

jGRASP

Tutorials

Tutorials* for the jGRASP™ 1.8.4 Integrated Development Environment

James H. Cross II and Larry A. Barowski

Copyright © 2006 Auburn University

All Rights Reserved

October 15, 2006

DRAFT

***These tutorials are from the jGRASP Handbook.**

Copyright © 2006 Auburn University

All Rights Reserved

Table of Contents

Overview of jGRASP and the Tutorials	1
1 Installing jGRASP	4
2 Getting Started	7
2.1 Starting jGRASP	8
2.2 Quick Start - Opening a Program, Compiling, and Running	9
2.3 Creating a New File	11
2.4 Saving a File	14
2.5 Generating a Control Structure Diagram	15
2.6 Folding a CSD	17
2.7 Line Numbers	18
2.8 Compiling a Program – A Few More Details	18
2.9 Running a Program - Additional Options	22
2.10 Using the Debugger	24
2.11 Opening a File – Additional Options	26
2.12 Closing a File	28
2.13 Exiting jGRASP	29
2.14 Exercises	29
2.15 Review and Preview of What’s Ahead	30
3 Getting Started with Objects	31
3.1 Starting jGRASP	32
3.2 Navigating to Our First Example Project	33
3.3 Opening a Project and UML Window	34
3.4 Compiling and Running the Program from UML Window	35
3.5 Exploring the UML Window	36
3.6 Viewing the Source Code in the CSD Window	37
3.7 Exploring the Features of the UML and CSD Windows	38
3.7.1 Viewing the source code for a class	38
3.7.2 Displaying class information	38
3.7.3 Displaying Dependency Information	38

3.8 Generating Documentation for the Project	39
3.9 Using the Object Workbench.....	40
3.10 Opening a Viewer Window	43
3.11 Invoking a Method.....	44
3.12 Invoking Methods with Object Parameters.....	45
3.13 Invoking Methods on Object Fields.....	45
3.14 Invoking Inherited Methods.....	46
3.15 Running the Debugger on Invoked Methods	46
3.16 Creating Objects from the CSD Window	46
3.17 Creating an Instance from the Java Class Libraries.....	48
3.18 Exiting the Workbench	48
3.19 Closing a Project	48
3.20 Exiting jGRASP	49
3.21 Exercises	50
4 Projects.....	51
4.1 Creating a Project.....	52
4.2 Adding files to the Project	54
4.3 Removing files from the Project	55
4.4 Generating Documentation for the Project (Java only)	56
4.5 Jar File Creation and Extraction	58
4.6 Closing a Project	58
4.7 Exercises	58
5 UML Class Diagrams	59
5.1 Opening the Project.....	60
5.2 Generating the UML	61
5.3 Compiling and Running from the UML Window.....	62
5.4 Determining the Contents of the UML Class Diagram	63
5.5 Laying Out the UML Diagram	66
5.6 Displaying the Members of a Class	67
5.7 Displaying Dependencies Between Two Classes	68
5.8 Navigating to Source Code via the Info Tab	69
5.9 Finding a Class in the UML Diagram.....	69
5.10 Opening Source Code from UML.....	69

5.11 Saving the UML Layout	70
5.12 Printing the UML Diagram	70
6 The Object Workbench	71
6.1 Invoking Static Methods from the CSD Window	72
6.2 Invoking Static Methods from the UML Window	74
6.3 Creating an Object for the Workbench	76
6.4 Invoking a Method	78
6.5 Invoking Methods with Parameters Which Are Objects	79
6.6 Invoking Methods on Object Fields	79
6.7 Selecting Categories of Methods to Invoke	80
6.8 Opening Object Viewers	82
6.9 Running the Debugger on Invoked Methods	83
6.10 Exiting the Workbench	83
7 The Integrated Debugger	84
7.1 Preparing to Run the Debugger	85
7.2 Setting a Breakpoint	85
7.3 Running a Program in Debug Mode	86
7.4 Stepping Through a Program – the Debug Buttons	88
7.5 Stepping Through a Program – without <i>Stepping In</i>	89
7.6 Stepping Through a Program – and <i>Stepping In</i>	91
7.7 Opening Object Viewers	92
7.8 Debugging a Program	94
8 The Control Structure Diagram (CSD)	95
8.1 An Example to Illustrate the CSD	96
8.2 CSD Program Components/Units	98
8.3 CSD Control Constructs	99
8.4 CSD Templates	103
8.5 Hints on Working with the CSD	104
8.6 Reading Source Code with the CSD	105
8.7 References	110
9 Viewers for Data Structures	111
9.1 Opening Viewers	112
9.2 Setting the View Options	114

9.3 Selecting Among Views	115
9.4 Presentation Views for LinkedList, HashMap, and TreeMap	117
9.5 Presentation (Data Structure Identifier) View	121
9.5.1 LinkedListExample.java	122
9.5.2 BinaryTreeExample.java	124
9.5.3 Configuring Views generated by the Data Structure Identifier	126
9.6 Summary of Views	128
9.7 Exercises	129

Overview of jGRASP and the Tutorials

jGRASP is a lightweight integrated development environment (IDE), created specifically to provide visualizations for improving the comprehensibility of software. jGRASP is implemented in Java, and thus, runs on all platforms with a Java Virtual Machine. As with the previous versions, jGRASP supports Java, C, C++, Ada, and VHDL, and it comes configured to work with several popular compilers to provide “point and click” compile and run. jGRASP, which is based on its predecessors, pcGRASP and UNIX GRASP (written in C/C++), is the latest IDE from the **GRASP** (Graphical Representations of Algorithms, Structures, and Processes) research group at Auburn University.

jGRASP currently provides for the automatic generation of three important software visualizations: (1) *Control Structure Diagrams* (Java, C, C++, Ada, and VHDL) for source code visualization, (2) *UML Class Diagrams* (Java) for architectural visualization, and (3) *Viewers* (Java) which provide dynamic views for primitives and objects including traditional data structures such as linked lists and binary trees. jGRASP also provides an innovative *Object Workbench* and *Debugger* which are tightly integrated with these visualizations. Each is briefly described below.

The **Control Structure Diagram (CSD)** is an algorithmic level diagram generated for Ada, C, C++, Java and VHDL. The CSD is intended to improve the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit. The CSD, designed to fit into the space that is normally taken by indentation in source code, is an alternative to flow charts and other graphical representations of algorithms. The goal was to create an intuitive and compact graphical notation that is easy to use. The CSD is a natural extension to architectural diagrams such as UML class diagrams.

The CSD window in jGRASP provides complete support for the CSD generation as well as editing, compiling, running, and debugging programs. After editing the source code, regenerating a CSD is fast, efficient, and non-disruptive (approximately 5000 lines/sec). The source code can be folded based on CSD structure (e.g., methods, loops, if statements, etc.), then unfolded level-by-level. Standard features for program editors such as syntax based coloring, cut, copy, paste, and find-and-replace are also provided.

The **UML Class Diagram** is currently generated for Java source code from all Java class files and jar files in the current project. Dependencies among the classes are depicted with arrows (edges) in the diagram. By selecting a class, its members can be displayed, and by selecting an arrow between two classes, the actual dependencies can be displayed. This diagram is a powerful tool for understanding a major element of object-oriented software - the dependencies among classes.

The **Viewers** for objects and primitives provide dynamic visualizations as the user steps through a program in debug mode or invokes methods for an object on the workbench. Textbook-like *Presentation* views are available for instances of classes that represent traditional data structures. A *data structure identifier* is used to determine if an object is a linked list or binary tree. When a positive identification is made, an appropriate

presentation view of the object is displayed. The *data structure identifier* is intended to work for user classes as well as the most commonly used classes in the Java Collections Framework (e.g., ArrayList, LinkedList, HashMap, and TreeMap). A future *Viewer API* will allow users to create custom dynamic views of their own classes.

The **Object Workbench**, in conjunction with the UML class diagram and CSD window, allows the user to create instances of classes and invoke their methods. After an object is placed on the Workbench, the user can open a viewer to observe changes resulting from the methods that are invoked. The Workbench paradigm has proven to be extremely useful for teaching and learning object-oriented concepts, especially for beginning students.

The **Integrated Debugger** works in conjunction with the CSD window, UML window, and the Object Workbench. The Debugger provides a seamless way for users to examine their programs step by step. The execution threads, call stack, and variables are easily viewable during each step. The jGRASP debugger has been used extensively during lectures as a highly interactive medium for explaining programs.

The *jGRASP Tutorials* provide best results when read while using jGRASP; however, they are sufficiently detailed to be read in a stand-alone fashion by a user who has experience with one or more other IDEs. The tutorials are quite suitable as supplemental assignments during the course. When working with jGRASP and the tutorials, students can use their own source code, or they can use the examples shown in the tutorials (..\jGRASP\examples\Tutorials\). Users may want to copy the examples folder to their own directories prior to modifying them. The Tutorials are listed below along with suggestions for their use.

1. **Installing jGRASP** – This tutorial can be skipped if jGRASP and the Java JDK have already been installed successfully. It is recommended for those students planning to install jGRASP and the Java JDK on their personal machines.
2. **Getting Started** – This tutorial is a good starting place for those new to jGRASP. It introduces the process of creating and editing Java source files, then compiling and running programs. It also includes generating the CSD for the program.
3. **Getting Started with Objects** – This tutorial is a good starting place for those interested in an *Objects First* approach to learning Java, but it assumes the reader will refer to Section 2 as needed. Projects, UML class diagrams, the Object Workbench, and Viewers are introduced.
4. **Projects** – This tutorial discusses the concept of a project file (.gpj) in jGRASP which stores all information for a specific project. This includes the names (and paths) of each file in the project, the project settings, and the layout of the UML diagram. Some users may want to work in projects from the beginning while others want to deal with projects only when programs have multiple classes or files.
5. **The UML Class Diagram** – This tutorial assumes the user understands the concept of a project and is able to create a one (Tutorial 4).
6. **The Object Workbench** – This tutorial assumes the user is able to create a project (Tutorial 4) and work with UML class diagrams (Tutorial 5). The workbench provides an exciting way to approach object-oriented concepts and programming by

allowing the user to create objects and invoke the methods directly rather than indirectly via a `main()` method.

7. ***The Integrated Debugger*** – This tutorial can be done anytime. Students should be encouraged to begin using the debugger early on so that they can step through their programs, even if only to observe variables as they change.
8. ***The Control Structure Diagram*** – This tutorial is perhaps best read as control structures such as the *if*, *if-else*, *switch*, *while*, *for*, and *do* statements are studied. However, for those already familiar with the common control structures of programming languages, the tutorial can be read anytime. The latter part contains some helpful hints on getting the most out of the CSD.
9. ***Viewers for Data Structures*** – This tutorial provides a more in-depth introduction to using Viewers with linked lists, binary trees, and other traditional data structures. Examples of *presentation* views are included for instances of `ArrayList`, `LinkedList`, `HashMap`, and `TreeMap`, as well non-JDK implementations for a linked list and binary tree.

For additional information and to download jGRASP, please visit our web site at the following URL. <http://www.jgrasp.org>

New in jGRASP 1.8

Perhaps the most notable change in version 1.8 involves the “look and feel” of the user interface. In addition, CSD generation can now be undone, and new viewers are now available for the commonly used classes in the Java Collections Framework.

Single vs. Multiple Menus and Toolbars – The default View of the desktop is now *single menu* and *single toolbar*. Previous versions of jGRASP provided a top level menu for the desktop and a top level menu for each open CSD or UML window. In single menu/toolbar mode, the CSD or UML window that has focus determines the content and functionality of the menu and toolbar. Users who prefer the pre-1.8 look-and-feel of *multiple menus* and *multiple toolbars* can select these options via the View menu.

CSD Undo – Beginning with version 1.8, CSD generation is treated like an edit, which means it can be undone like any other edit (Edit – Undo, or Ctrl-Z). Note that undoing a *Generate CSD* is different from a *Remove CSD* operation. The Undo returns the file to the prior state, whereas the Remove CSD removes the diagram but leaves changes in indentation and/or line feeds resulting from the previous Generate CSD.

Viewers for Objects and Primitives – New viewers for primitives, array, `ArrayList`, `LinkedList`, `HashMap`, `TreeMap`, and non-JDK linked lists and binary trees have been added to jGRASP. Each viewer provides multiple views of an object. For example, the new array viewer includes *basic* view, *array elements* view, and *presentation* view (or “textbook view”), as well as a special *two-dimensional array elements* view. A new *data structure identifier* provides *presentation* views for both JDK and non-JDK classes representing traditional data structures.

1 Installing jGRASP

The current version of jGRASP is available from <http://www.jgrasp.org> in four separate files: two are self-extracting for **Microsoft Windows**, one is for **Mac OS X**, and the fourth is a generic **ZIP** file. Although the generic ZIP file can be used to install jGRASP on any system, it is primarily intended for **Linux** and **UNIX** systems. If you are on a Windows machine, either “jGRASP exe” or “jGRASP JRE exe” is recommended.

jGRASP exe (2.8 MB) – Windows self-extracting exe file. The full Java 2 Platform Standard Edition (J2SE) Development Kit (hereafter referred to as JDK) must be installed to run jGRASP and compile and run Java programs.

jGRASP JRE exe (15.6 MB) – Windows self-extracting exe file with Java Runtime Environment (JRE). This version includes a copy of the JRE so that no Java installation is required to run jGRASP itself; *however, the JRE does not include the Java compiler. If you will be compiling and running Java programs, you must also install the full JDK.* The *jGRASP JRE* version of jGRASP is convenient if you will be compiling programs in languages other than Java.

jGRASP pkg.tar.gz (3.1 MB) – Mac OS X tarred and gzipped package file (requires admin access to install). J2SDK is pre-installed on Mac OS X machines.

jGRASP (3.4 MB) – Zip file. After unzipping the file, refer to README file for installation instructions. *The full JDK must be installed in order to run jGRASP and to compile and run Java programs.*

Installing on Windows 95/98/2000/XP – After downloading either “jGRASP exe” or “jGRASP JRE exe” (described above), simply double click on the .exe file, and the script will take you through the steps for installing jGRASP. If you are uncertain about a step, you should accept the default by clicking Next and/or pressing ENTER key. When you have completed the installation, you should find the jGRASP icon on your desktop. jGRASP should also be listed on the Window’s Start – Programs menu.

Installing on Mac OS X – To install jGRASP on a Mac OS X machine, a root password is required. When you download jGRASP, the install file (.pkg.tar.gz) should unzip and untar automatically. If this does not happen, you can use Stuffit Expander [or from a terminal, use "gunzip jgrasp*.tar.gz" then "tar xf jgrasp*.tar"]. You should now be able to double click on the .pkg file to continue the installation. The first time you run jGRASP, the CSD font will be installed on your system, and a soft link to the jGRASP startup script (for command line execution) will be created in /usr/bin or your \$HOME/bin directory.

