

3 Getting Started with Objects

If you are an experienced IDE user, you may be able to do this tutorial without having done the previous tutorial, *Getting Started*. However, at some point you should read the previous tutorial and make sure you can do the exercises at the end. The topics presented in this tutorial are applicable to Java.

Objectives – When you have completed this tutorial, you should be able to use projects, UML class diagrams, the Object Workbench, Viewers, and Interactions in jGRASP. These topics are especially relevant for an *objects first* or *objects early* approach to learning Java.

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

- 3.1 Starting jGRASP
- 3.2 Navigating to Our First Example Project
- 3.3 Opening a Project and UML Window
- 3.4 Compiling and Running the Program from UML Window
- 3.5 Exploring the UML Window
- 3.6 Viewing the Source Code in the CSD Window
- 3.7 Exploring the Features of the UML and CSD Windows
- 3.8 Generating Documentation for the Project
- 3.9 Using the Object Workbench
- 3.10 Opening a Viewer Window
- 3.11 Invoking a Method
- 3.12 Invoking Methods with Parameters That Are Objects
- 3.13 Invoking Methods on Object Fields
- 3.14 Showing Categories of Methods
- 3.15 Creating Objects from the CSD Window
- 3.16 Using Interactions
- 3.17 Running the Debugger on Invoked Methods
- 3.18 Creating an Instance from the Java Class Libraries
- 3.19 Exiting the Workbench
- 3.20 Closing a Project
- 3.21 Exiting jGRASP
- 3.22 Review of Toolbar Buttons
- 3.23 Exercises

3.1 Starting jGRASP

A Java program consists of one or more class files. During the execution of the program, object instances can be created and then manipulated toward some useful purpose by invoking the methods provided by their respective classes. In this tutorial, we'll examine a simple program called PersonalLibrary that consists of five Java classes. In jGRASP, these five Java files are organized as a project.



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 are working on a PC in a computer lab and you do not 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 across the top and three panes. The *left pane* has tabs for **Browse**, **Find**,

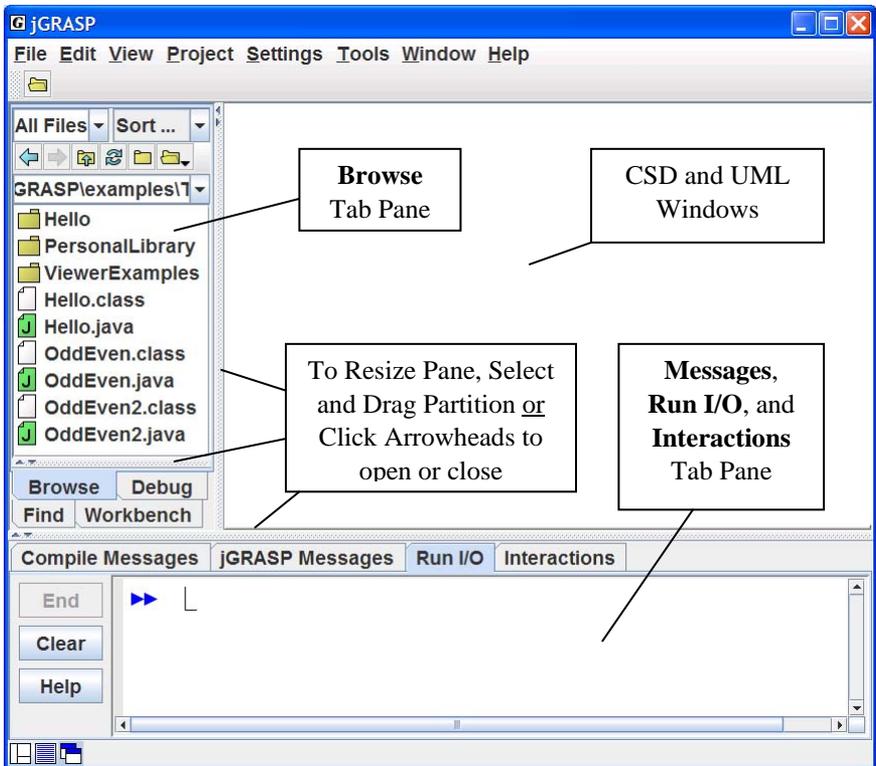


Figure 3-1. The jGRASP Virtual Desktop

Debug, and Workbench. The large *right pane* is for UML and CSD windows. The *lower pane* has tabs for jGRASP messages, Compile messages, Run Input/Output, and Interactions.

3.2 Navigating to Our First Example Project

Example programs are available in the jGRASP folder in the directory where it was installed (e.g., C:\Program Files\jGRASP\examples\Tutorials). You should copy the Tutorials folder to one of your own folders (e.g., in your *My Documents* folder) so that any changes you make will not be lost when jGRASP is upgraded.

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 Browse tab or by clicking on the up-arrow as indicated in the figure below. The left-arrow and right-arrow allow you to navigate *back* and *forward* to directories that have already been visited during the session. The refresh button  updates the Browse pane. In the example below, the Browse tab is displaying the contents of the Tutorials folder.

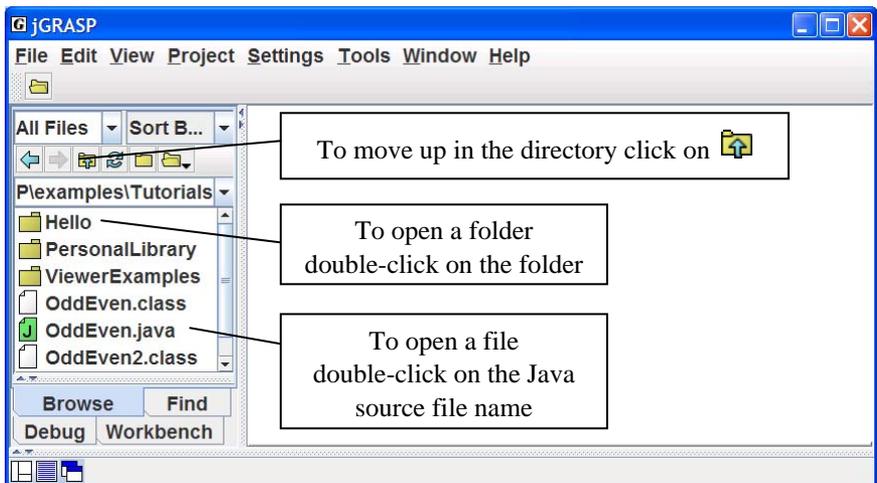


Figure 3-2. The jGRASP Virtual Desktop

3.3 Opening a Project and UML Window

After double-clicking the PersonalLibraryProject folder, the Java source files in the project as well as the jGRASP project file are displayed in the Browse tab. Double-click on the project file (PersonalLibraryProject.gpj) to open the project as shown in **Step 1** below. After the project is opened, the Browse tab is split into two sections, the upper section for files and the lower section for open projects, as shown below in Figure 3-3.

We are now ready to open a UML window and generate the class diagram for the project. As indicated in **Step 2** below, simply double-click on the UML symbol shown beneath the project name in the open projects section of the Browse tab. Alternatively, on the desktop menu you can click **Project > Generate/Update UML Class Diagram**.

After you have opened the UML window, you can compile and run your program in the traditional way using the toolbar buttons or the Build menu. However, from an *objects first* perspective, you can also create objects directly from your classes, place them on the Workbench, and then invoke their methods. Both of these approaches are explored below.

