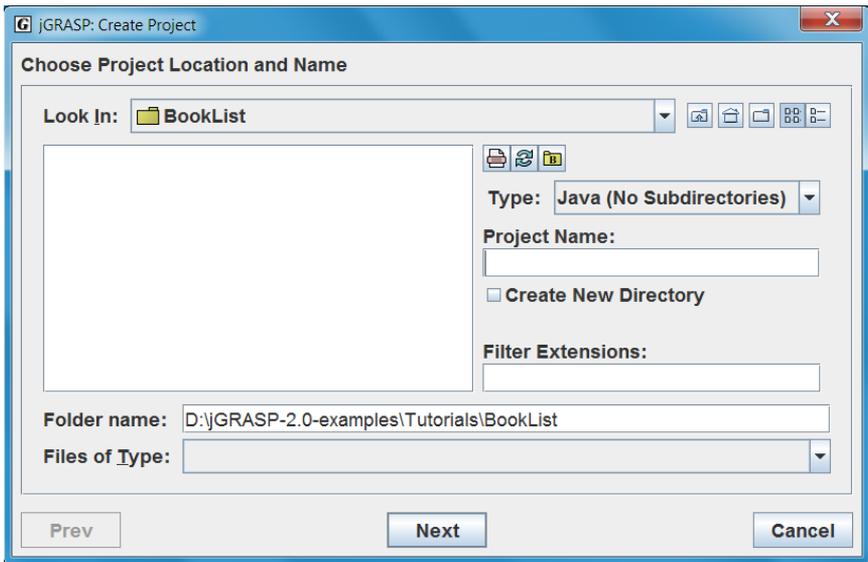


Figure 2-18. Create Project dialog – “BookList” entered for Project Name

After entering the project name, click the **Next** button at the bottom of the dialog. On the next page of the dialog, click the Create button. This should open the Add Source Files dialog as shown in Figure 2-19.

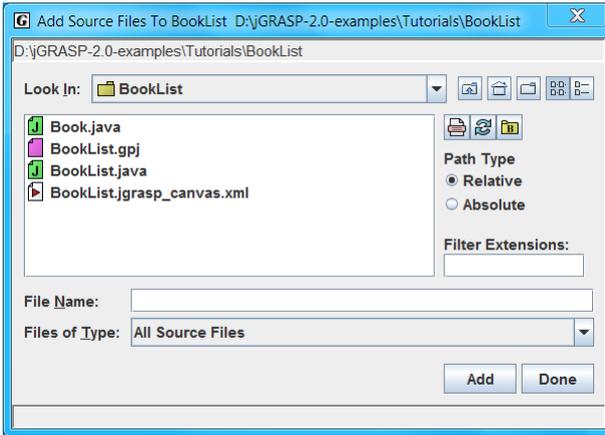


Figure 2-19. Add files dialog after BookList project has been created

Select Book.java and click the **Add** button, select BookList.java and the click **Add** button, and finally, click the **Done** button. You should now see the BookList project in the **Open Projects** section of the Browse tab (Figure 2-20).