Currently, if you install Java 1.5 on Mac OS X, it is not automatically accepted as the default compiler. However, you can change the start up setting for jGRASP so that it will use Java 1.5 for itself and to compile and run your programs. Here are the steps to take after jGRASP 1.8.4 is installed:

- 1) Start up jGRASP in the usual way.

- 2) On the menu, select "Settings" - - "jGRASP Startup Settings" (near the bottom).
- 3) On the Startup Settings dialog, you should see "[Default]" selected as the Java executable for running jGRASP. Click the down-arrow to the right of "[Default]" and you see a list of the versions of Java installed on your machine. You may see both 1.5 and 1.5.0 in the list. Select either one of these, then click "OK".
- 4) Close and re-start jGRASP. Just ignore our warning message about using jGRASP with the new version of Java. In fact, click the option to not show it again.

If you were trying different compiler settings, etc. on your machine, you need to make sure the current environment in the compiler settings (Settings - - Compiler Settings - - Workspace) is set to:

jdk (integrated debugger, HotSpot VM, prefer JDK compiler) - generic

This is the default setting. You may need to stretch the Settings dialog to see the complete entry in the "Current Environment" window at the bottom of the dialog. If it is not set correctly, click the small square to the right of the entry. This will select the default setting above; a small black square inside the white square indicates the default has been selected. Click "Apply" and "Okay".

This setting tells jGRASP to "prefer" the java compiler from the same version of Java used to run jGRASP (i.e., the version you specified in the Startup Settings).

Installing on x86 Linux, SPARC Solaris, and NetBSD/i386 – Unzip the distribution (.zip) file in the directory where you wish to install jGRASP. This will create a jgrasp directory containing all the files. You may want to add the "bin" subdirectory of this directory to your execution path or create a soft link to .../jgrasp/bin/jgrasp from a directory on the executable path.

Compilers - Although jGRASP includes settings for a number of popular compilers, it does not include any compilers. Therefore, if the compiler you need is not already installed on your machine, it must be installed separately. Since these are generally rather large files, the download time may be quite long depending on your connection speed. If a compiler is available to you on a CD (e.g. with a textbook), you may save yourself time by installing it from the CD rather than attempting to download it.

Compiler Settings - jGRASP includes settings for the following languages/compilers. The default compiler settings are underlined. Note that links for those that can be freely downloaded are included for your convenience.

Ada (GNAT)

<ftp://cs.nyu.edu/pub/gnat/3.15p/winnt/gnat-3.15p-nt.exe>

C, C++ (GNU/Cygnus, Borland, Microsoft)

<http://sources.redhat.com/cygwin/>

http://www.borland.com/downloads/download_cbuilder.html

FORTTRAN (GNU/Cygnus)

Included with Cygwin, see (2) above. Note that FORTRAN is currently treated as Plain Text so there is no CSD generation.

Java (J2SE JDK, Jikes)

<http://java.sun.com/j2se/1.5.0/download.jsp>

Assembler (MASM)

Note that assembler is treated as Plain Text so there is no CSD generation.

After you have installed the compiler(s) of your choice, you will be ready to begin working with jGRASP. If you are not using the default compiler for a particular language (e.g., *JDK* for Java), you may need to change the Compiler Settings by clicking on **Settings – Compiler Settings – Workspace**. Select the appropriate language, and then select the environment setting that most nearly matches the compiler you have installed. Finally, click *Use* on the right side of the Settings dialog. For details see [Compiler Environment Settings](#) in **jGRASP Help**.

Starting jGRASP - You can start jGRASP by double clicking on the icon on your Windows desktop. See the next section for details.



jGRASP

2 Getting Started


Java will be used in the examples in this section; however, the information applies to all supported languages for which you have installed a compiler (e.g., Ada, C, C++, Java) unless noted otherwise. In any of the language specific steps below, simply select the appropriate language and source code. 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. If you have installed jGRASP on your own PC, you should see the jGRASP icon on the Windows desktop.

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 the Control Structure Diagram (CSD) window, including generating the CSD, folding your source code, numbering the lines, and stepping through the program in the integrated debugger.

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 Generating a Control Structure Diagram
- 2.6 Folding a CSD
- 2.7 Line Numbers
- 2.8 Compiling a Program – A Few More Details
- 2.9 Running a Program - Additional Options
- 2.10 Using the Debugger
- 2.11 Opening a File – Additional Options
- 2.12 Closing a File
- 2.13 Exiting jGRASP
- 2.14 Exercises
- 2.15 Review and Preview of What’s Ahead

2.1 Starting 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 in a computer lab and you don't see the jGRASP icon on the desktop, try the following: click **Start – Programs – 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 plus three resizable panes. The *left pane* has tabs for **Browse, Debug, Find, and Workbench** (Project tab is combined with the Browse tab beginning in version 1.7). The Browse tab, which is the default when jGRASP is started, lists the files in the current directory. The large *right pane* is for UML and CSD Windows. The *lower pane* has tabs for jGRASP messages, Compile messages, and Run Input/Output. The panes can be resized by moving the horizontal or vertical partitions that separate them. Select the partition with the mouse (left-click and hold down) then drag the partition to make a pane larger or smaller. You can also click the arrowheads on the partition to open and close the pane.

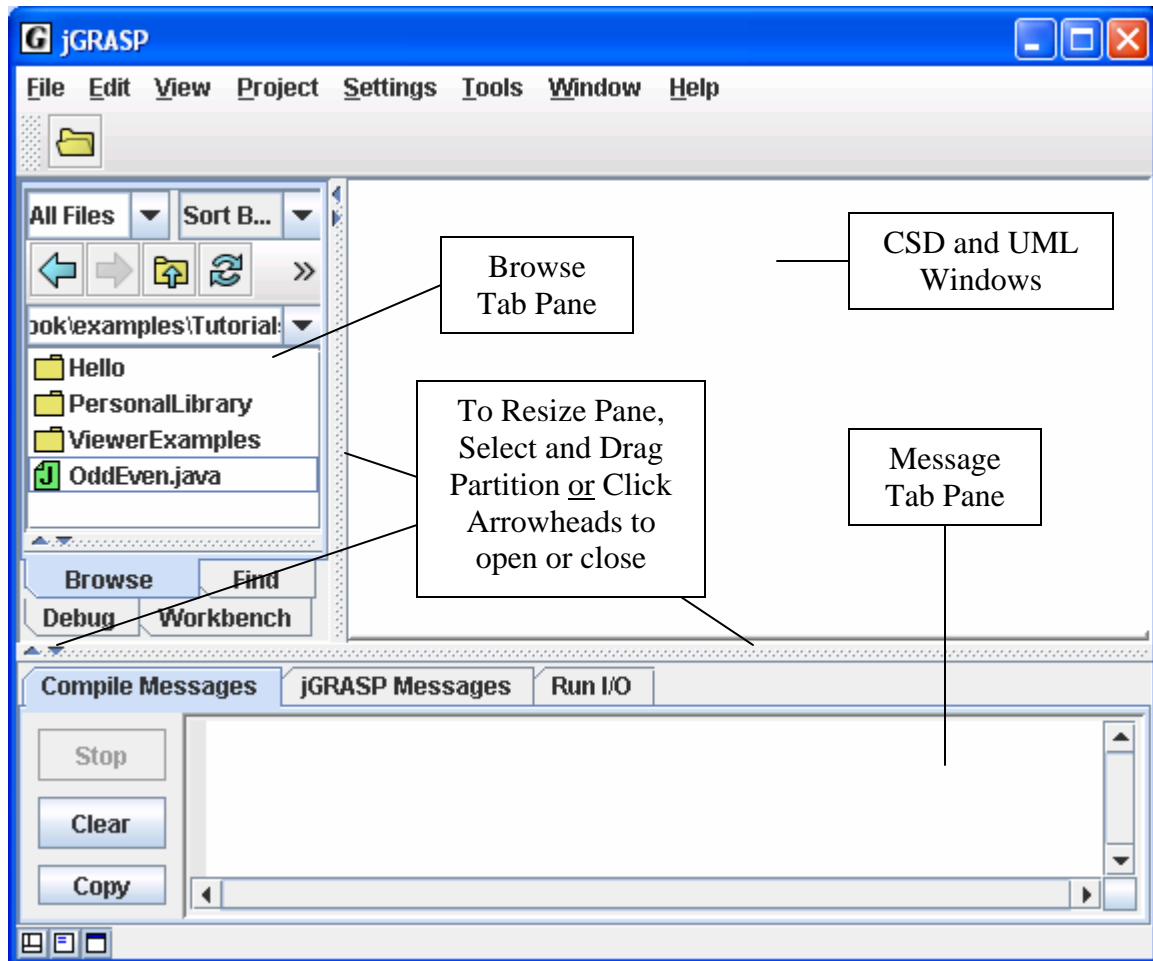





Figure 2-1. The jGRASP Virtual Desktop

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). If jGRASP was installed by a system administrator, you may not have write privileges for these files. If this is the case, you should copy the tutorial folder to one of your personal folders (e.g., in your *My Documents* 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 pane. Below, the Browse tab is displaying the contents of the Tutorials folder.

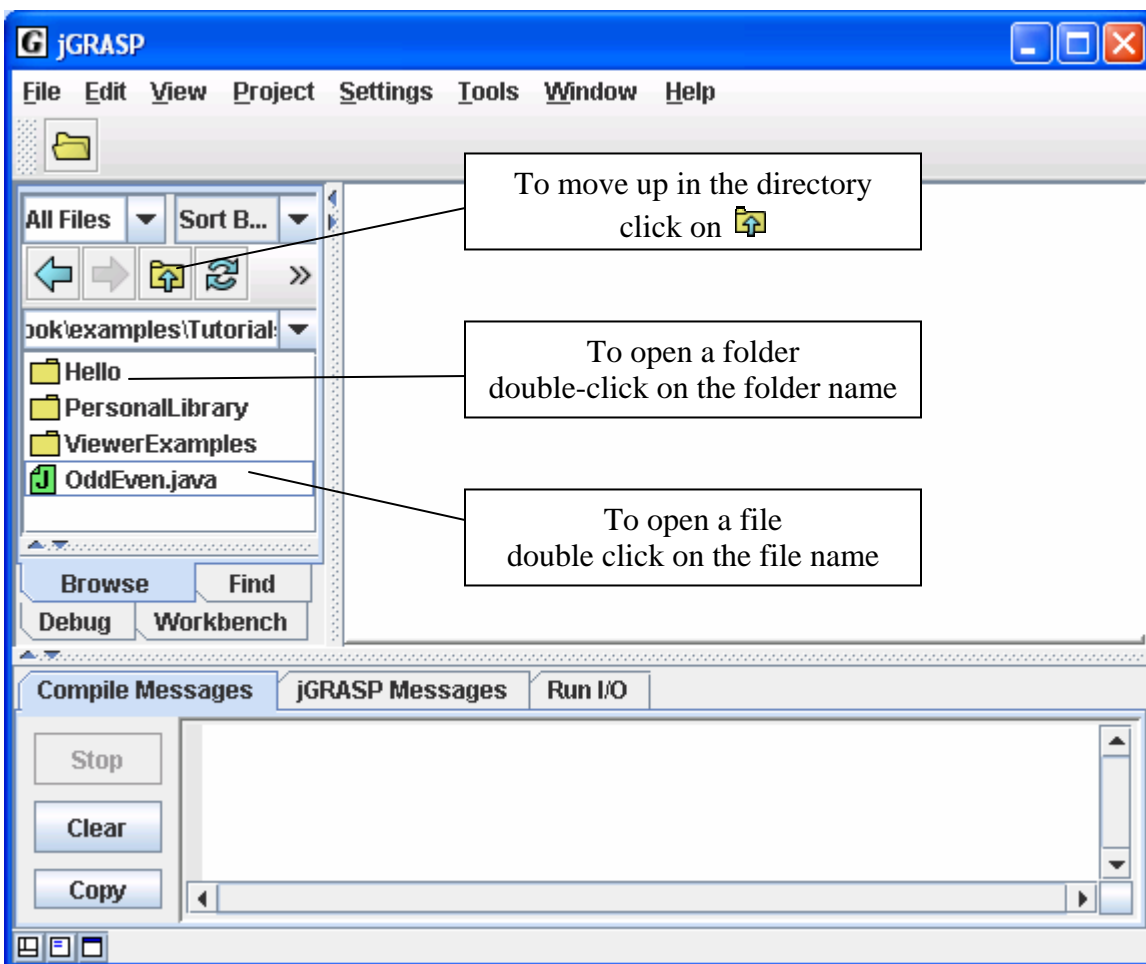


Figure 2-2. The jGRASP Virtual Desktop

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 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 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 of the Message pane.