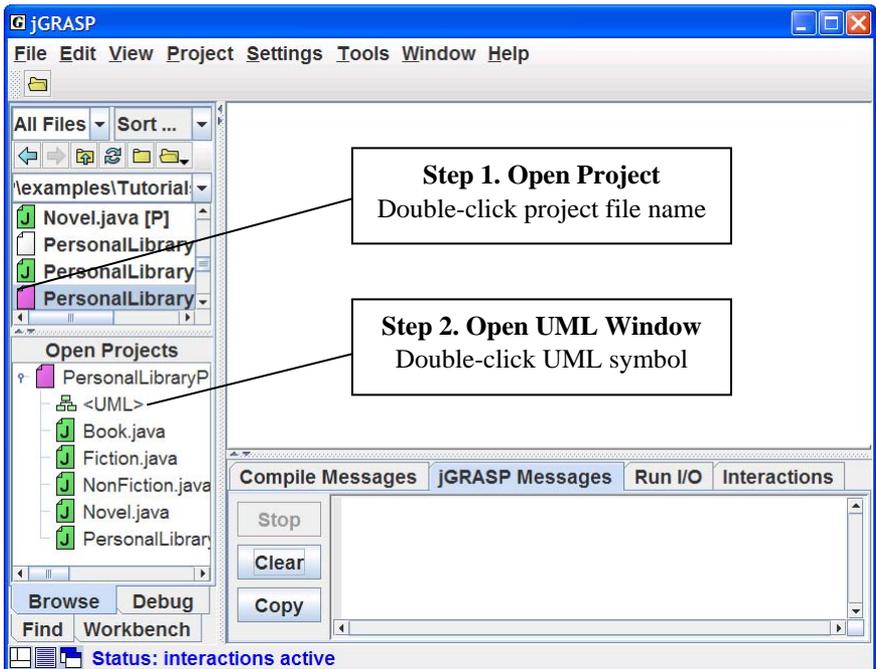


Figure 3-3. Opening a project file and UML window

3.4 Compiling and Running the Program from UML Window

You can compile the files in the UML window by clicking the green plus **+** as indicated in **Step 3** below. Note that the classes in the UML diagram become crosshatched with red lines when they need to be recompiled. After a successful compile, the classes should be green again. If at least one the classes in the diagram has a *main* method, you can also run the program by clicking the Run button **⚡** as shown by **Step 4**. When you compile or run the program, the respective Compile Messages or Run I/O tab pops open in the lower pane to show the results.

TIP: Usually the reason for compiling a program is because you have modified or “added” something, hence the green plus **+**.

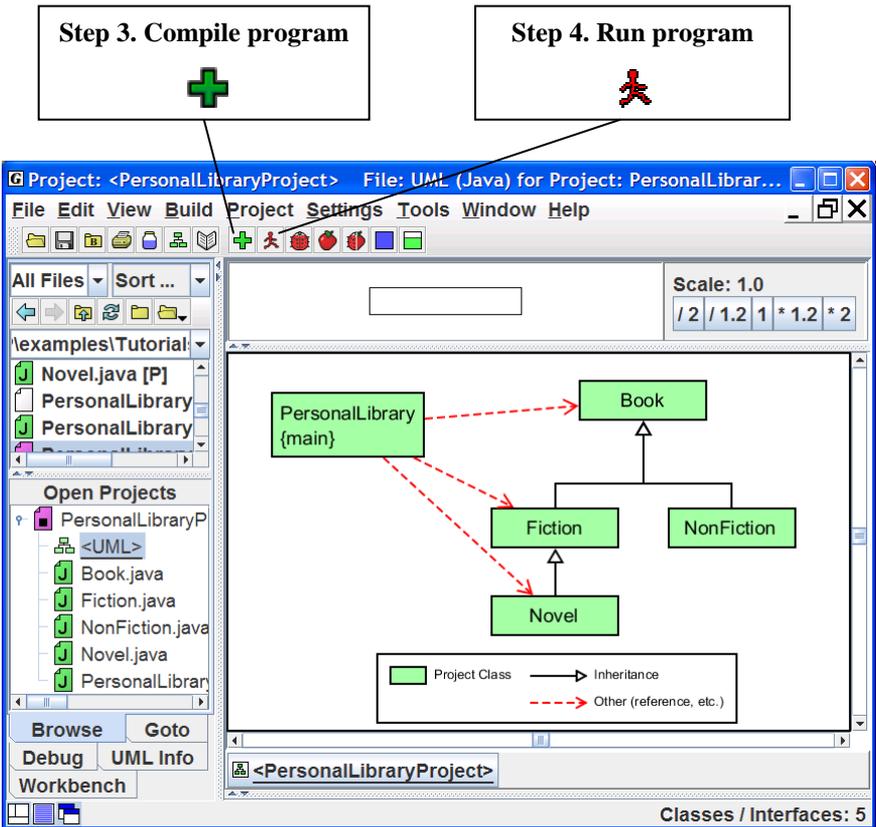


Figure 3-4. After loading file into CSD Window

3.5 Exploring the UML Window

In the Figure 3-5, a UML window for the PersonalLibraryProject has been opened and the class diagram has been generated. Below the toolbar is a panning rectangle which can be used to move around in the UML diagram. A set of scaling buttons is located to the right of the panning rectangle. Try clicking each of the scaling buttons one or more times to see the effect on the UML diagram. Clicking “1” resets the diagram to its original size. The **Update UML** button on the toolbar can be used to regenerate the diagram in the event any of the classes in the project are modified outside of jGRASP (e.g., edited or compiled). Just below the UML window is the windowbar which contains a button for each UML or CSD window that is opened. Clicking the button pops its window to the top. Windowbar buttons can be reordered by dragging them around on the windowbar.

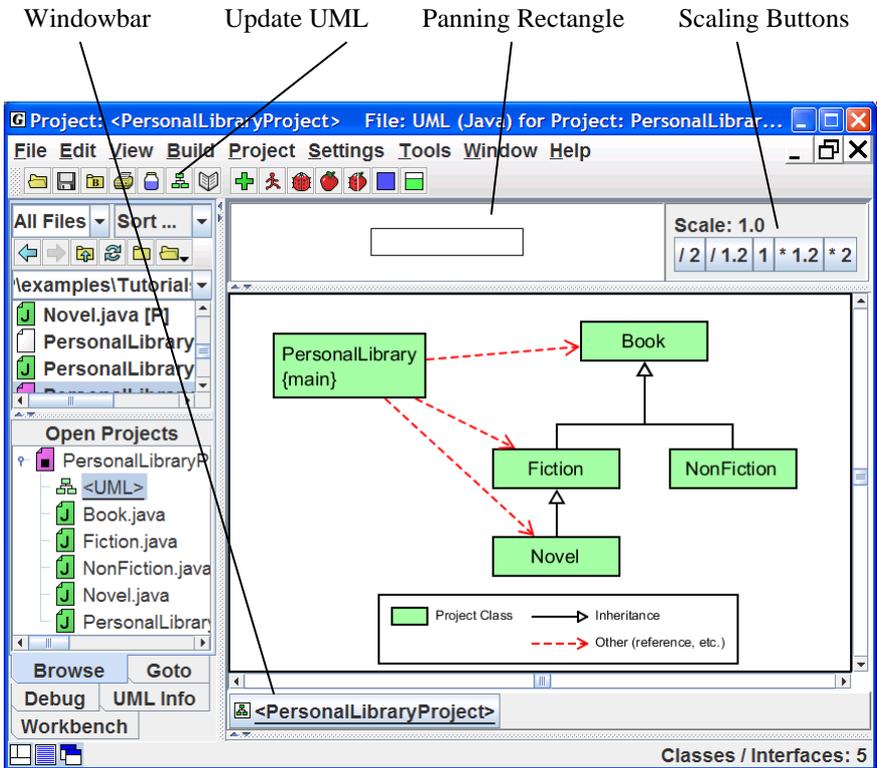


Figure 3-5. UML window with PersonalLibraryProject

3.6 Viewing the Source Code in the CSD Window

To view the source code for a class in the UML diagram, simply double-click on the class symbol, or in the Browse tab, double-click the file name in the Files or Open Projects sections. Each of these will open the Java file in a CSD window, which is a full-featured editor for entering and updating your program. Notice that with the CSD window open the toolbar buttons now include Generate CSD, Remove CSD, Number Lines, Compile, and Run, as well as buttons for Create Instance and Invoke Method.