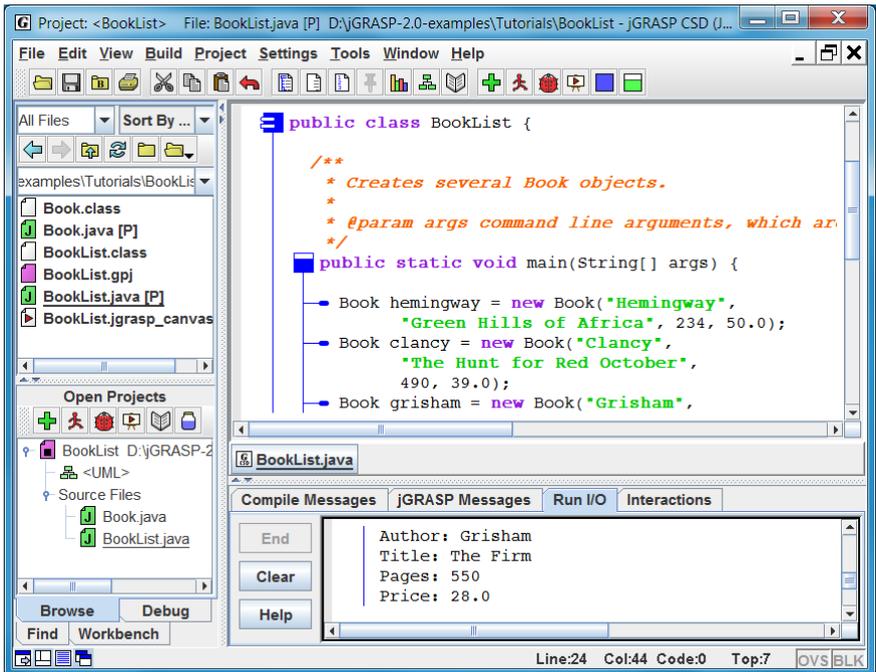


Figure 2-20. Desktop after the BookList project has been opened and the program has been compiled and run

In the file section of the Browse tab, the files in the project are marked with [P] after the file name. In the Open Projects section, a project can be folded and unfolded as needed. The *active* project has a black square in the project symbol.

Add a file to a project by dragging the file from the files section of the Browse tab and dropping it in a project in the Open Projects section. You can also right-click on the project and select Add Files to open the Add Files dialog.

Remove a file from a project by right-clicking on the project and selecting Remove From Project.

Close a project by right-clicking on the project and selecting Close.

Delete a project by right-clicking on the project file in the files section of the Browse tab and selecting Delete.

Open an existing project by double-clicking on the project file in the files section of the Browse tab (e.g., BookList.gpj).



Notice that **Open Projects** has its own toolbar. These buttons **compile**  all files in the project, **run**  the project's main method or applet, **debug**  the project's main method or applet, **run in canvas**  for the project's main method or applet, **generate documentation**  for the project, and **create a jar or zip file**  for the project. These work for the *active* project.

Now let's run the BookList project in the canvas by clicking the **run in canvas** button  on the Open Projects toolbar. After the canvas window opens with the existing canvas file, click the play button  to see the three Book objects as they are created by stepping into the Book constructor. You can speed up the program by decreasing the delay between steps with Delay slider. Figure 2-21 shows the canvas after the program has ended.

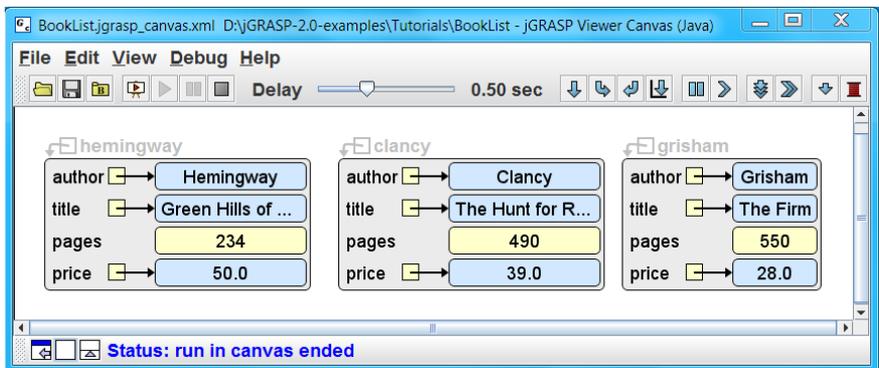


Figure 2-21. Canvas after Run in Canvas for BookList project has ended

To wrap up this introduction to projects, let's generate the documentation for the BookList project by clicking the **generate documentation** button  on the Open Projects toolbar. This uses the *javadoc* utility to generate a set of HTML files from the javadoc comments in your source files. These HTML files are saved in a folder called BookList_doc, which you should see in the Browse tab after you have successfully generated the project documentation. The index.html file should open automatically in your default Internet browser when the generate operation completes as shown in Figure 2-22 below. The classes in the project (**Book** and **BookList**) are listed alphabetically on the left with hyperlinks to their respective documentation. The information for the first class in the list (**Book**) is shown in the right pane. Clicking on **author** in the **Field Summary** in the right pane will take you to the documentation for author, the first field in the Book class.