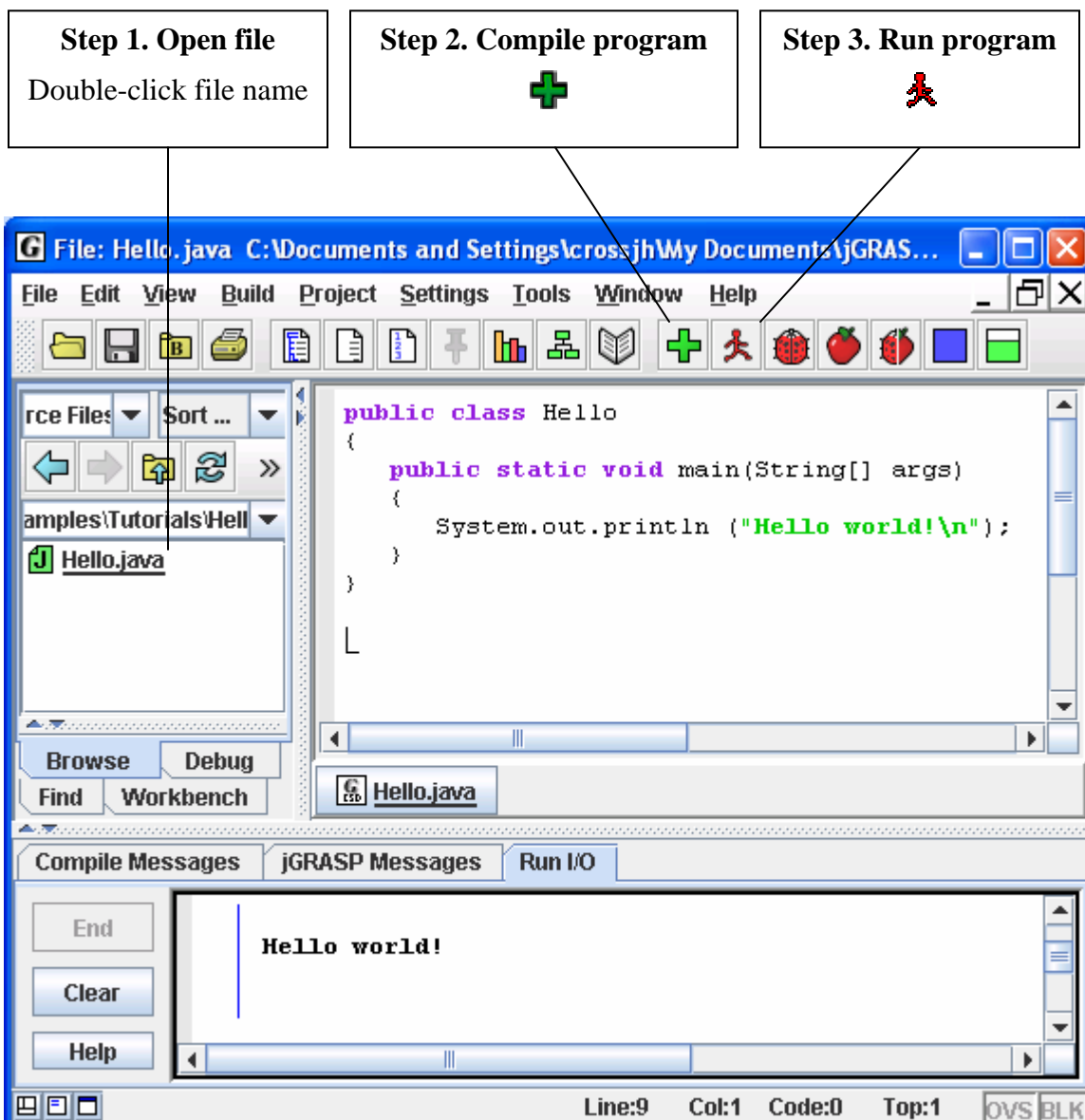


Figure 2-3. After loading file into CSD Window

2.3 Creating a New File

To create a new Java file within the Desktop, click on **File – New File – Java**. 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 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 the *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. Later when you have numerous windows open, you may want to reorder the buttons by dragging them around on the windowbar.



In the upper right corner of the CSD window are three buttons that control its

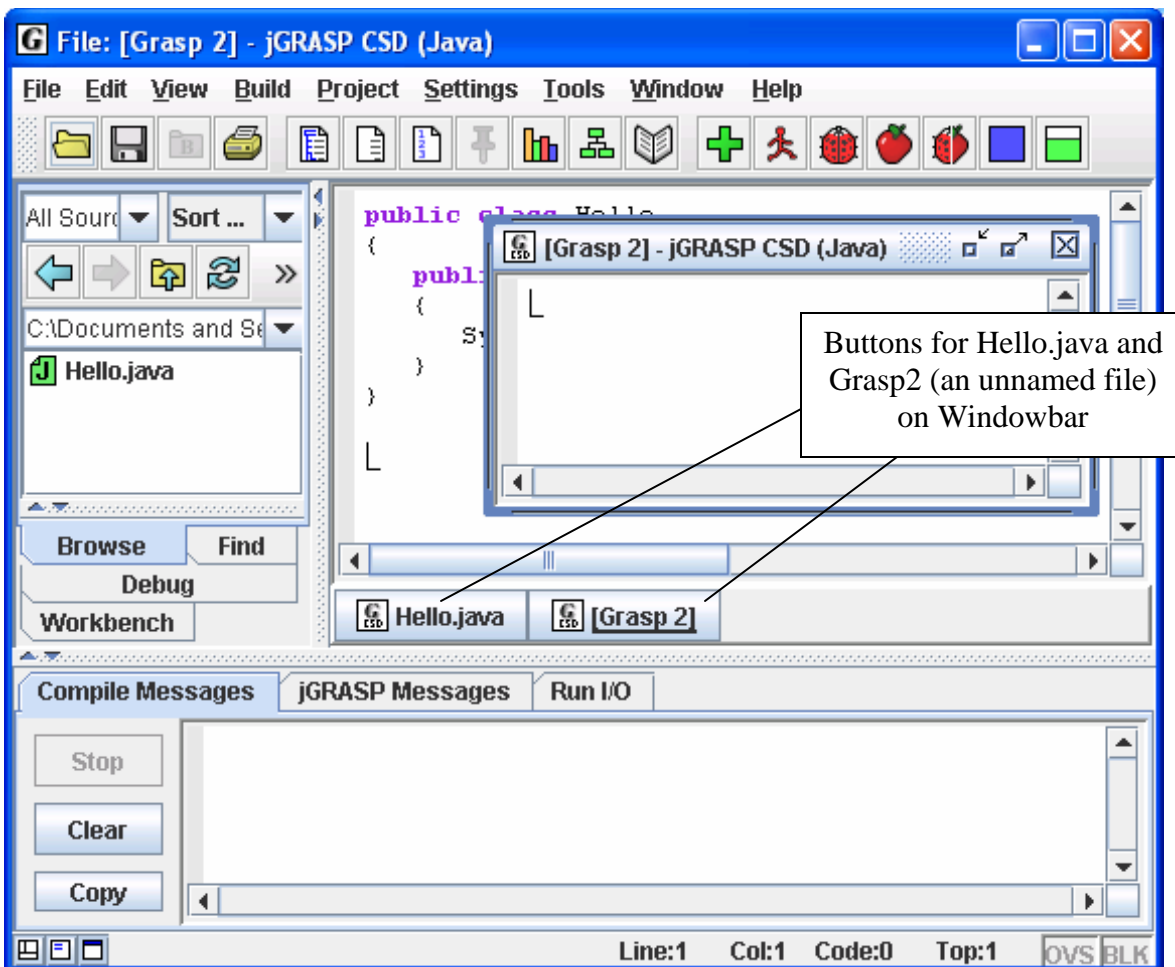


Figure 2-4. Opening a CSD Window for Java

display. The first button minimizes the CSD window; the second button maximizes the CSD window or, if it is already maximized, the button restores the CSD window to its previous size. 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 maximized within the virtual Desktop. The “L” shaped cursor in the upper left corner of the empty window indicates where text will be entered.

TIP: If you want all of your CSD windows to be maximized automatically when you open them, click **Settings – Desktop**, and then click **Open Desktop Windows Maximized** (note that a check mark indicates this option is turned ON).

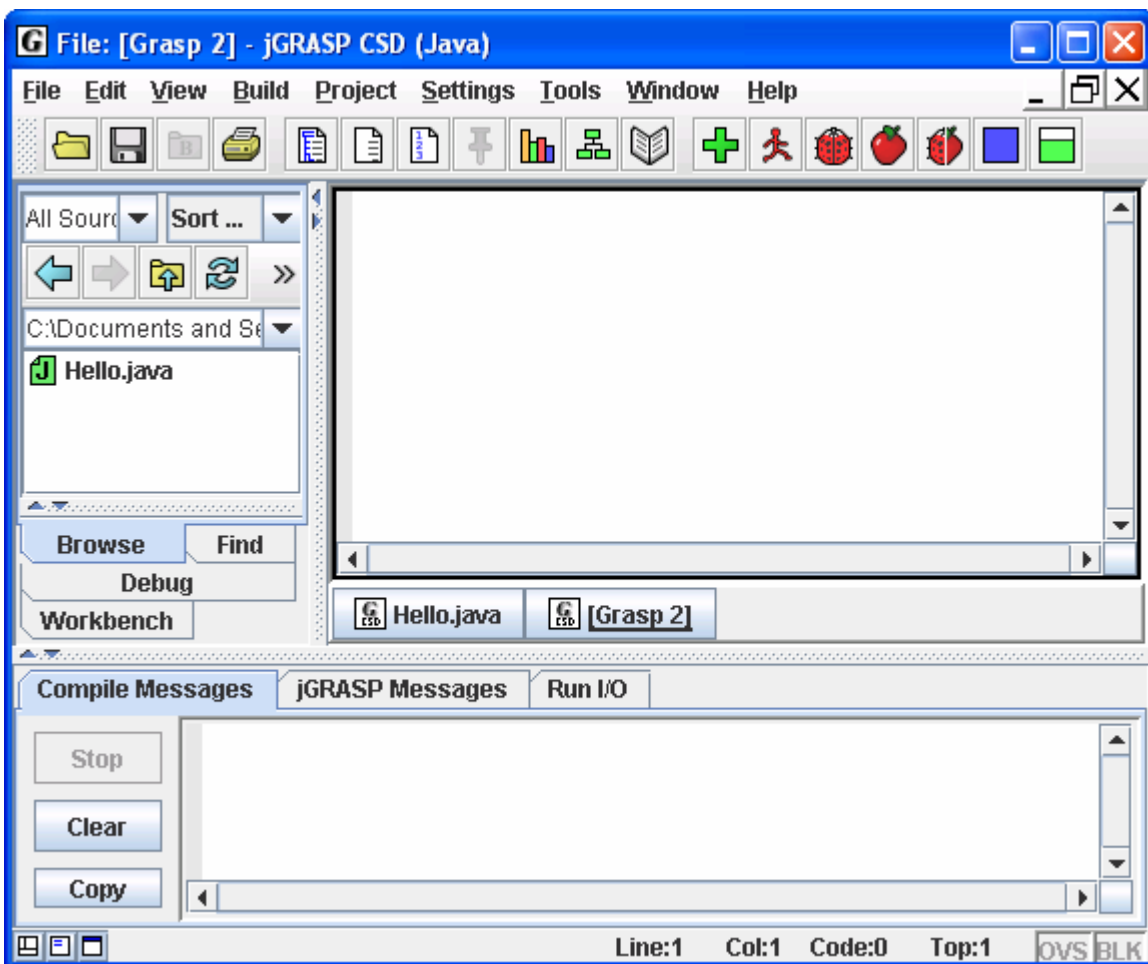


Figure 2-5. CSD Window maximized in Desktop

Type in 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.

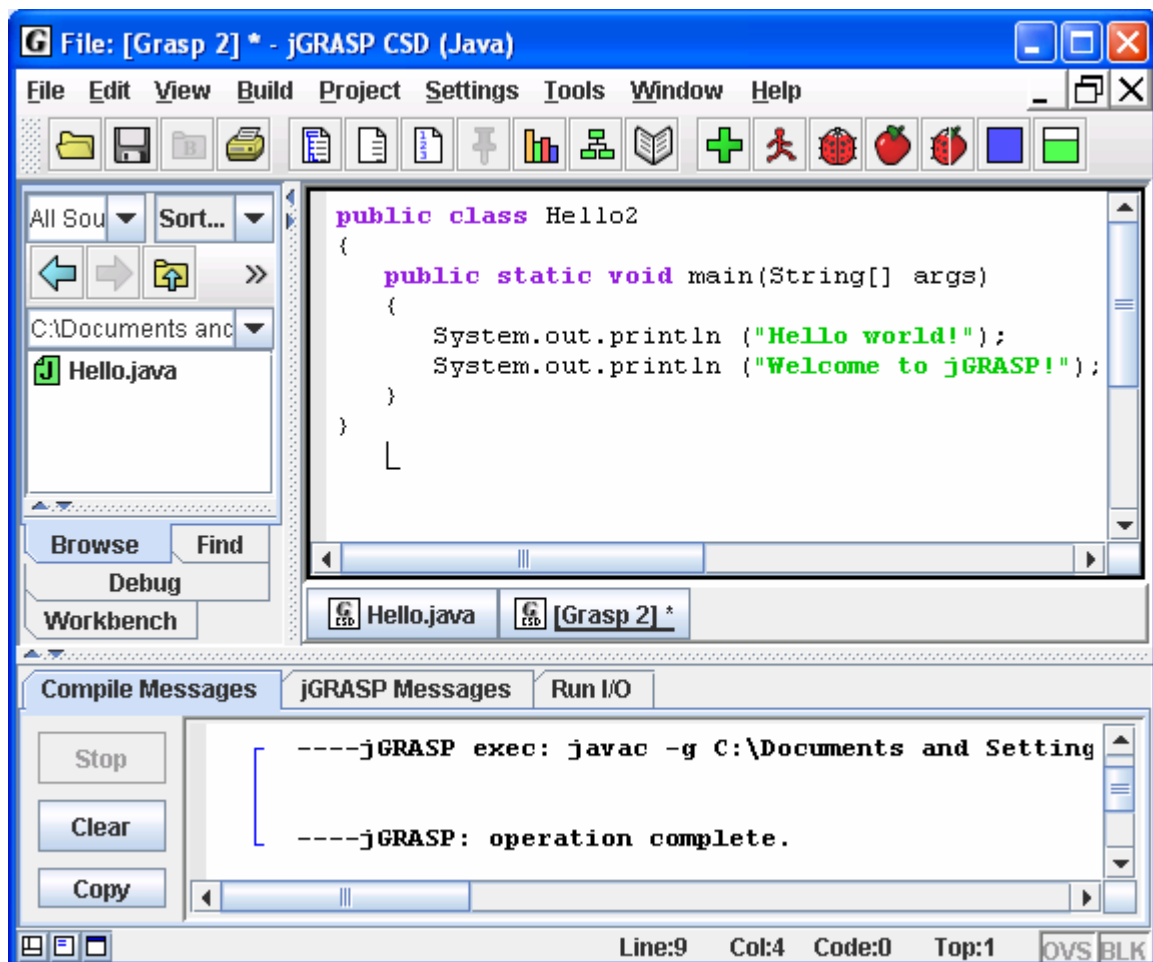




Figure 2-6. CSD Window with program entered

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 (see Figure 2-7), or
- (3) Pressing Ctrl-S (i.e., while pressing the Ctrl key, press the “s” key).

If the file has not been saved previously, the Save dialog box 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 2.8 on the next page). If you do not see the program in the Browse tab, you may need to navigate to the directory where the file was saved. **TIP:** Click  on the toolbar to change the Browse tab to the directory of the current file.