Generate a CSD

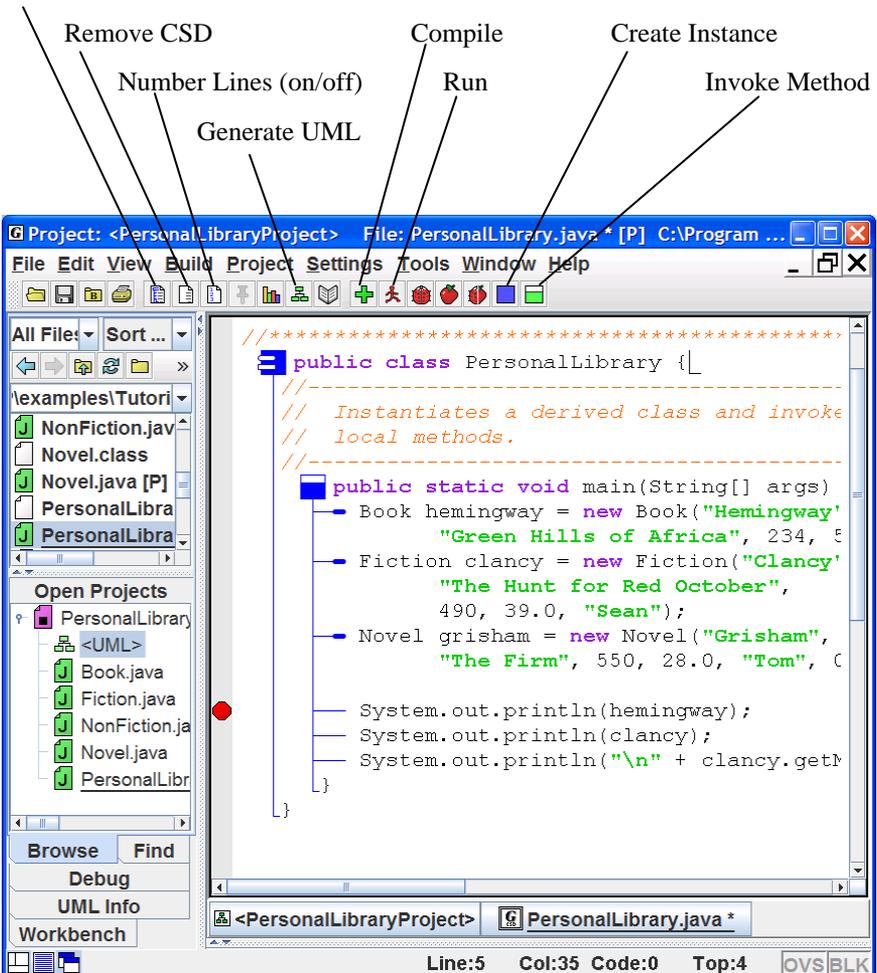


Figure 3-6. After the CSD is generated

3.7 Exploring the Features of the UML and CSD Windows

Once you have a UML window open with your class diagram, you are ready to do some exploring. The steps below are intended to give you a semi-guided tour of some of the features available from the UML and CSD windows.

3.7.1 Viewing the source code for a class

- (1) In the UML diagram, double-click on the PersonalLibrary class. This should open the source file in a CSD window. Notice a button for this CSD window is added to the windowbar. You should also see a button for the UML window.
- (2) Review the source code in the CSD window; generate the CSD; fold and unfold the CSD; turn line numbers on and off. [See Sections 2.7 - 2.9 in *Getting Started* for details.]
- (3) On the windowbar, click the button for the UML window to pop it to the top. *Remember to do this anytime you need to view the UML window.*
- (4) View the source code for the other classes by: (1) double-clicking on the class in the UML diagram, (2) double-clicking on the class in the Open Projects section of the Browse tab, or (3) double-clicking on the file name in the upper section of the Browse tab.
- (5) Close one or more of the CSD windows by clicking the **X** in the upper right corner of the CSD window.

3.7.2 Displaying class information

- (1) In the UML window, select the Fiction class by left-clicking on it.
- (2) Right-click on it and select Show Class Info. This should pop the **UML Info** tab to the top in the left pane of the Desktop, and you should be able to see the **fields**, **constructors**, and **methods** of the Fiction class.
- (3) In the UML Info tab, double-click on the getMainCharacter() method. This should open a CSD window with the first executable line in the method highlighted.
- (4) Close the CSD window by clicking the X in the upper right corner.

3.7.3 Displaying Dependency Information

- (1) In the UML window, select the arrow between PersonalLibrary and Fiction by left-clicking on it.
- (2) If the UML Info tab is not showing in the left pane of the desktop, right-click on the arrow and select Show Dependency Info. Alternatively, you can click the UML Info tab near the bottom of the left pane.

- (3) Review the information listed in the UML tab. As the arrow in the diagram indicates, PersonalLibrary uses a constructor from Fiction as well as the `getMainCharacter()` method.
- (4) Double-click on the `getMainCharacter` method. This should open a CSD window for PersonalLibrary with the line highlighted where the method is invoked.

3.8 Generating Documentation for the Project

With your Java files organized as a project, you have the option to generate project level documentation for your Java source code in a standard format. To begin the process of generating the documentation, click **Project > Generate Documentation**. Alternatively, if the UML window is in focus, click the Generate Documentation button  on the toolbar. This will bring up the “Generate Documentation for Project” dialog, which asks for the directory where the generated HTML files are to be stored. The default directory name is the name of the project with “_doc” appended to it. Thus, for the example, the default will be PersonalLibraryProject_doc. Using the default name is recommended so that your documentation directories will have a standard naming convention. However, you are free to use any directory as the target. Pressing the **Default** button will get you back to the default directory in the event a different directory is listed. When you click **Generate** on the dialog, jGRASP calls the javadoc utility, included with the JDK, to create a complete hyper-linked document. The documentation is then opened in a Documentation Viewer as shown below for PersonalLibraryProject.