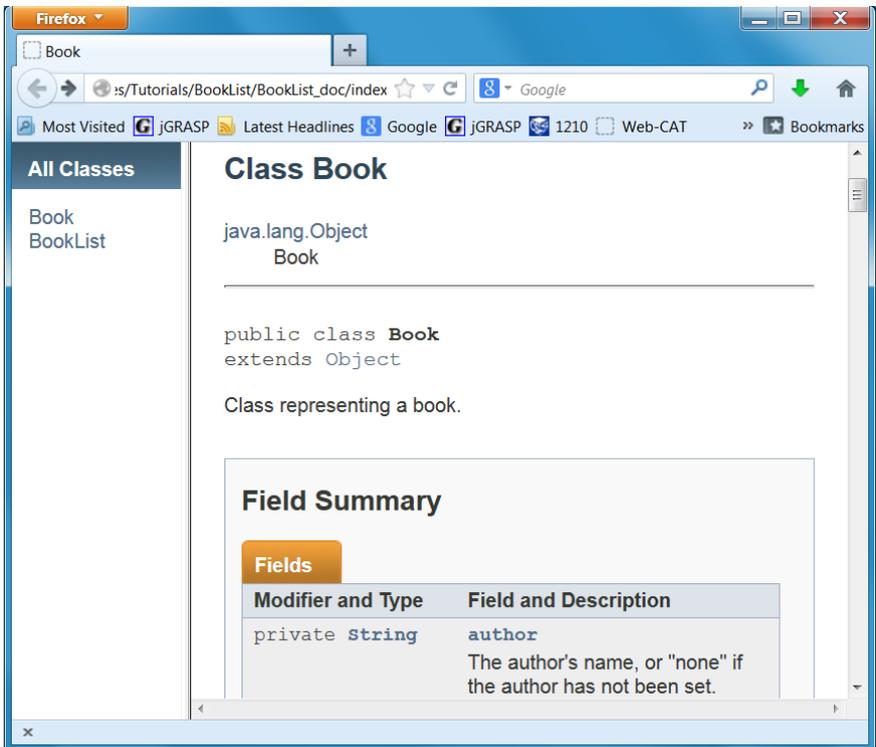


Figure 2-22. Browser with documentation for BookList project

2.12 Generating and Using a UML Class Diagram (Java only)

When your program consists of multiple java files, a UML class diagram can be very useful to show the dependencies among your classes (i.e., how your classes relate to one another). In order to generate a UML class diagram, your source files must be in a project as described in the previous section. The UML class diagram can then be generated from the Java files in the project. You can have a project with only one source file containing a single Java class, but the UML class diagram will not be very interesting.

Let's consider the **BookList** project that we created in the previous section. If the project is not already open, double-click on the BookList.gpj file in the files section of the Browse tab. With the BookList project in the Open Projects section, generate the UML class diagram by double-clicking on the UML symbol below the project name or by single-clicking the UML button on the desktop toolbar. This will open the UML window with the generated diagram as shown in Figure 2-23 below.