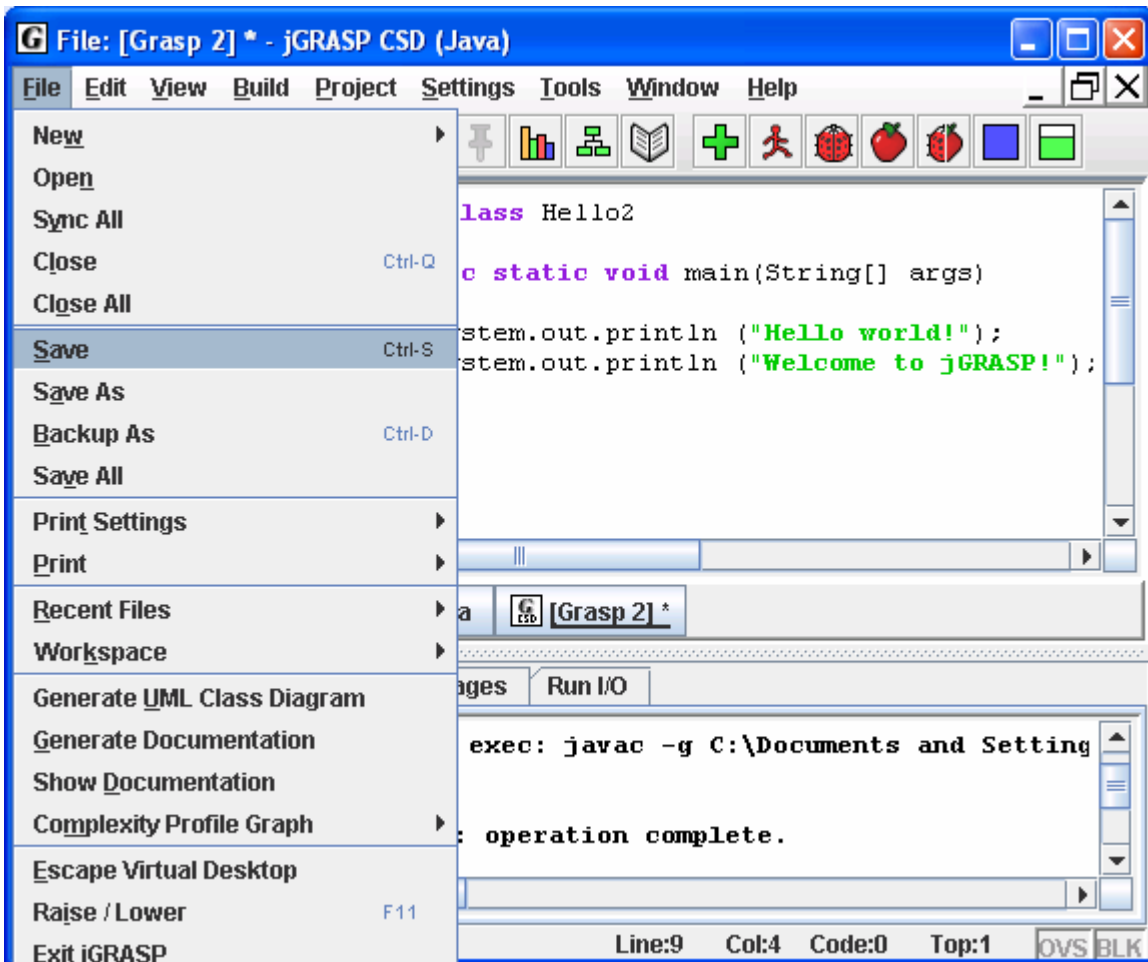



Figure 2-7. Saving a file from the CSD Window

2.5 Generating a Control Structure Diagram

You can generate a Control Structure Diagram in the CSD window whenever you have a syntactically correct program. Note that CSD generation does not do type checking so, even though the CSD may generate okay, 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 in the figure below. After you are able to successfully generate a CSD, go on to the next section below.

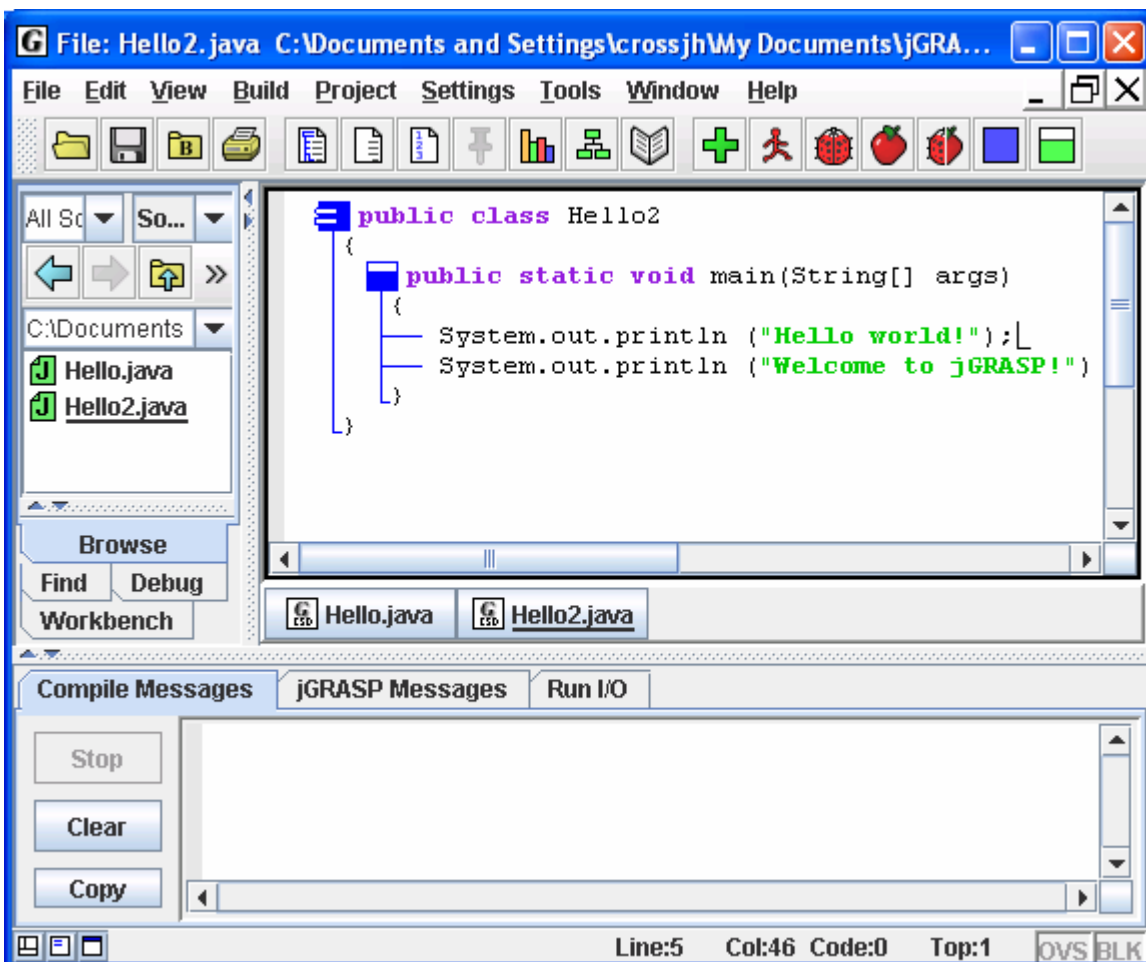



Figure 2-8. After CSD is generated

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 2-9, 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 usually provides a more detailed error message (see [Compiling a Program](#) below).

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.

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

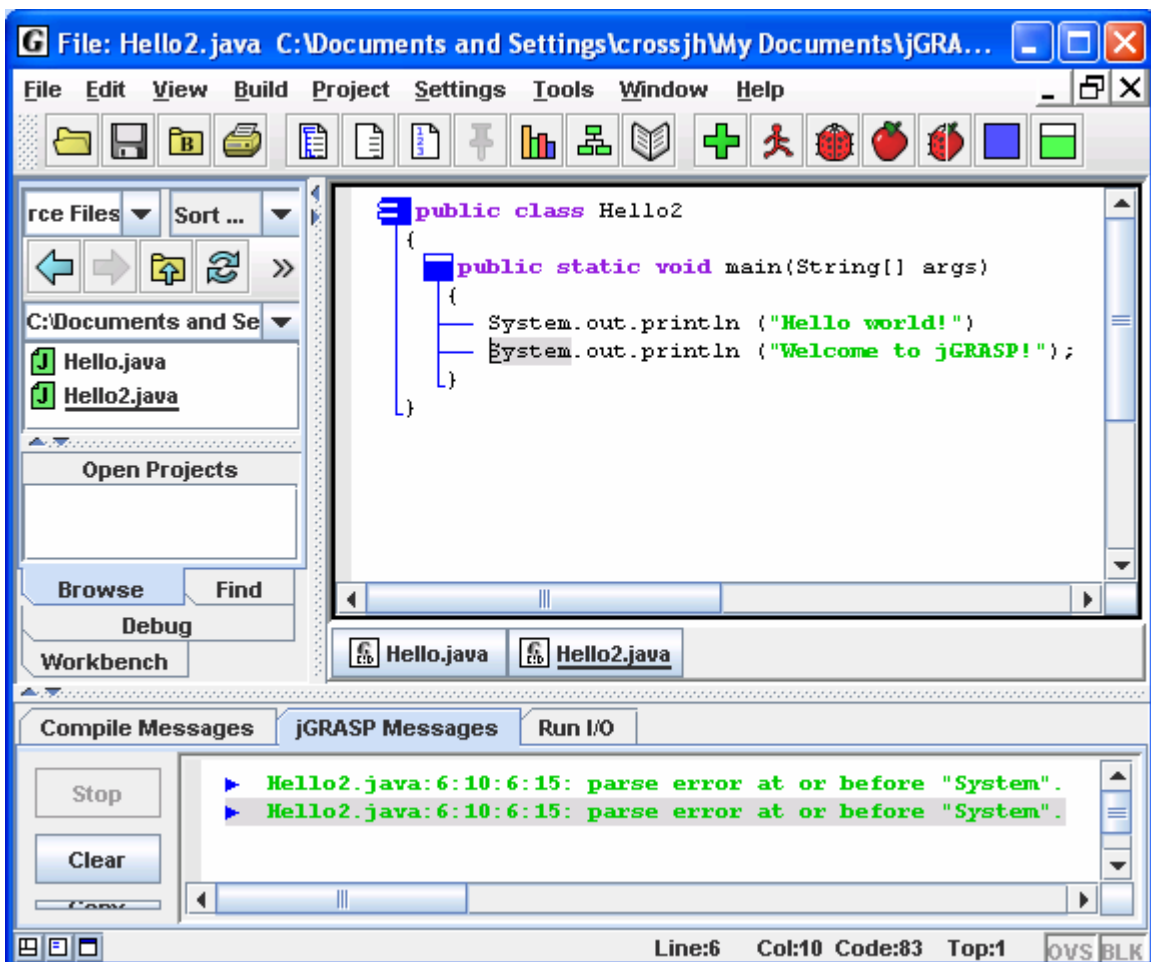


Figure 2-9. Syntax error detected

apparent as the size and complexity of your programs increase.

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.

2.6 Folding a CSD

Folding is a CSD 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 2-10). 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 is 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.

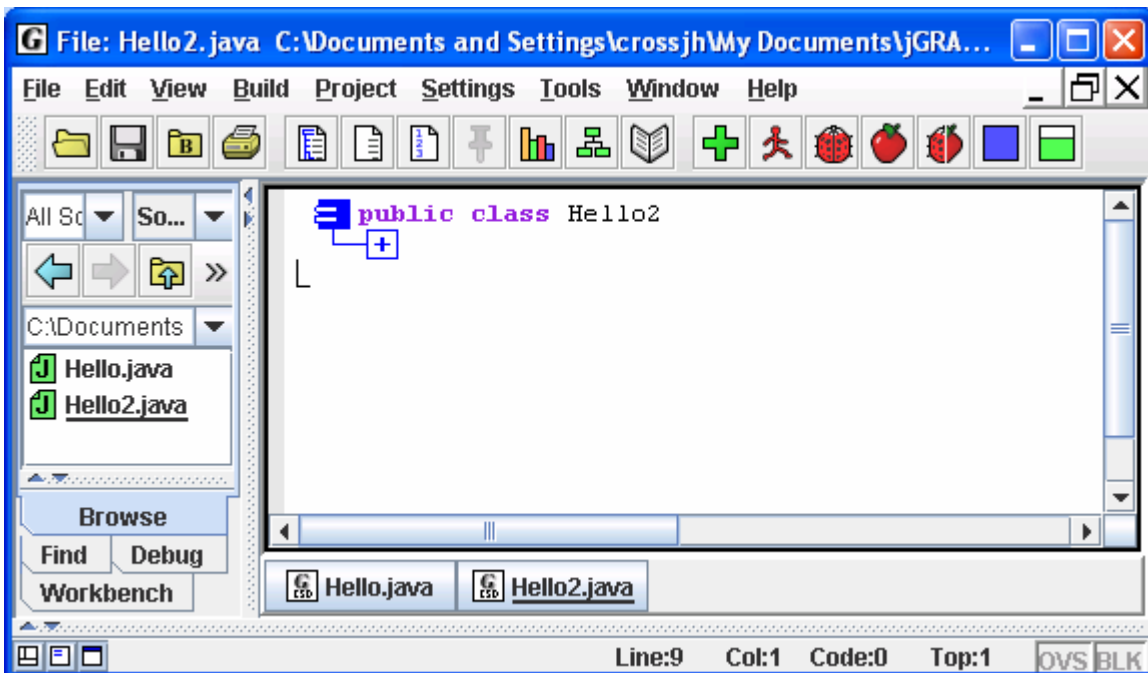




Figure 2-10. Folded CSD

2.7 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 2-11.

 Line numbers can be turned on and off by clicking the Toggle Line Numbers button on the CSD window toolbar or via the View menu.

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” the line numbers to avoid the incrementing by clicking on the Freeze Line Numbers button. To unfreeze the line numbers, click the button again. This feature is also available on the View menu.