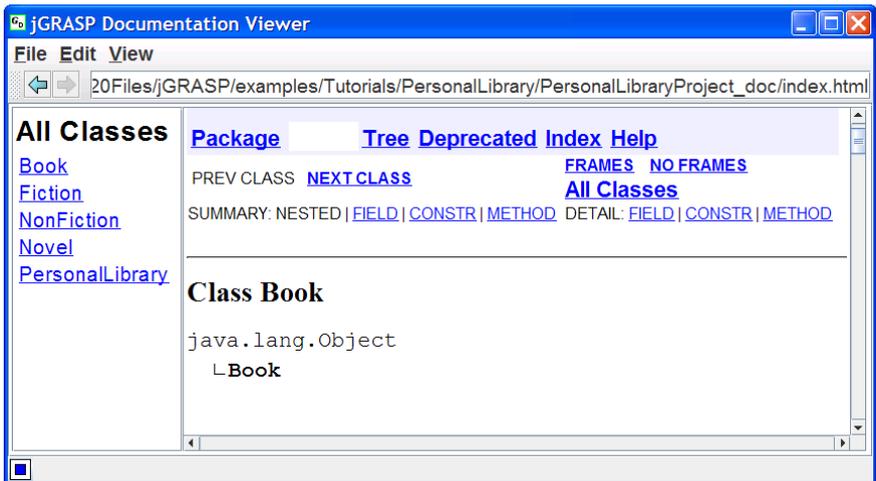


Figure 3-7. After generating documentation for PersonalLibraryProject

3.9 Using the Object Workbench

Now we are ready to begin exploring the Object Workbench. The figure below shows the UML window opened for the PersonalLibraryProject. Earlier, we learned how to run the program as an application using the Run button . Since *main* is a static method, we can also invoke it directly from the class diagram by right-clicking on PersonalLibrary and selecting **Invoke Method**. Alternatively, you can select the PersonalLibrary class, and then click the Invoke Method button  on the toolbar. When the Invoke Method dialog pops up, select and invoke *main* (without parameters). Try this now.

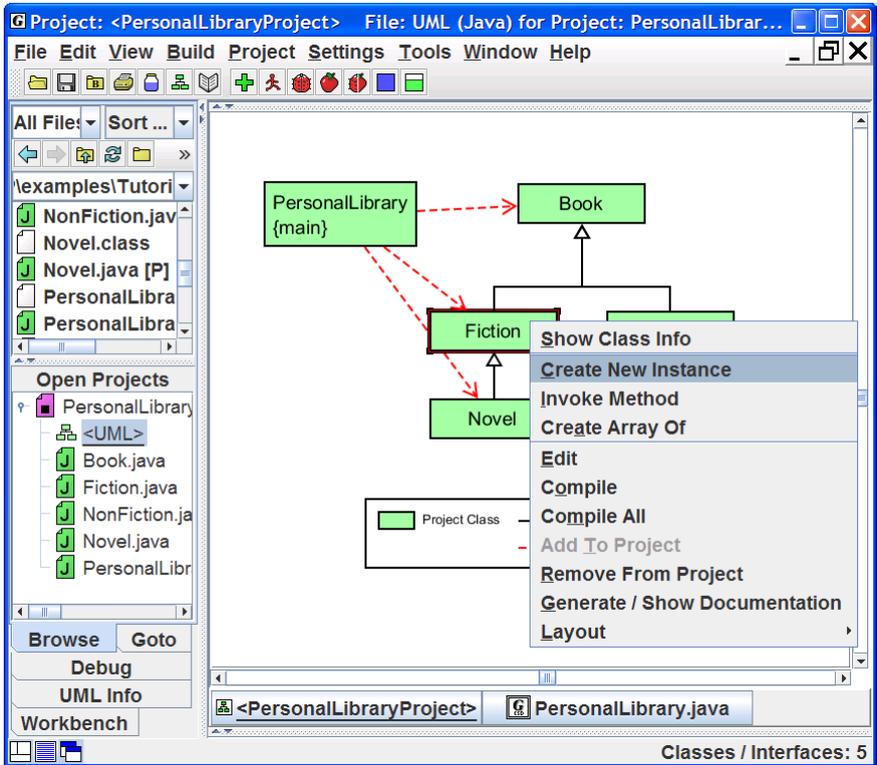


Figure 3-8. Creating an Object for the Workbench

The focus of this and the next several sections is on creating objects and placing them on the workbench. We begin by right clicking on the Fiction class in the UML diagram, and then selecting **Create New Instance**, as shown in Figure 3-8. Alternatively, select the Fiction class, and then click the Create Instance button  on the toolbar. A list of constructors will be displayed in a dialog box.

Click on “stick-pin”  to keep dialog open.



Figure 3-9. Selecting a constructor

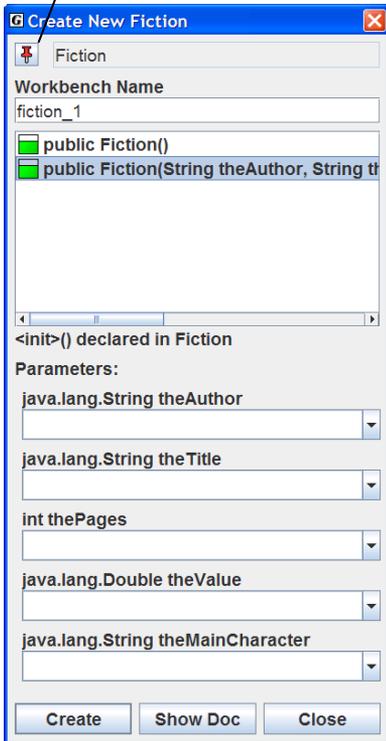


Figure 3-10. Constructor with parameters

If a parameterless constructor is selected as shown in Figure 3-9, then clicking **Create** will immediately place the object on the workbench. However, if the constructor requires parameters, the dialog will expand to display the individual parameters as shown in Figure 3-10. The values for the parameters should be filled in prior to clicking **Create**. Be sure to enclose strings in double quotes. In either case, the user can set the name of the object being constructed or accept the default assigned by jGRASP. Also, the “stick-pin”  located in the upper left of the dialog can be used to make the Create dialog remain open. This is convenient for creating multiple instances of the same class. If the project documentation has been generated, clicking the **Show Doc** button on the dialog will display the documentation for the constructor selected.

In Figure 3-11, the Workbench tab is shown after two instances of Fiction and one of Novel have been created. The

second object, `fiction_2`, has been expanded so that the fields (`mainCharacter`, `author`, `title`, and `pages`) can be viewed. An object can be expanded or contracted by clicking on its name. Notice that three fields in `fiction_2` are also objects (i.e., instances of the `String` class); they too can be expanded.

Notice that objects and object fields have various shapes and colors associated with them. Objects are represented by squares and primitives are represented by triangles. Top level objects are indicated by blue square symbols (e.g., `fiction_2`). The symbols for fields declared in an object are either a square for an object (e.g., `author`) or a triangle for a primitive type (e.g., `pages`). A green symbol indicates the field is declared within the class (e.g., `mainCharacter` in `fiction_2`), and an orange symbol means the field was declared in a superclass (e.g., `author` was declared in `Book`). A red bar on a symbol means the field is inaccessible from its current context; the object was declared as either private or protected (e.g., `mainCharacter`). A gray bar indicates the field is not visible and that a cast would be required to refer to it. Finally, a red-gray bar means the field is inaccessible and not visible. These colors/bars also apply to methods.