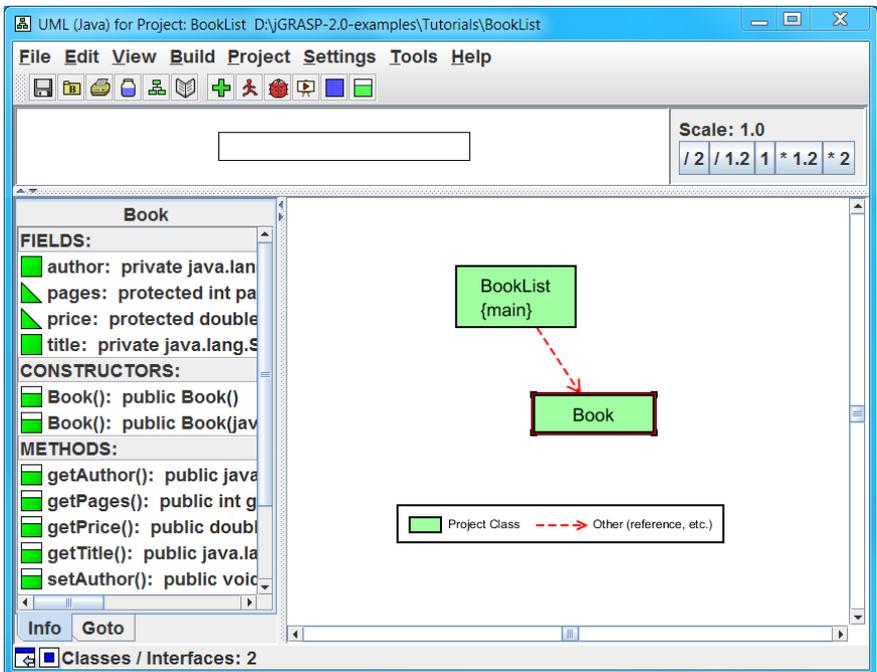


Figure 2-23. UML class diagram for the BookList project

When the Book class is selected, as shown above by the red outline, its members (fields, constructors, and methods) are displayed in the **Info** tab to the left. Double-clicking on a member will take you to the member's definition in the

source file. For example, double-clicking on the `getAuthor()` method will take you to the first executable statement in the `getAuthor()` method in the `Book` class. When the red dependency arrow between the two classes is selected, the actual dependencies are displayed in the Info tab.

 You can move the UML window into or out of the desktop and/or keep the window “always-on-top” with the buttons in the lower left corner of the window. If you are using the UML window to navigate in your files, you’ll likely want it outside of the desktop and positioned so that your code is not obscured. However, if you are primarily using the UML class diagram to create instances of classes for the workbench and invoke their methods, you may want to move the UML window into the desktop. You will then have the Workbench tab with the objects conveniently located on the left and the Interactions tab below the UML window as shown in Figure 2-24. Note that you can right-click on `Book` in the UML window and select **Create New Instance** in order to create an instance (e.g., `book_1`) for the workbench as indicated in the figure. You then can right-click on `book_1` to invoke its methods. You can also enter statements and expressions in the Interactions tab as well as open a canvas window and then drag `book_1` onto it to see it in a viewer. For example, entering the expression `book_1` in the Interactions tab prints its `toString()` value.

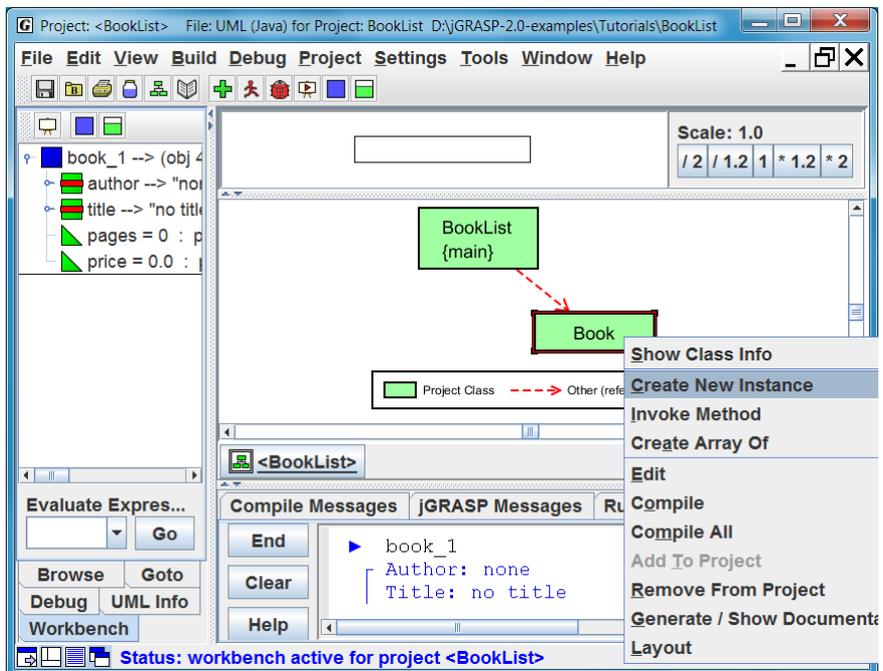


Figure 2-24. UML class diagram for the `BookList` inside the desktop

2.13 Opening a File – Additional Options

A file can be opened in a CSD window in a variety of ways. Each of these is described below.