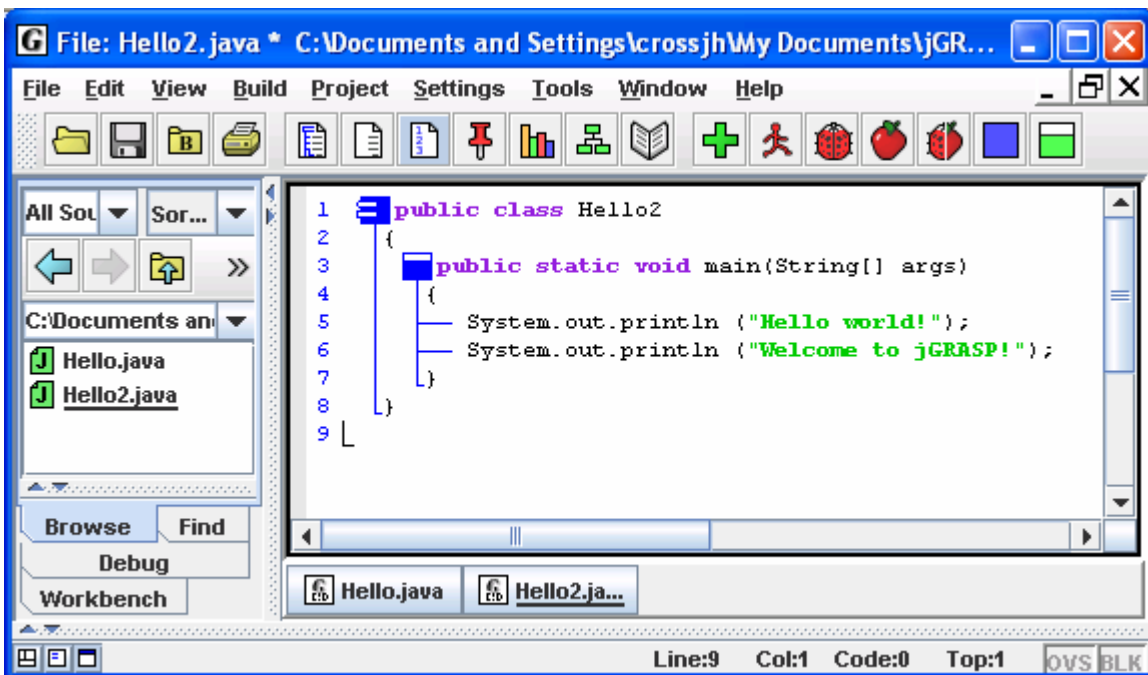




Figure 2-11. Line numbers in the CSD Window

2.8 Compiling a Program – A Few More Details

When you have a program in the CSD window, either by loading a source file or by typing it in and saving it, you are ready to compile the program. When you compile your program, the file is automatically saved if Auto Save is ON, which it is by default. Auto Save can be turned on/off by clicking **Settings – Auto Save**. If you are compiling a language other than Java, you will need to “compile and link” the program.

 Compile a Java program in jGRASP by clicking the Compile button or by clicking on the Compiler menu: **Build – Compile** (Figure 2-12).

 Compile and Link the program (if you are compiling a language other than Java) by clicking on the Compile and Link button or by clicking on the Build menu: **Build – Compile and Link**. Note, this option will not be visible on the toolbar and menu in a CSD window for a Java program.

In the figure below, also note that **Debug Mode** is checked ON. This should always be left on so that the `.class` file created by the compiler will contain information about variables in your program that can be displayed by the debugger and Object Workbench.

The results of the compilation will appear in the **Compile Messages tab** in the lower window of the Desktop. If your program compiled successfully, you should see the message “operation complete” with no errors reported, as illustrated in Figure 2-13. Now you are ready to “Run” the program (see 2.9 Running A Program – Additional Options).

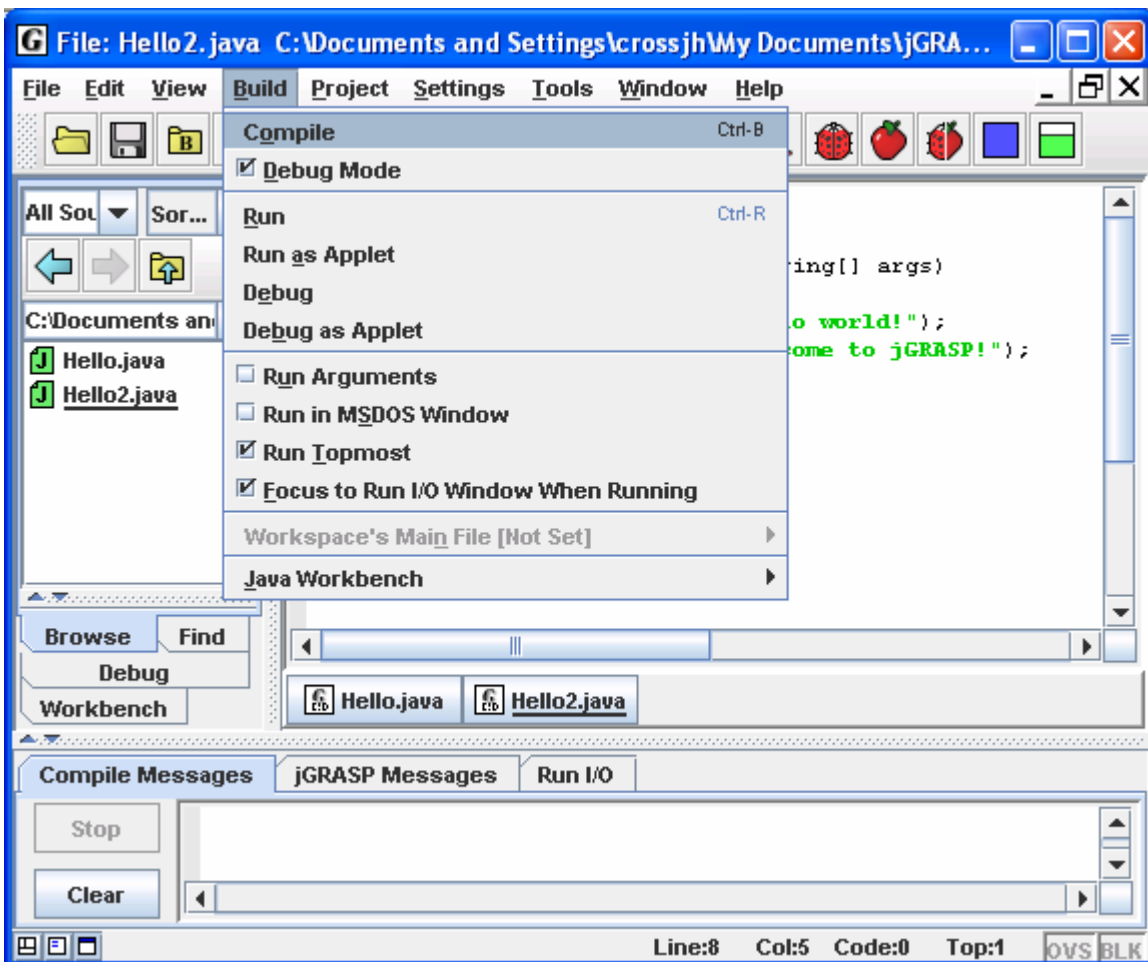


Figure 2-12. Compiling a program

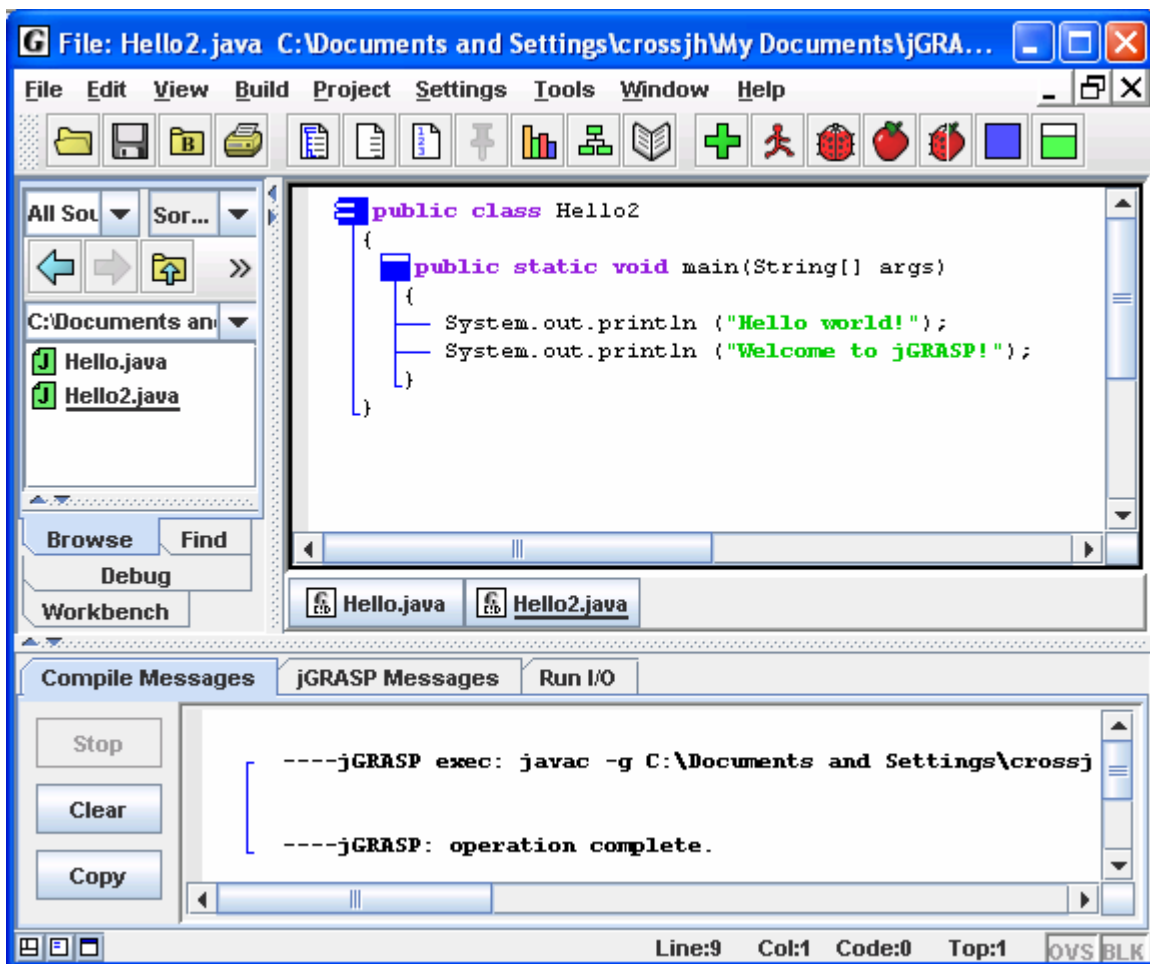


Figure 2-13. A successful compilation

Error Messages

An error message indicating “file not found,” generally means jGRASP could not find the compiler. For example, if you are attempting to compile a Java program and the message indicates that “javac” was not found, this means the Java compiler (javac) may not have been installed properly. Go back to Section 1, Installing jGRASP, and be sure you have followed all the instructions. Once the Java JDK is properly installed and set up, any errors reported by the compiler should be about your program.

Figure 2-14 shows a program with a missing “)” in the first println statement. The error description is highlighted in the Compiler Message tab, and jGRASP automatically scrolls the CSD window to the line where the error most likely occurred and highlights it.

If multiple errors are indicated, you should correct all that are obvious and then compile the program again. Sometimes correcting one error can clear up several error messages.

Only after you have “fixed” all reported errors will your program actually compile, which means a .class file will be created for your .java file. After this .class file has been created, you can “Run” the program as described in the next section.

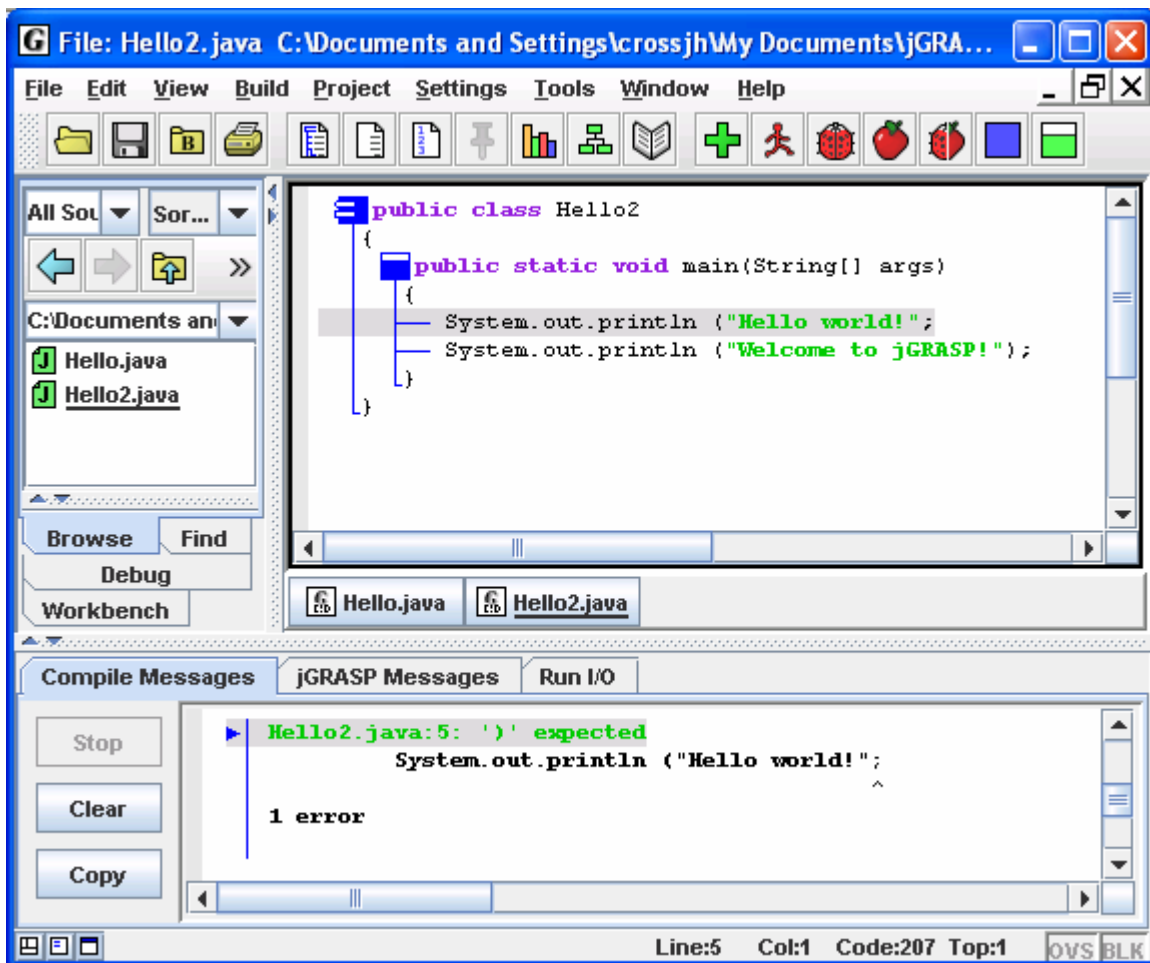


Figure 2-14. Compile time error reported

2.9 Running a Program - Additional Options

At this point you should have successfully compiled your program. Two things indicate this. First, there should be no errors reported in the Compile Messages window. Second, you should have a Hello2.class file listed in the Browse pane, assuming the pane is set to list "All Files."