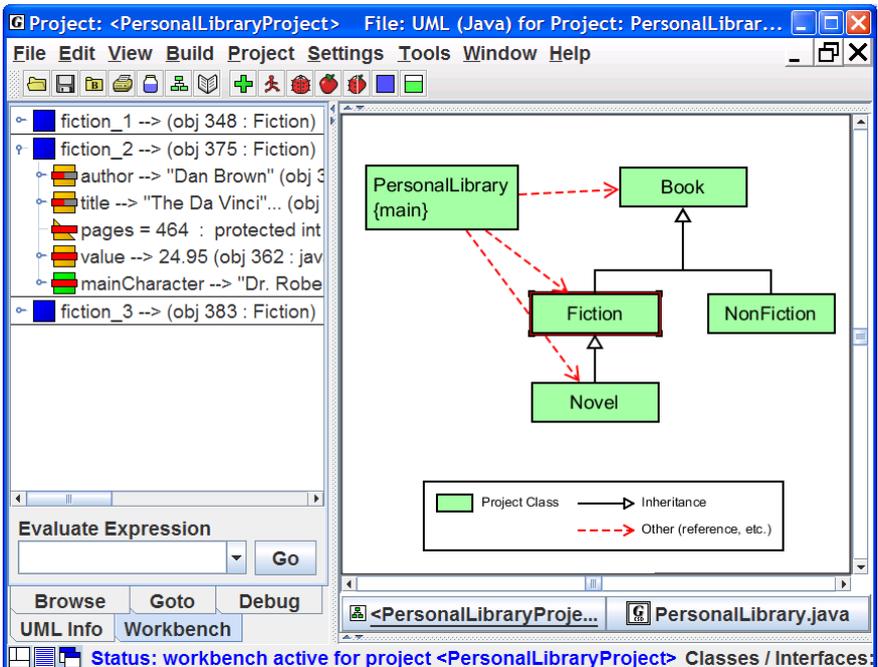


Figure 3-11. Workbench with three Fiction objects

3.10 Opening a Viewer Window

A separate *Viewer* window can be opened for any object or field of an object in the Workbench or Debug tabs. To open a viewer, left-click on an object in the Workbench tab and while holding down the left mouse button, drag it from the workbench to the location where you want the viewer to open. When you start to drag the object, a viewer symbol should appear to indicate a viewer is being opened. At a minimum, a viewer provides the *basic* view similar to the one in the Workbench and Debug tabs. However, some objects will have additional views. For example, the viewer for a String object will display its text value fully formatted. Figure 3-12 shows a viewer on the *mainCharacter* field in *fiction_2*.

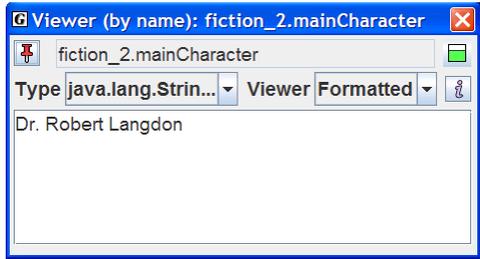


Figure 3-12. Viewer on *fiction_2.mainCharacter*

Figure 3-13 shows a viewer opened for *Basic* view on the “pages” field of *fiction_2*, which is an int primitive type. Figure 3-14 shows the viewer set to *Detail* view, which shows the value of pages in decimal, hexadecimal, octal, and binary. The Detail view for float and double values shows the internal exponent and mantissa representation used for floating point numbers. Note that the last view selected will be used the next time a Viewer is opened on the same class or type. Special presentation views are provided for instances of array, ArrayList, LinkedList, HashMap, and TreeMap. When running in Debug mode, a viewer can also be opened on any variable in the Debug tab.

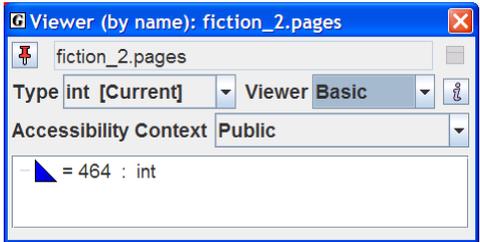


Figure 3-13 Viewer with *Basic* View of Primitive int

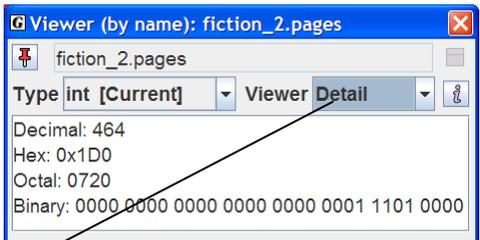


Figure 3-14 Viewer with *Detail* View of Primitive int

Select view from drop-down list.

Note that the viewer in Figure 3-12, which contains an object, has an Invoke Method button ; however the viewers for the ints in Figures 3-13 and 3-14 do not since primitives have no methods associated with them.

3.11 Invoking a Method

To invoke a method on an object in a viewer (see Figure 3-12), click the Invoke Method button . To invoke a method for an object on the workbench, select the object, right click, and then select **Invoke Method**. In Figure 3-15, `fiction_2` has been selected, followed by a right mouse click, and then Invoke Method has been selected. A list of visible user methods will be displayed in a dialog box as shown in Figure 3-16. You can also display all visible methods by selecting the appropriate option. After one of the methods is selected and the parameters filled in as necessary, click **Invoke**. This will execute the method and display the return value (or void) in a dialog, as well as display any output in the usual way. If the method updates a field (e.g., `setMainCharacter()`), the effect of the invocation is seen in appropriate object field in the Workbench tab. The “stick-pin” located in the upper left of the dialog can be used to make the Invoke Method dialog remain open. This is useful when invoking multiple methods for the same object. The Show Doc button will be enabled if documentation has

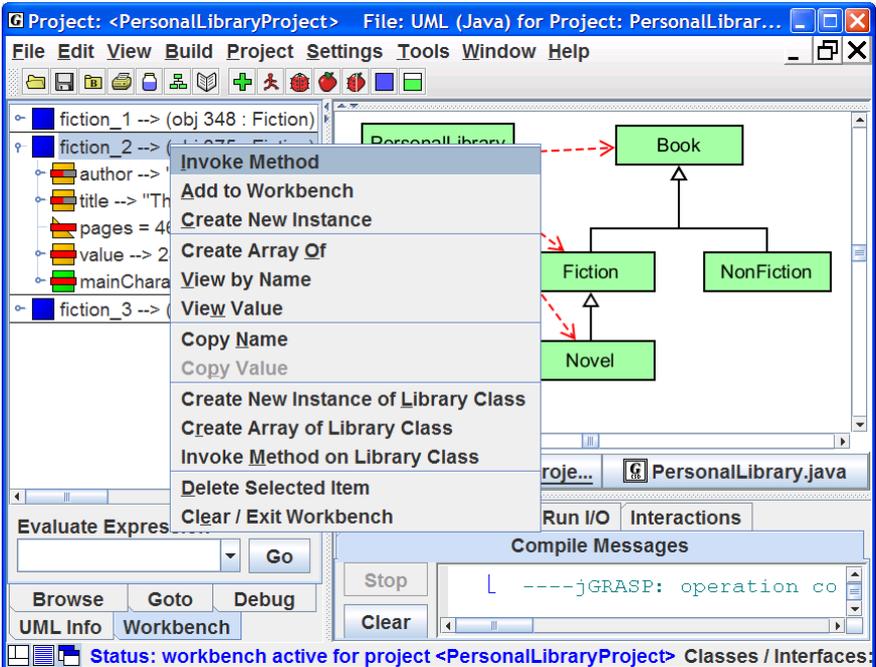


Figure 3-15. Workbench with two instances of Fiction

been generated for the project.