- (1) **Browse Tab** - If the file is listed in jGRASP Browse tab, you can simply double click on the file name, and the file will be opened in a new CSD window. We did this back in section **2.2 Quick Start**. You can also drag a file from the Browse tab and drop it in the CSD window area.
- (2) **Menu or Toolbar** - On the menu, click **File > Open** or Click the Open File button  on the toolbar. Either of these will bring up the Open File dialog shown in Figure 2-25.

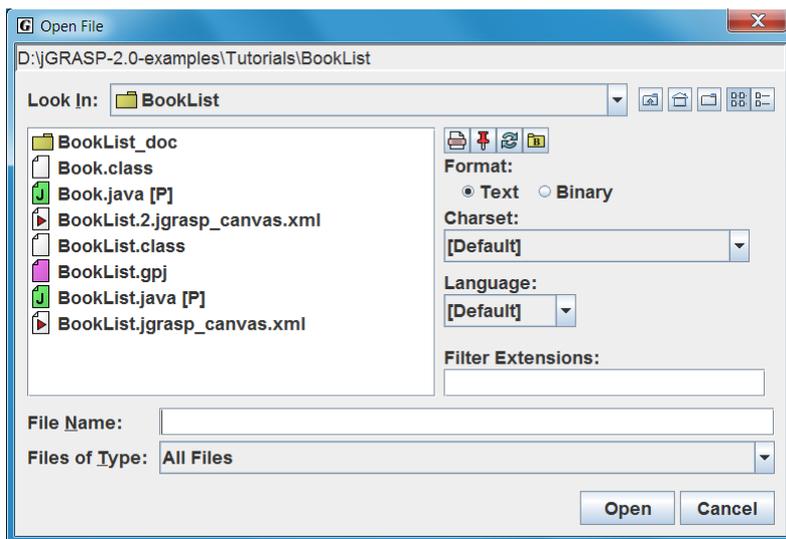


Figure 2-25. Open File dialog

- (3) **Windows File Browser** - If you have a Windows file browser open (e.g., My Computer, My Documents, etc.), and the file is marked as a jGRASP file, you can just double click the file name.
- (4) **Windows File Browser (drag and drop)** - If you have a Windows file browser open (e.g., My Computer, My Documents, etc.), you can drag a file from the file browser to the jGRASP Desktop and drop it in the area where the CSD window would normally be displayed.

In all cases above, if a file is already open in jGRASP, the CSD window containing it will be popped to the top of the Desktop rather than jGRASP opening a second CSD window with the same file.

Multiple CSD Windows – When you have multiple files open, each is in a separate CSD window or internal frame which may be maximized (default) or unmaximized. Each program can be compiled and run from its respective CSD window. When multiple windows are open, the single menu and toolbar are active for the top window only, which is said to have “focus” in the desktop. In Figure 2-26, two CSD windows have been opened (unmaximized) – one with DinnerGroup.java and the other with DinnerGroup2.java. If the window in which you want to work is visible, simply click the mouse on it to bring it to the top. If you have many windows open, you may need to click the **Window** menu, and then click the file name in the list of the open files. However, the easiest way to give focus to a window is to click the window’s button on the *windowbar* below the CSD window. These buttons can be reordered by dragging/dropping them on the windowbar as described earlier. In the figure below, the windowbar has buttons for DinnerGroup and DinnerGroup2. Notice that DinnerGroup2.java is underlined both on the windowbar and in the Browse tab to indicate that it has the current focus. DinnerGroup2.java is also displayed in the desktop’s blue title bar.