To run the program, click **Build – Run** on the toolbar (Figure 2-15). The options on the Build menu allow you to run your program: as an application (**Run**), as an Applet (**Run as Applet**), as an application in debug mode (**Debug**), as an Applet in debug mode (**Debug as Applet**). Other options allow you to pass Run arguments, Run in an MS-DOS window rather than the jGRASP Run I/O message pane, and Run Topmost to keep frames and dialogs of the program on top jGRASP components.



You can also run the program by clicking the Run button on the tool bar.

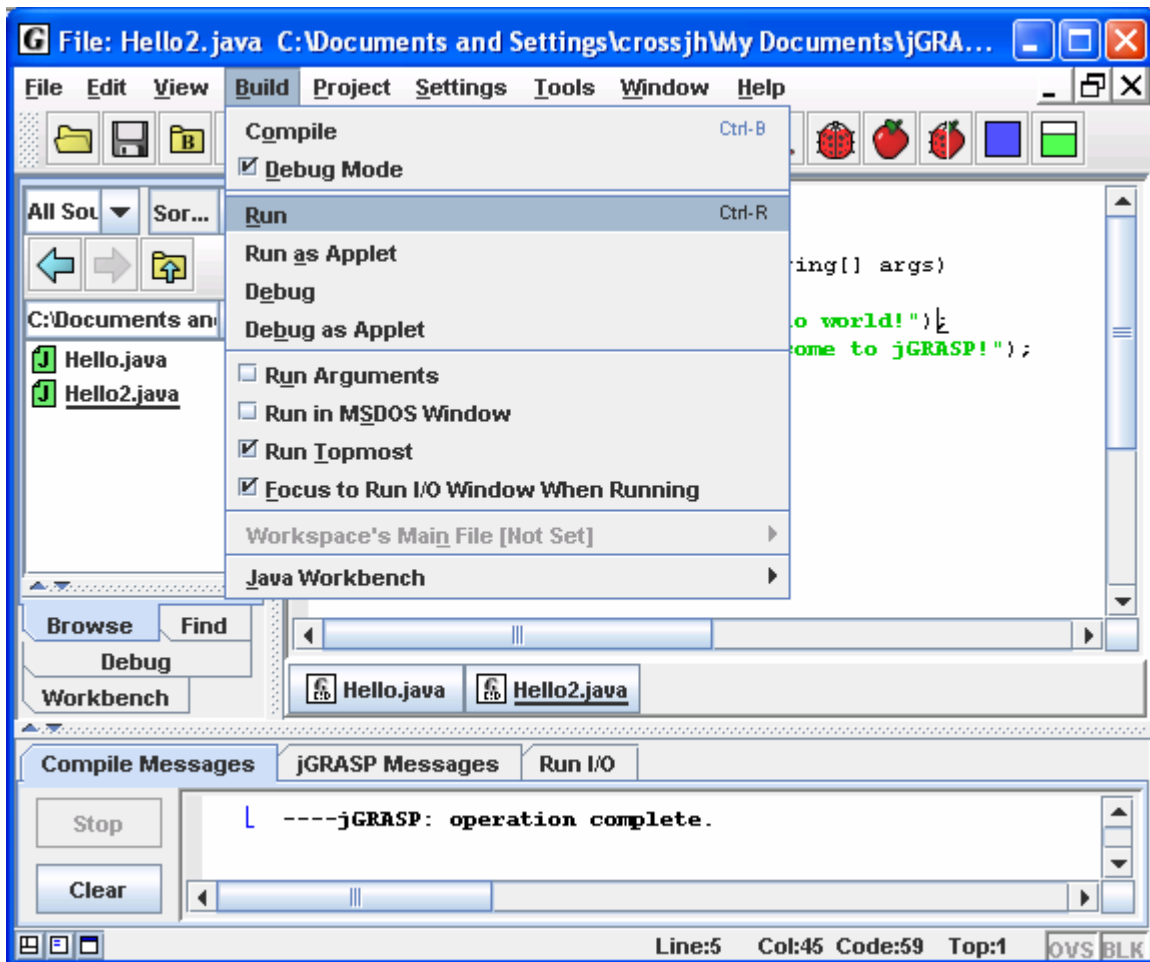


Figure 2-15. Running a program

Output

If a program has any standard input and/or output, the Run I/O tab in the lower pane pops to the top of the Desktop. In Figure 2-16, the output from running the Hello2 program is shown in Run I/O tab.

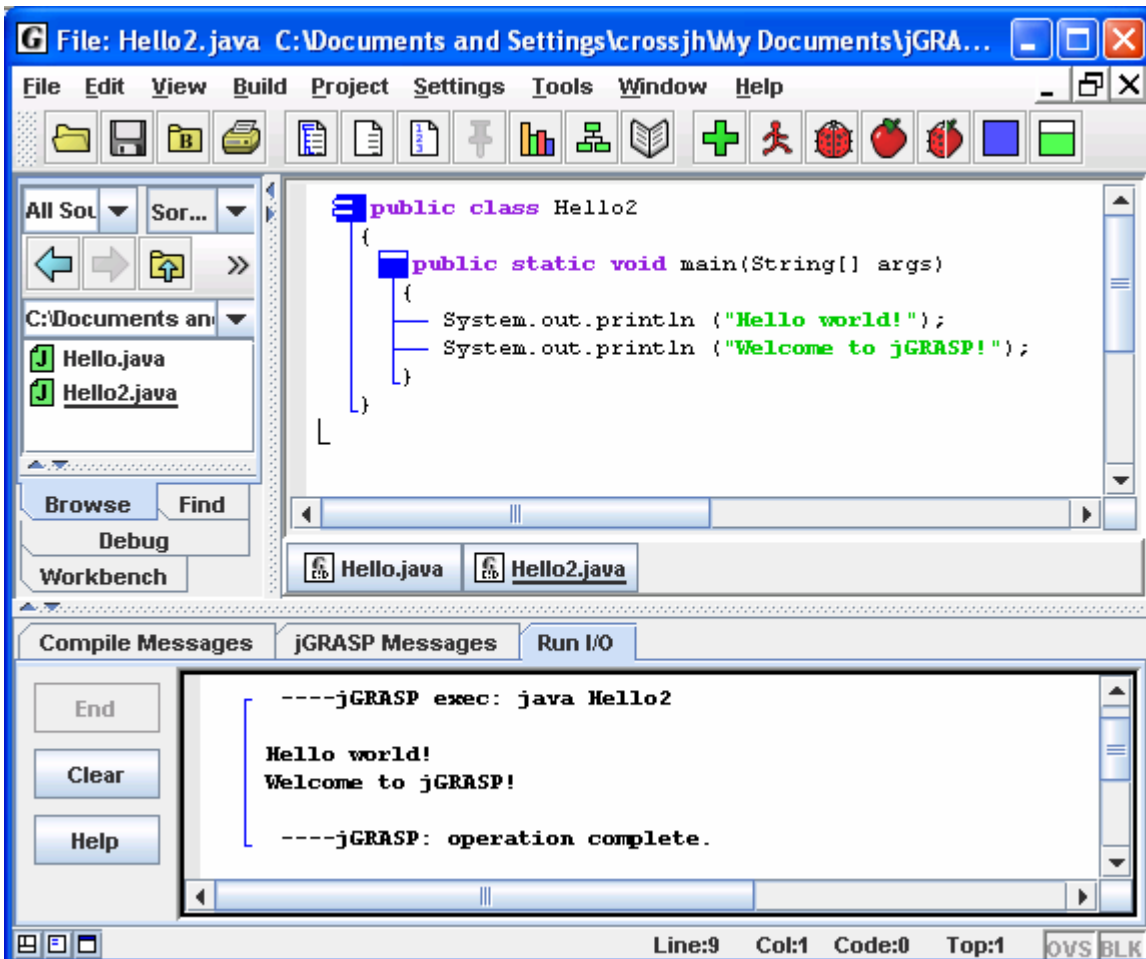




Figure 2-16. Output from running the program

2.10 Using the Debugger

jGRASP provides an easy-to-use visual Debugger 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. To set a breakpoint, hover the mouse over the gray column to the left of the line where you want to set the breakpoint. When you see the red breakpoint symbol, left-click the mouse to 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 2-17). 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.

To start the debugger on an application, click the debug button  on the toolbar. Alternatively, you can click **Build – Debug**. When the debugger starts, the Debug tab

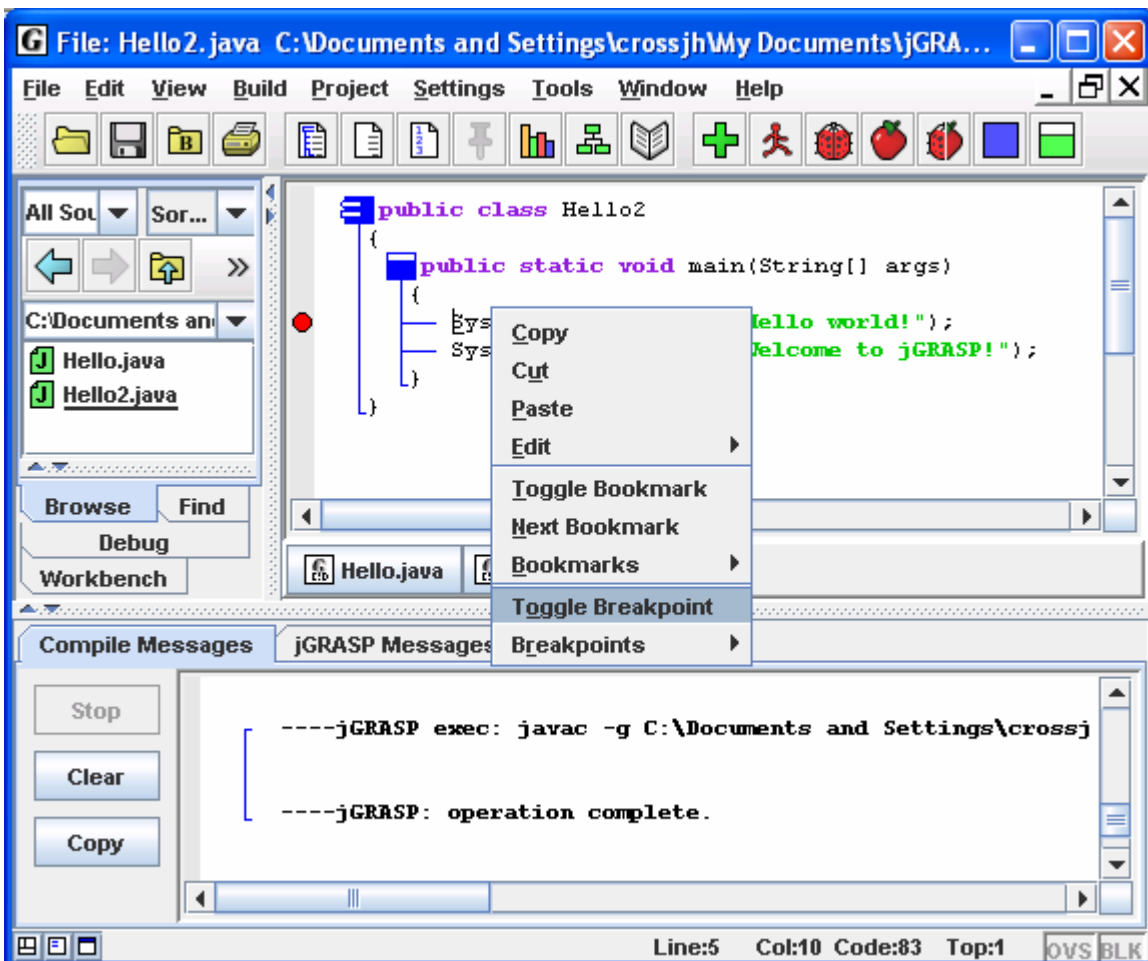



Figure 2-17. Setting a breakpoint

should pop up in place of the Browse tab, and your program should stop at the breakpoint as shown in Figure 2-18 below.

The debugger control buttons are



located at the top of the Debug tab. Only one of the buttons 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 pane will go blank to signal 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.

In the example below, 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 pane. 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 pane 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. For a complete description of the other debugger control buttons, see the tutorial on the *Integrated Debugger*.

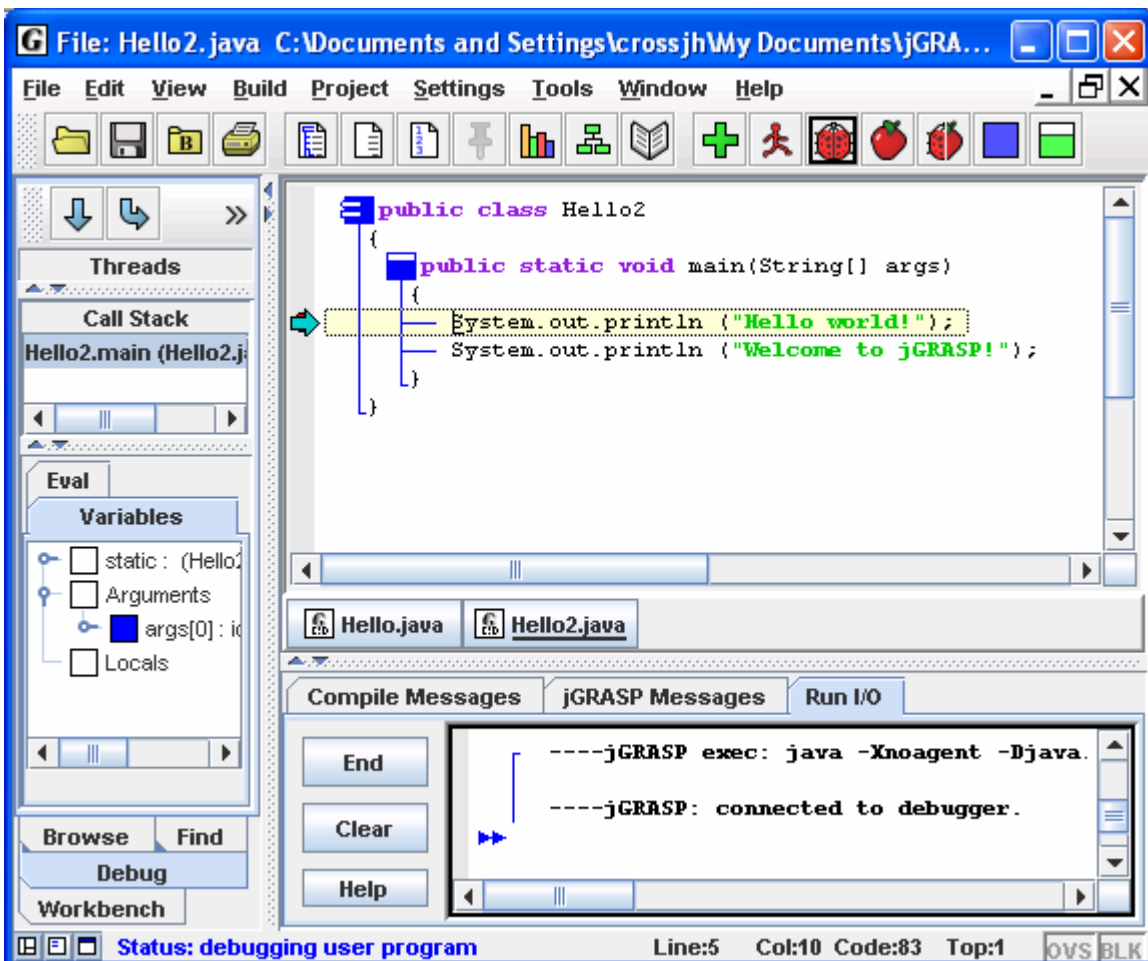



Figure 2-18. Stepping with the Debugger

2.11 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 **2.1 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 illustrated in Figure 2-19.