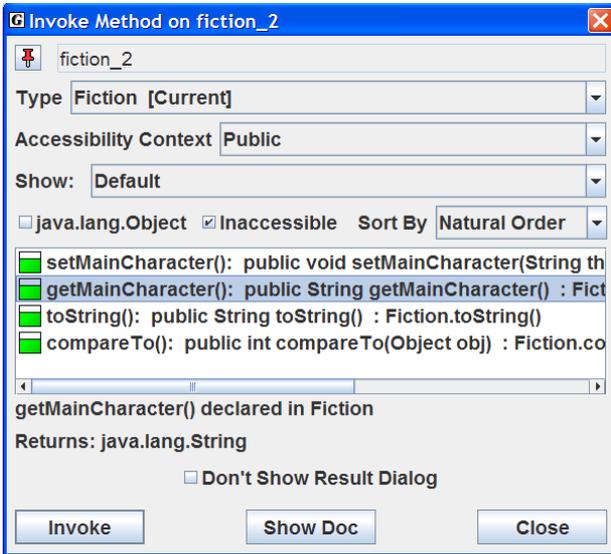


Figure 3-16. Selecting a method

As indicated above, perhaps one of the most compelling reasons for using the workbench approach is that it allows the user to create an object and invoke each of its methods in isolation. Thus, with an instance of `Fiction` on the workbench, each of its four methods: `getMainCharacter()`, `setMainCharacter()`, `toString()`, and `compareTo()` can be invoked directly. By carefully reviewing the results of the method invocations, we can informally test the class without the need for a driver with a *main()* method.

3.12 Invoking Methods with Parameters That Are Objects

In the example above, we created three instances of `Fiction`. Instances of any class in the UML diagram can be created and placed on the workbench. If the constructor requires parameters that are primitive types and/or strings, these can be entered directly, with any strings enclosed in double quotes. However, if a parameter requires an object, then you must create an object instance on the workbench first. Then you can simply drag the object from the workbench to the parameter field in the Invoke Method dialog. You can also use the *new* operator to create an instance when entering the value of a parameter.

3.13 Invoking Methods on Object Fields

If you have an object in the Workbench tab pane, you can expand it to reveal its fields. Recall, in Figure 3-11, `fiction_2` had been expanded to show its fields (`mainCharacter`, `author`, `title`, `pages`, and `mainCharacter`). Since the field `mainCharacter` is itself an object of the `String` class, you can invoke any of the `String` methods. For example, right-click on `mainCharacter` and select **Invoke Method**. When the dialog pops up (Figure 3-17), scroll down and select the first `toUpperCase()` method and click **Invoke**. This should pop up the Result dialog with “ROBERT LANGDON” as the return value (Figure 3-18). This method call has no effect on the value of the field for which it was called; it simply returns the string value converted to uppercase.

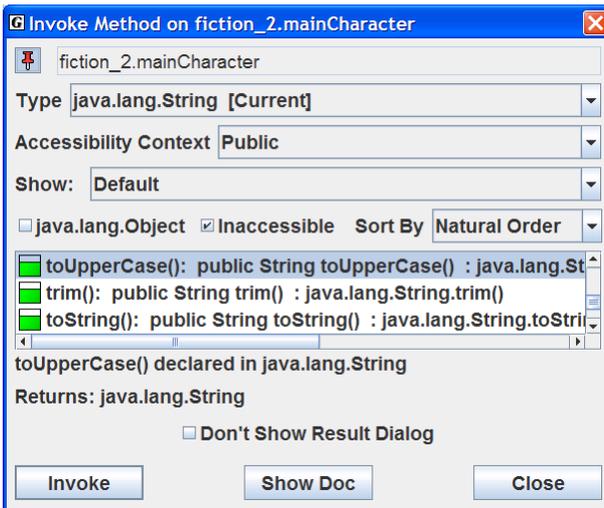


Figure 3-17. Invoking a `toUpperCase()` method on `fiction_2.mainCharacter`

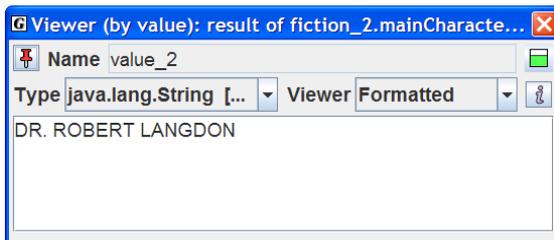


Figure 3-18. Result of `fiction_2.mainCharacter.toUpperCase()`

3.14 Showing Categories of Methods

The methods shown in the Invoke Method dialog based on the category selected in the “**Show:**” field. The “**Show: Default**” category includes the methods declared in the object’s class and all of its superclasses except the Object class. A number of other useful categories are also available in the dialog. For example, Figure 3-19 shows the “**Declared in java.lang.Object**” category selected for `fiction_2`. These are the methods that `fiction_2` inherited from the Object class. The orange color coding of the method symbols indicates “inherited” methods. Notice that a `toString()` method was declared in the Object class and that it has gray bar on the orange method symbol indicating that the method is not visible. Since `Fiction` has its own `toString()` method, it is overriding the inherited method. If you invoke the one declared in Object, the rules of Java are such that the one declared in `Fiction` is actually executed. However, jGRASP allows you to invoke Object’s version by turning on (check box) **Invoke Non-virtual**. To view categories of methods, click the **Show** drop-down list on the dialog as indicated below.

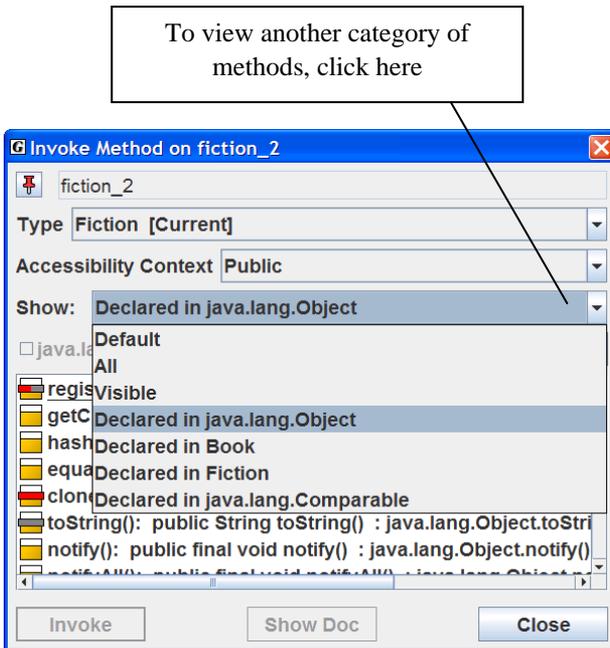


Figure 3-19. Showing methods declared in java.lang.Object

3.15 Creating Objects from the CSD Window

In addition to creating instances of classes from the UML class diagram, instances can be created directly from the CSD window after the class has been compiled. Figure 3-20 shows a CSD window containing class Fiction. From the menu, select **Build > Java Workbench > Create New Instance**. Buttons are also available on the toolbar for Create New Instance  and Invoke Static Method  (remember that only static methods can be invoked from a class). You can always create instances from the CSD window even if you have not created a project and UML diagram. This makes it convenient to quickly create an instance for the workbench and then invoke its methods.

Click  to create an instance of the class in the CSD window.

Click  to invoke a static method. Note that Fiction has no static methods; try this with PersonalLibrary and you should see *main* in the list).