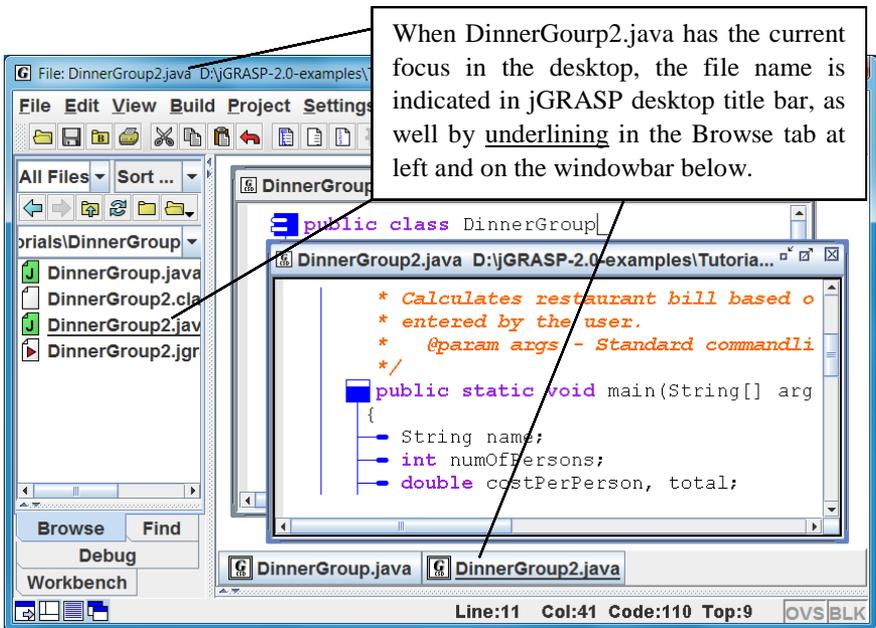


Figure 2-26. Multiple files open

2.14 Closing a File

The open files in CSD windows can be closed in several ways.

- (1)  If the CSD window is maximized, you can close the window and file by clicking the Close button at the right end of the top level Menu.
- (2)  If the CSD window is not maximized, click the Close button in the upper right corner of the CSD window itself.
- (3) **File Menu** – Click **File > Close** or **Close All Files**. The latter closes all CSD windows. To close “all” UML windows, you need close all projects by clicking **Projects > Close All**.
- (4) **Window Menu** – Click **Window > Close All Windows**.

In each of the scenarios above, if the file has been modified and not saved, you will be prompted to *Save and Exit*, *Discard Edits*, or *Cancel* before continuing.

2.15 Exiting jGRASP

When you have completed your session with jGRASP, you should always close (or “exit”) jGRASP rather than let your computer close it when you log out or shut down. However, you don’t have to close the files you have been working on before exiting jGRASP. When you exit jGRASP, it remembers the files you have open, including their window size and scroll position, before closing them. If a file was edited during the session, jGRASP prompts you to save or discard the changes. The next time you start jGRASP, it will open your files, and you will be ready to begin where you left off. For example, open the Hello.java file and then exit jGRASP by one of the methods below. After jGRASP closes down, start it up again and you should see the Hello.java program in a CSD window. This feature is so convenient that many users tend to leave a few files open when they exit jGRASP. However, if a file is really not being used, it is best to go ahead and close the file to reduce the clutter on the windowbar.

Close jGRASP in either of the following ways:

- (1) Click the Close button  in the upper right corner of the desktop; or
- (2) On the File menu, click **File > Exit jGRASP**.

2.16 Review and Preview of What's Ahead

As a way of review and also to look ahead, let's take a look at the jGRASP *toolbar*. Hovering the mouse over a button on the toolbar provides a "tool hint" to help identify its function. Also, **View > Toolbar Buttons** allows you to display *text* labels on the buttons. Figure 2-27 shows the name/function of each button.

While many of these buttons were introduced in this section, some were assumed to be self-explanatory (e.g., Print, Cut, Copy, etc.), and several others will be covered in other tutorials).