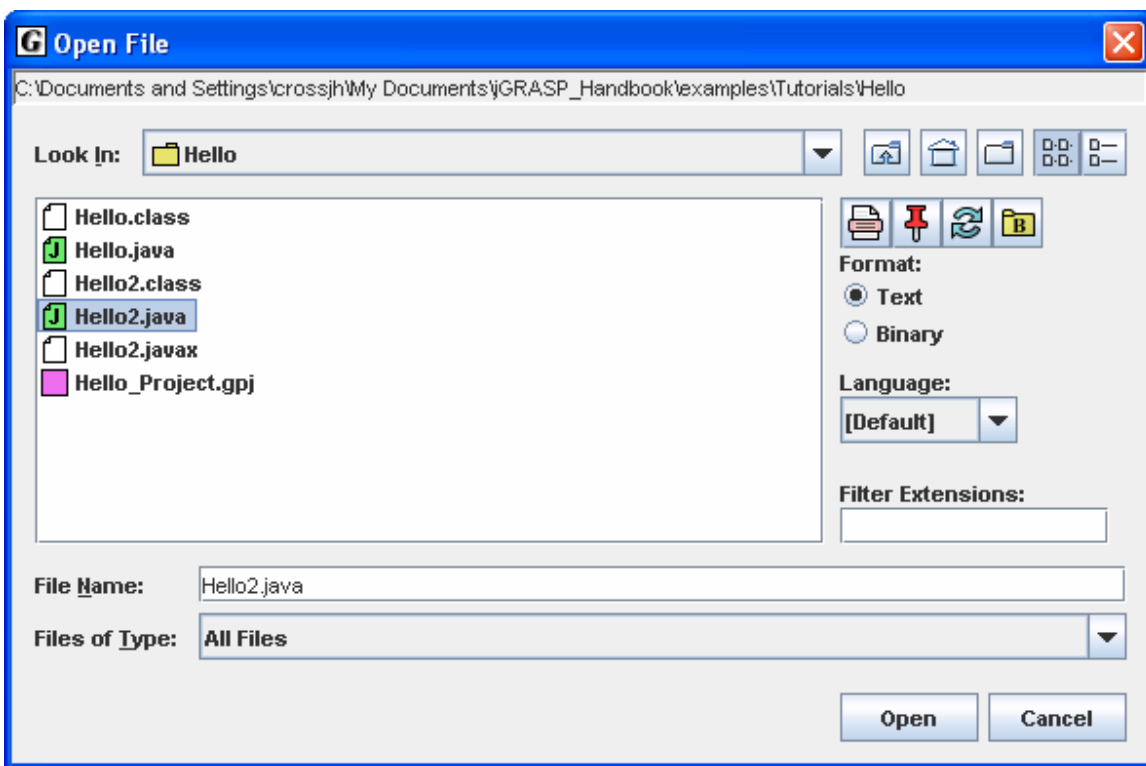


Figure 2-19. Open File dialog

- (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. Each program can be compiled and run from its respective CSD window. When multiple windows are open, the single menu and toolbar go with the top window only, which is said to have “focus” in the desktop. In Figure 2-20, two CSD windows have been opened. One contains Hello.java and the other contains Hello2.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, 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. As described earlier, these buttons can be reordered by dragging/dropping them on the windowbar. In the figure below, the windowbar has buttons for Hello and Hello2. Notice that Hello2.java is underlined both on the windowbar and in the Browse tab to indicate that it has the current focus. Hello2.java is also displayed in the desktop’s blue title bar.

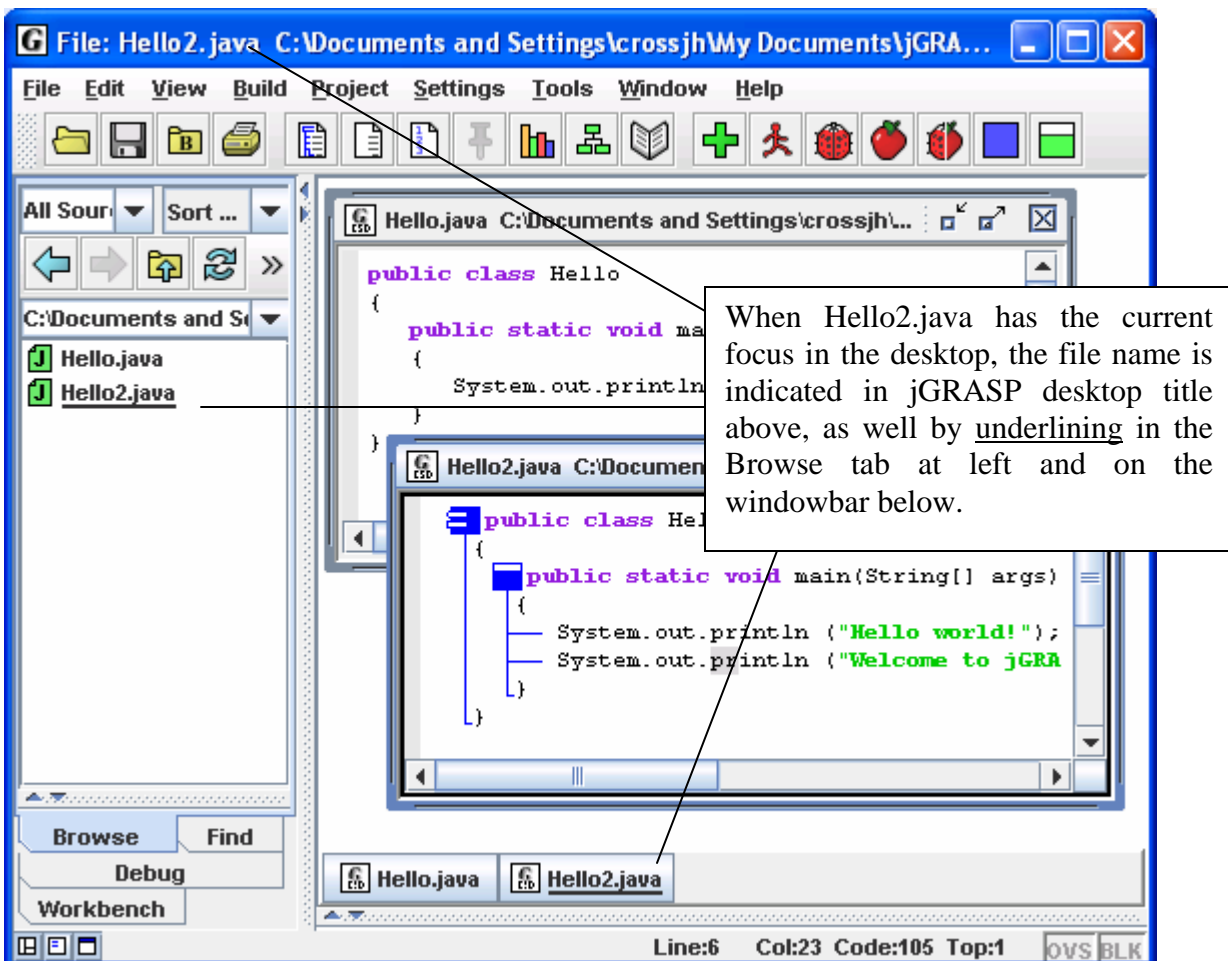




Figure 2-20. Multiple files open

2.12 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 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**.
- (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. After the files are closed, your Desktop should look like the figure below, which is how we began this tutorial.

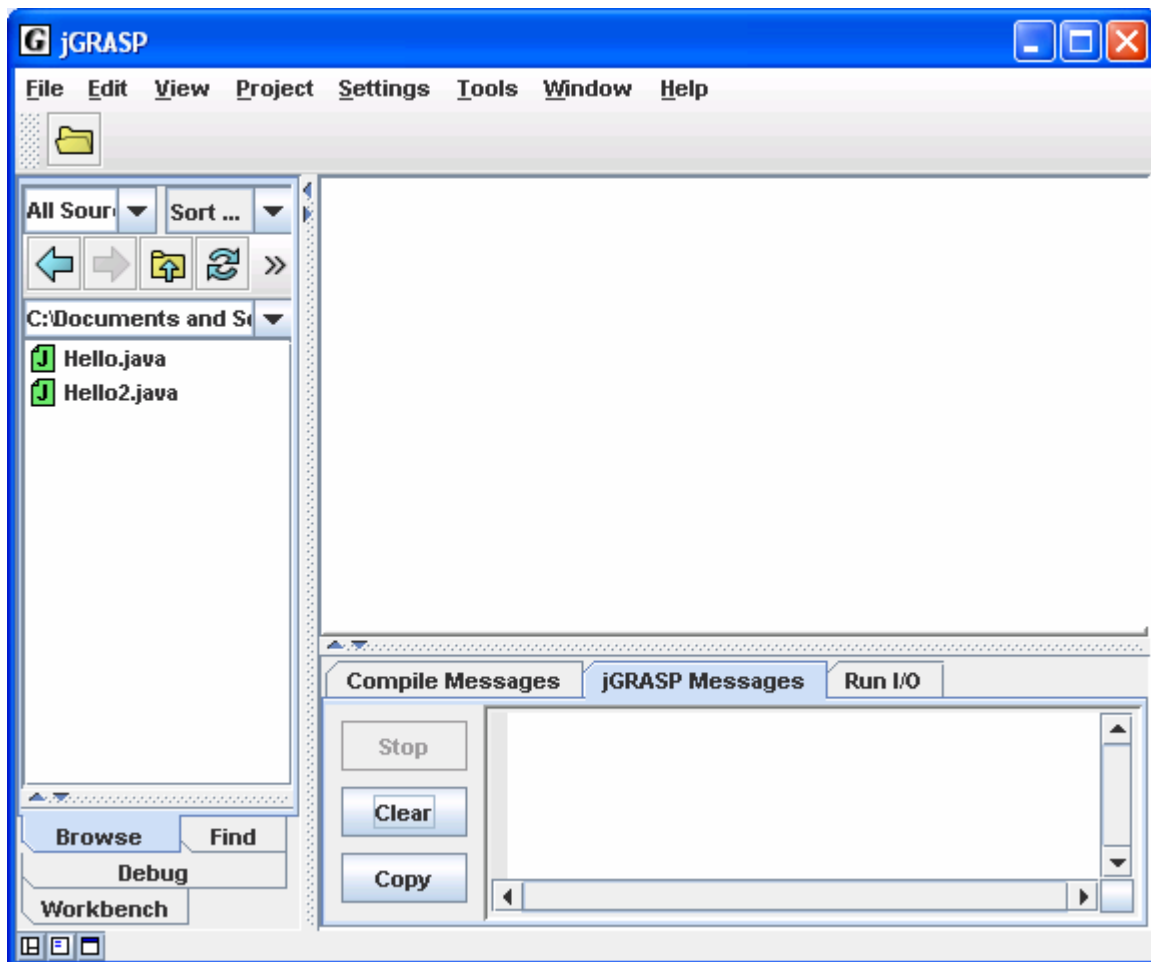



Figure 2-22. Desktop with all CSD Windows closed

2.13 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**.

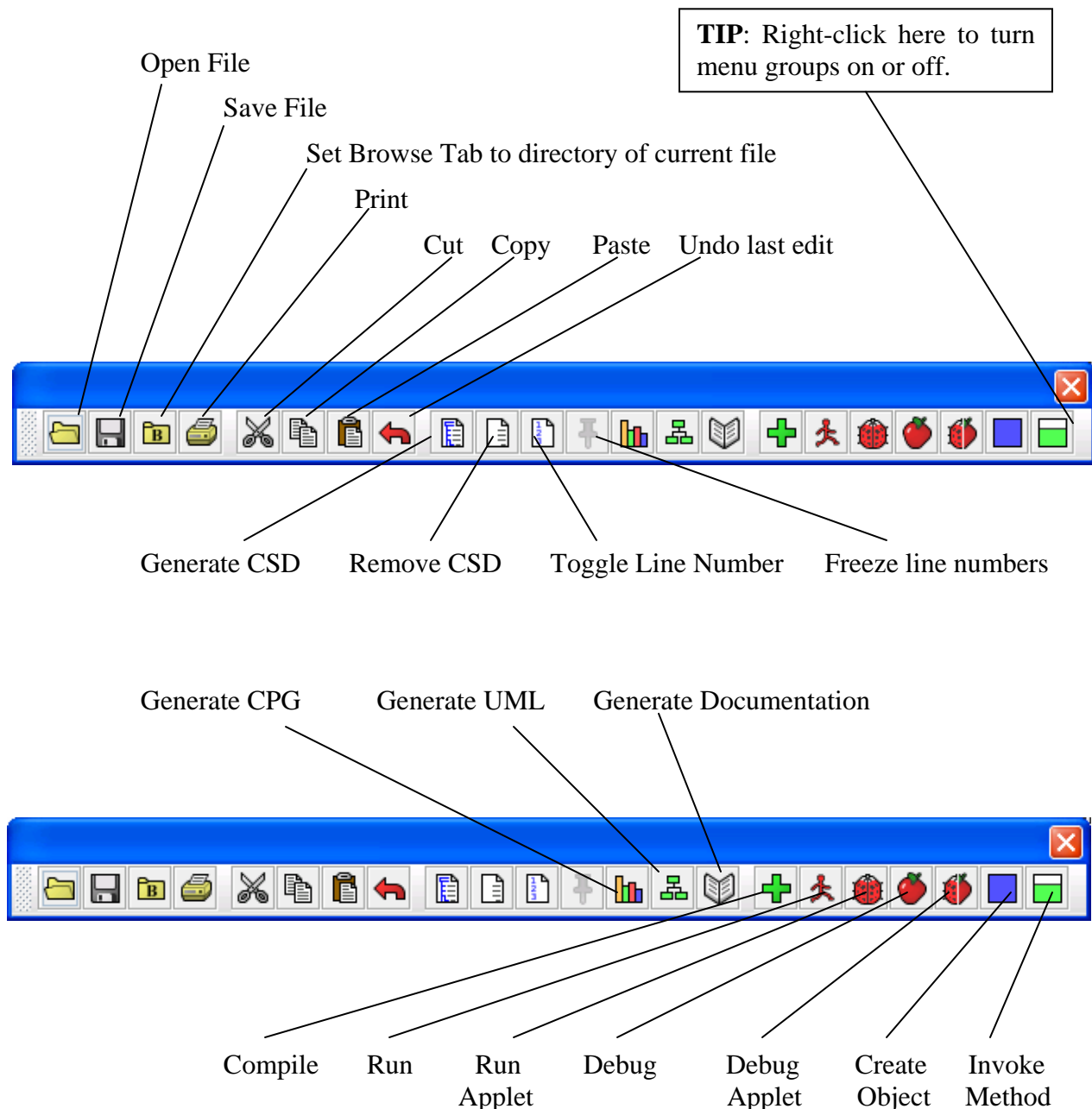
2.14 Exercises

- (1) Create your own program then save, compile, and run it.
- (2) Generate the CSD for your program. On the View menu, turn on Auto Generate CSD (**Settings – CSD Window Settings** – then (checkbox) **Auto Generate CSD**).
- (3) Display the line numbers for your program.
- (4) Fold up your program then unfold it in layers.
- (5) 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.] Recompile your program.
- (6) 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.
- (7) If you have other Java programs available, open one or more of them, then repeat steps (1) through (5) above for each program.