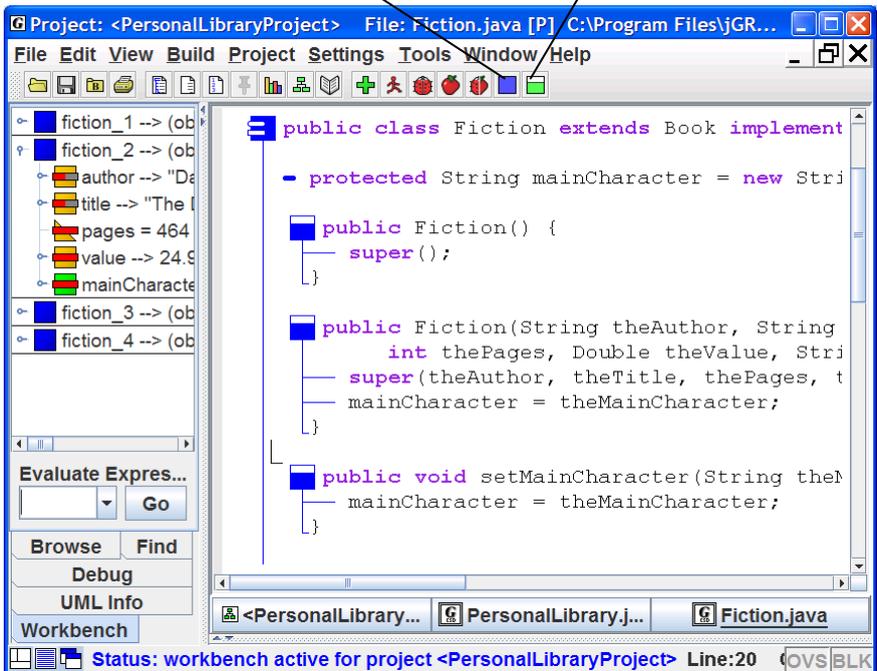


Figure 3-20. Creating an Instance from the CSD Window

3.16 Using Interactions

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 provide a convenient interface for working with items in the workbench or debug tabs. In fact, when you enter code that creates an object or primitive, the item is placed on the workbench where it can be inspected by unfolding and/or opening a viewer on it. Interactions can be especially helpful when learning and experimenting with objects and other elements in the Java language.

Consider Figure 3-21 where the context for Interactions is the UML window for the PersonalLibraryProject. Typing the following statement and pressing ENTER creates an instance of Novel on the workbench.

```
Novel n = new Novel();
```

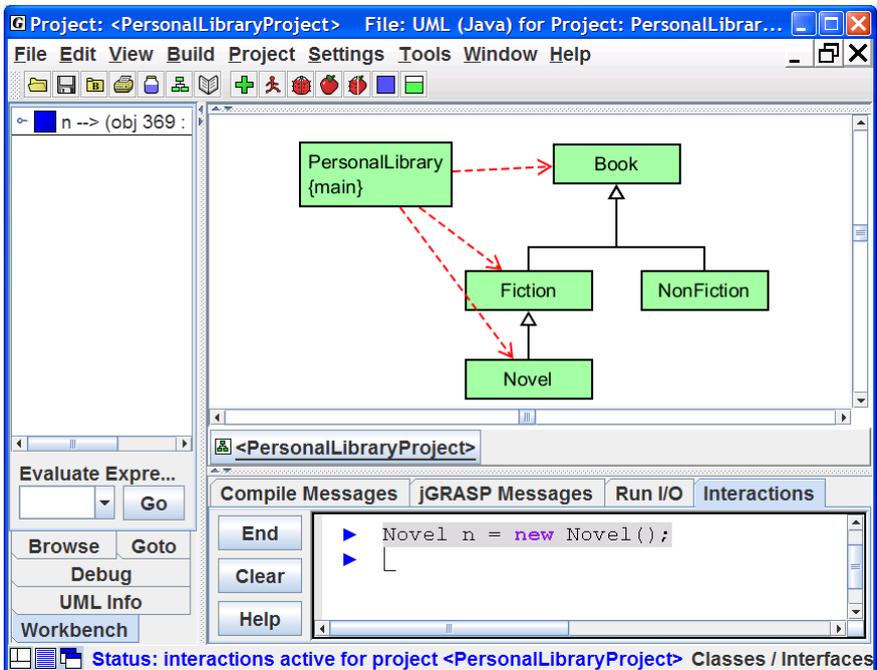


Figure 3-21. Using Interactions

With `n` on the workbench, we can now type statements or expressions that reference `n` and have them executed or evaluated immediately when ENTER is pressed. For example, typing `n` (followed by ENTER) is an expression that evaluates to the value of `n`, which is the Novel that was just created. For object

values, the result of invoking `toString()` on the object is displayed, as shown in Figure 3-22.

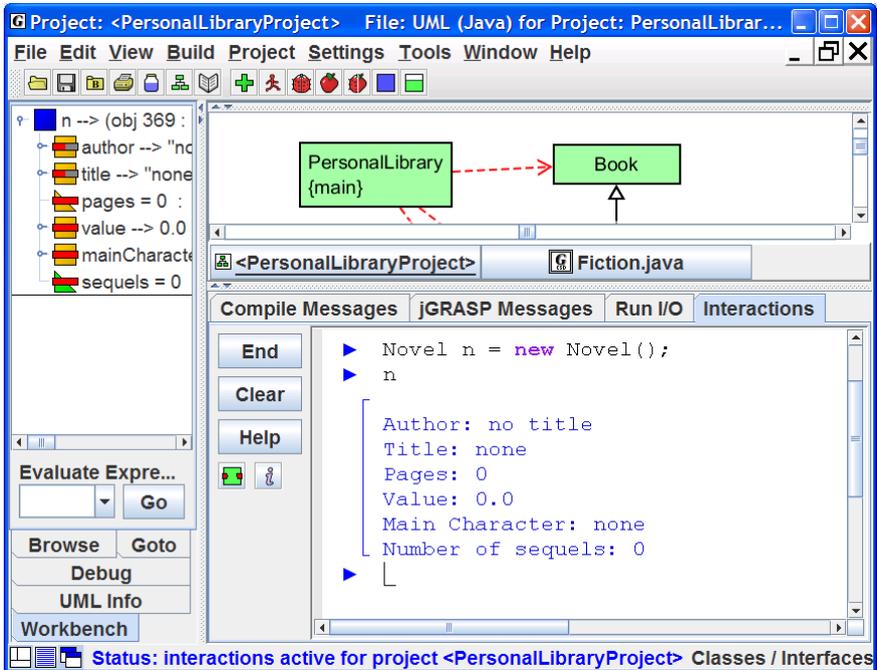


Figure 3-22. Entering and evaluating the expression `n`

When working with Interactions, mistakes will generate messages similar to those from the compiler. To correct a statement without retyping it, use the UP and DOWN arrow keys to scroll through the previous statements (history) one by one until you find it. Then use the LEFT and RIGHT arrow keys or mouse to move around within the statement in order to make the desired changes. Finally, press ENTER to execute the statement again.

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

```
( "The current value of n:" + n );
```

ENTER

If you simply press the ENTER at the end of the after the first line, Interactions will attempt to execute the incomplete statement and you get an error message.

Interactions in jGRASP can be a very useful tool, especially when learning new features, and you are encouraged to experiment with it.