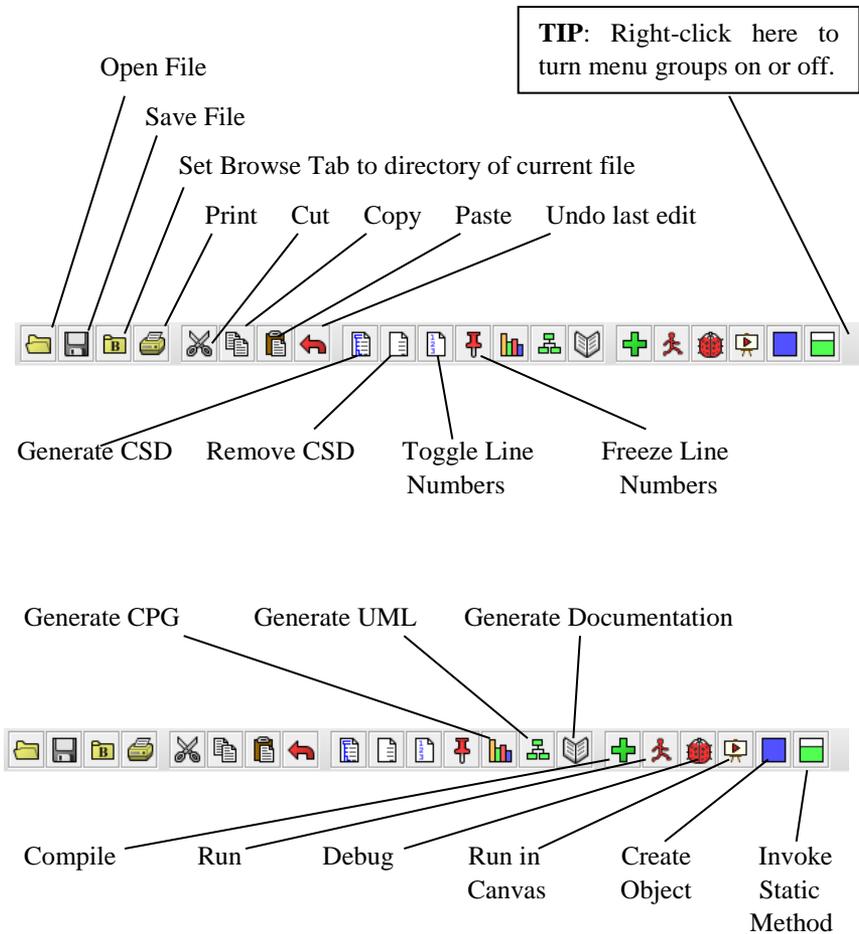


Figure 2-27. Toolbar

2.17 Exercises

- (1) Create your own program then save, compile, and run it. Alternative, if already have a program saved in a file, you can open the file, then compile and run it in a CSD window.
- (2) Enter several statements and expressions in **Interactions** to immediately see the results of their execution and/or evaluation.
- (3) Generate the CSD for your program. On the View menu, turn on Auto Generate CSD (**Settings > CSD Window Settings** – then (checkbox) **Auto Generate CSD**).
- (4) Turn line numbers *on* and *off* for your program. Decide which you prefer.
- (5) Fold your program and then unfold it in layers. Fold your program and then unfold it all at once by clicking the same symbol you clicked to fold it.
- (6) On the Build menu, make sure Debug Mode is *on* (indicated by a check box). [Note that Debug Mode should be *on* by default, and we recommend that this be left *on*.] Compile your program.
- (7) Set a breakpoint on the first executable line of your program then run it with the debugger. Step through each statement, checking the Run I/O window for output.
- (8) Run your program in the canvas. Click step until you see variables of interest and then drag each variable onto the canvas. Arrange the viewers then save the canvas. Run in the canvas again and then click the play button. Alternatively, use the debug controls to *step* or *step-in*.
- (9) If you have program consisting of multiple files, create a project and add the Java files. Otherwise, use the BookList program. Generate the UML class diagram and create one more objects and invoke methods as appropriate.
- (10) If you have other Java programs available, open one or more of them, then repeat steps (1) through (9) above for each program.

Notes