3.17 Running the Debugger on Invoked Methods

When objects are on the workbench, the workbench is actually running Java in debug mode to facilitate the workbench operations. Thus, if you open a class in the CSD window and set a breakpoint in a method and then invoke the method from the workbench, the CSD window will pop to the top when the breakpoint is reached. When this occurs, you can single step through the program, examining variables in the Debug tab or you can open a separate viewer for a particular variable as described above in Section 3-10. See the Tutorial entitled “The Integrated Debugger” for more details.

3.18 Creating an Instance from the Java Class Libraries

You can create an instance of any class that is available to your program, which includes the Java class libraries. Find the Workbench menu at the top of the UML window. Click **Workbench > Create New Instance of Class**. In the dialog that pops up (Figure 3-23), enter the name of a class such as `java.lang.String` or select a class from the drop-down list, and click OK. This should pop up a dialog containing the constructors for `String`. Select an appropriate constructor, enter the argument(s), and click Create. This places the instance of the class on the workbench where you can invoke any of its methods as described earlier.



Figure 3-23.
Creating an instance of `String`

3.19 Exiting the Workbench

The workbench is *running* whenever you have objects on it or if you have invoked `main()` directly from the class diagram. If you attempt to do an operation that conflicts with workbench, such as compiling a class, jGRASP will prompt you with a

message indicating that the workbench is active and ask you if it is OK to end the Workbench (see Figure 3-24). The prompt is to let you know that the operation you are about to perform will clear the workbench. You can also clear or exit the workbench by right-clicking in the Workbench tab pane and selecting **Clear/Exit Workbench**.

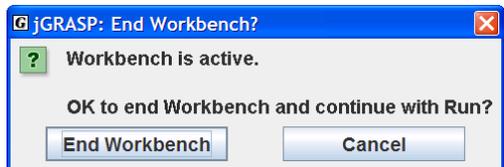


Figure 3-24. Making sure it is okay to exit the Workbench

3.20 Closing a Project

If you leave one or more projects open when you exit jGRASP, they will be opened again when you restart jGRASP. You should close any projects you are not using to reduce clutter in the Open Projects section of the Browse tab.

Here are two ways to close a project:

- (1) From the Desktop menu – Click **Project > Close** or **Close All Projects**.
- (2) In the Open Projects section of the **Browse** tab – Right-click on the project name and select **Close** or **Close All Projects**.

All project information is saved when you close the project as well as when you exit jGRASP.

3.21 Exiting jGRASP

When you have completed your session with jGRASP, you should “exit” (or close) jGRASP rather than leaving it open for Windows to close when you log out or shut down your computer. When you exit jGRASP, it saves its current state and closes all open files. If a file was edited during the session, it 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.

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**.

When you try to exit jGRASP while a process such as the workbench is still running, you will be prompted (Figure 3-25) to make sure it is okay to quit jGRASP.



Figure 3-25. Making sure it is okay to exit jGRASP

3.22 Review of Toolbar Buttons

Figure 3-26 provides a review of the buttons on the jGRASP toolbar. If you forget the function of a button, simply move the mouse over it to display the tool hint.

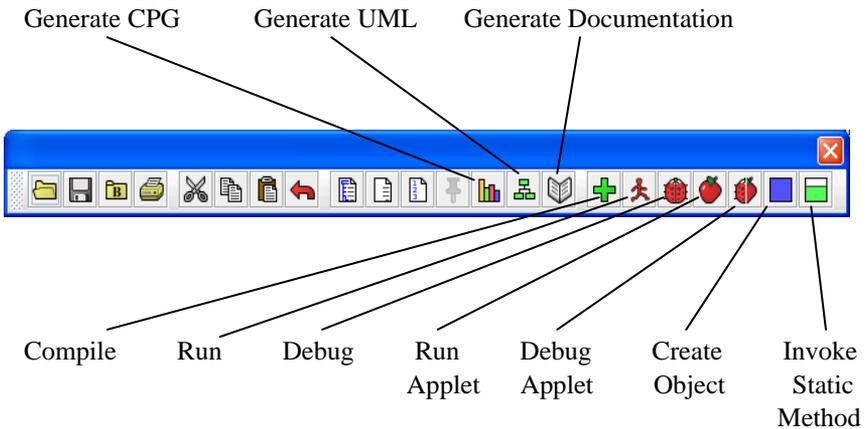
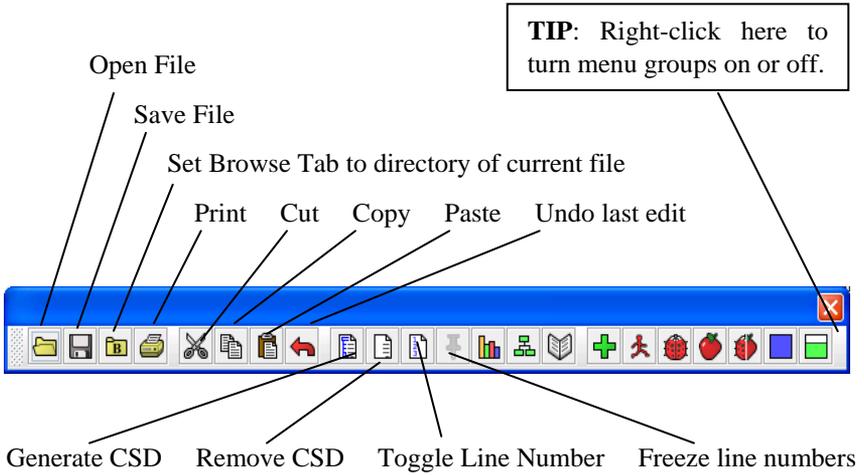


Figure 3-26. Toolbar

3.23 Exercises

- (1) Create a new project (**Project > New**) named PersonalLibraryProject2 in the same directory folder as the original PersonalLibraryProject. During the create step, add the file Book.java to the new project.
 - a. After the new project is created, add the other Java files in the directory to the project. Do this by dragging each file from the Files section of the Browse tab and dropping it in PersonalLibraryProject2 in the open projects section.
 - a. Remove a file from PersonalLibraryProject2. After verifying the file was removed, add it back to the project.
- (2) Generate the documentation  for PersonalLibraryProject2, using the default name for the documentation folder. After the Documentation Viewer pops up:
 - a. Click the Fiction class link in the API (left side).
 - b. Click the Methods link to view the methods for the Fiction class.
 - c. Visit the other classes in the documentation for the project.
- (3) Close the project.
- (4) Open the project by double-clicking on the project file in the files section of the Browse tab.
- (5) Generate the UML class diagram for the project.
 - a. Display the class information for each class.
 - b. Display the dependency information between two classes by selecting the appropriate arrow.
 - c. Compile  and run  the program using the buttons on the toolbar.
 - d. Invoke main() directly from the class diagram.
 - e. Create three instances of Fiction from the class diagram. Open Novel in a CSD window, then create two instances of Novel from the CSD window
 - f. Invoke some of the methods for one or more of these instances.
 - g. Open an object viewer for one or more String fields of one of the instances.
- (6) Use Interactions to enter statements and expressions that reference items on the workbench. Create new objects by entering statements such as:

```
Novel myNovel = new Novel();
```

- (7) Open the CSD window for PersonalLibrary.java.
 - a. Set a breakpoint on the first executable statement.
 - b. From the UML window, start the debugger by clicking the Debug button.
 - c. Step through the program, watching the objects appear in the Debug tab as they are created.
 - d. Restart the debugger. This time click “step in” instead of “step”. This should take you into the constructors, etc.
- (8) If you have other Java programs available, repeat the steps above for each program.

Notes