2.15 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.

While many of these buttons were introduced in this section, some were assumed to be self-explanatory (e.g., Print, Cut, Copy, etc.). Several others will be covered in the next section along with Projects and the Object Workbench (e.g., Generate UML, Generate Documentation, Create Object, and Invoke Method). Section 9 provides an in depth look at the CSD, which can be read at any time, but is most relevant when control structures are studied (e.g., selection, iteration, try-catch, etc.).



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.


Objectives – When you have completed this tutorial, you should be able to use projects, UML class diagrams, the Object Workbench, and Viewers 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
 - 3.7.1 Viewing the source code for a class
 - 3.7.2 Displaying class information
 - 3.7.3 Displaying Dependency Information
- 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 Object Parameters
- 3.13 Invoking Methods on Object Fields
- 3.14 Invoking Inherited Methods
- 3.15 Running the Debugger on Invoked Methods
- 3.16 Creating Objects from the CSD Window
- 3.17 Creating an Instance from the Java Class Libraries
- 3.18 Exiting the Workbench
- 3.19 Closing a Project
- 3.20 Exiting jGRASP
- 3.21 Exercises

3.1 Starting jGRASP

A Java program consists of one or more class files, each of which defines a set of objects. During the execution of the program, objects 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.

 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 don't see the jGRASP icon on the desktop, try the following: click **Start – Programs – 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 plus three panes. The *left pane* has tabs for **Browse**, **Find**, **Debug**, and **Workbench**. The large *right pane* is for UML and CSD windows. The *lower pane* has tabs for jGRASP messages, Compile messages, and Run Input/Output.

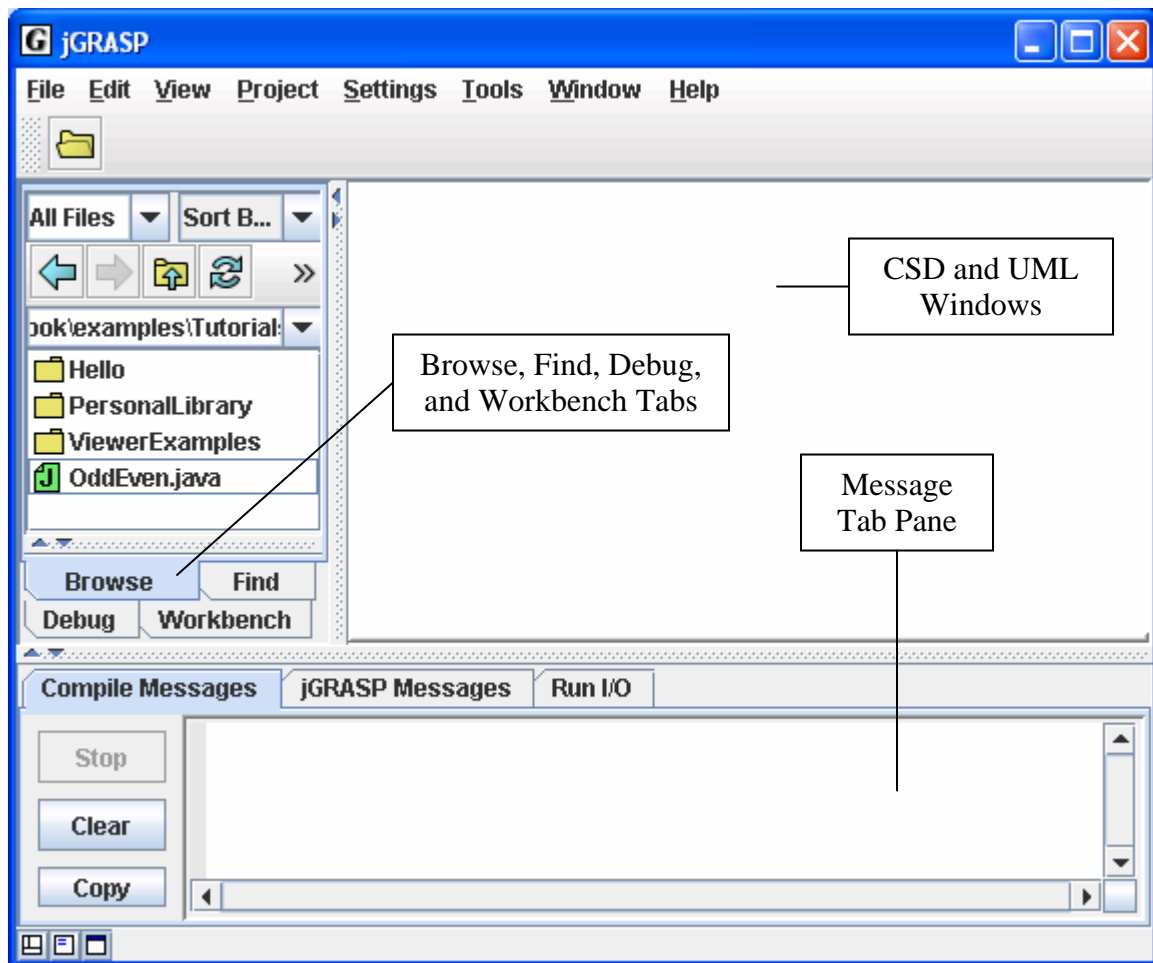


Figure 3-1. The jGRASP Virtual Desktop

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). If jGRASP was installed by a system administrator, you may not have write privileges for these files. If this is the case, you should copy the Tutorials folder to one of your own folders (e.g., in your *My Documents* folder).

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 “R” refreshes the Browse pane. In the example below, the Browse tab is displaying the contents of the Tutorials folder.

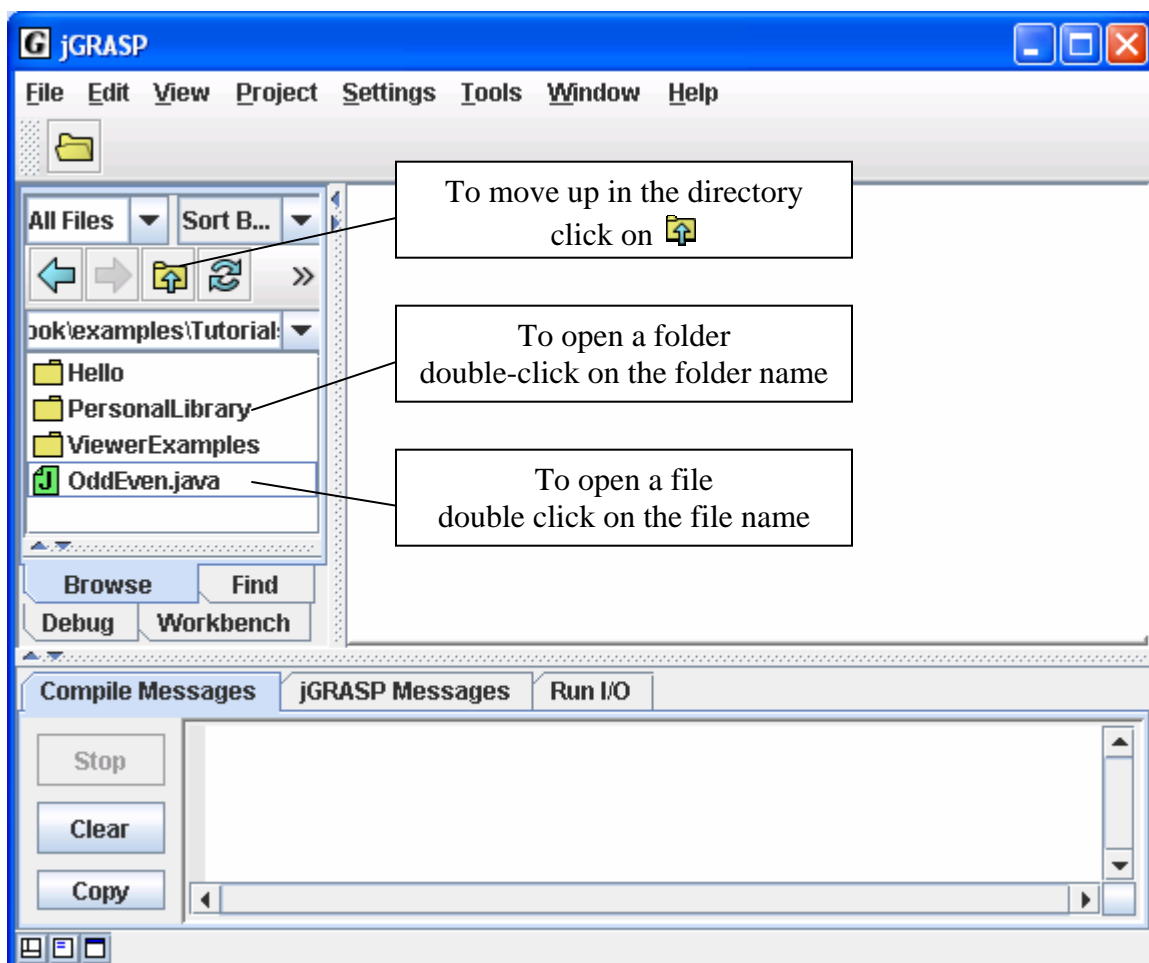


Figure 3-2. The jGRASP Virtual Desktop

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. To open the project, double-click on the project file (PersonalLibraryProject.gpj), 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 indicated below.

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 and place them on the Workbench and then invoke their methods. Both of these approaches are explored below.

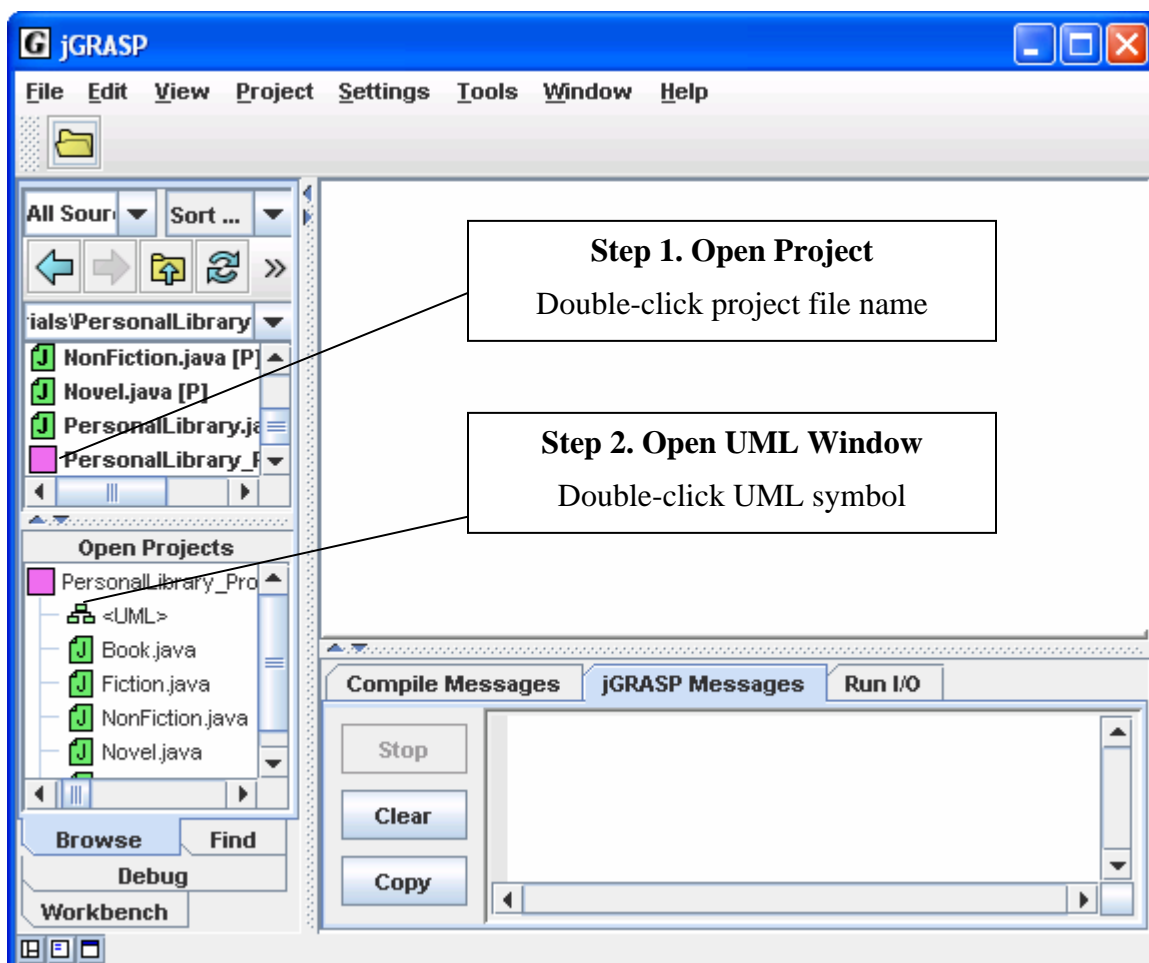





Figure 3-3. After loading file into CSD Window

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 .

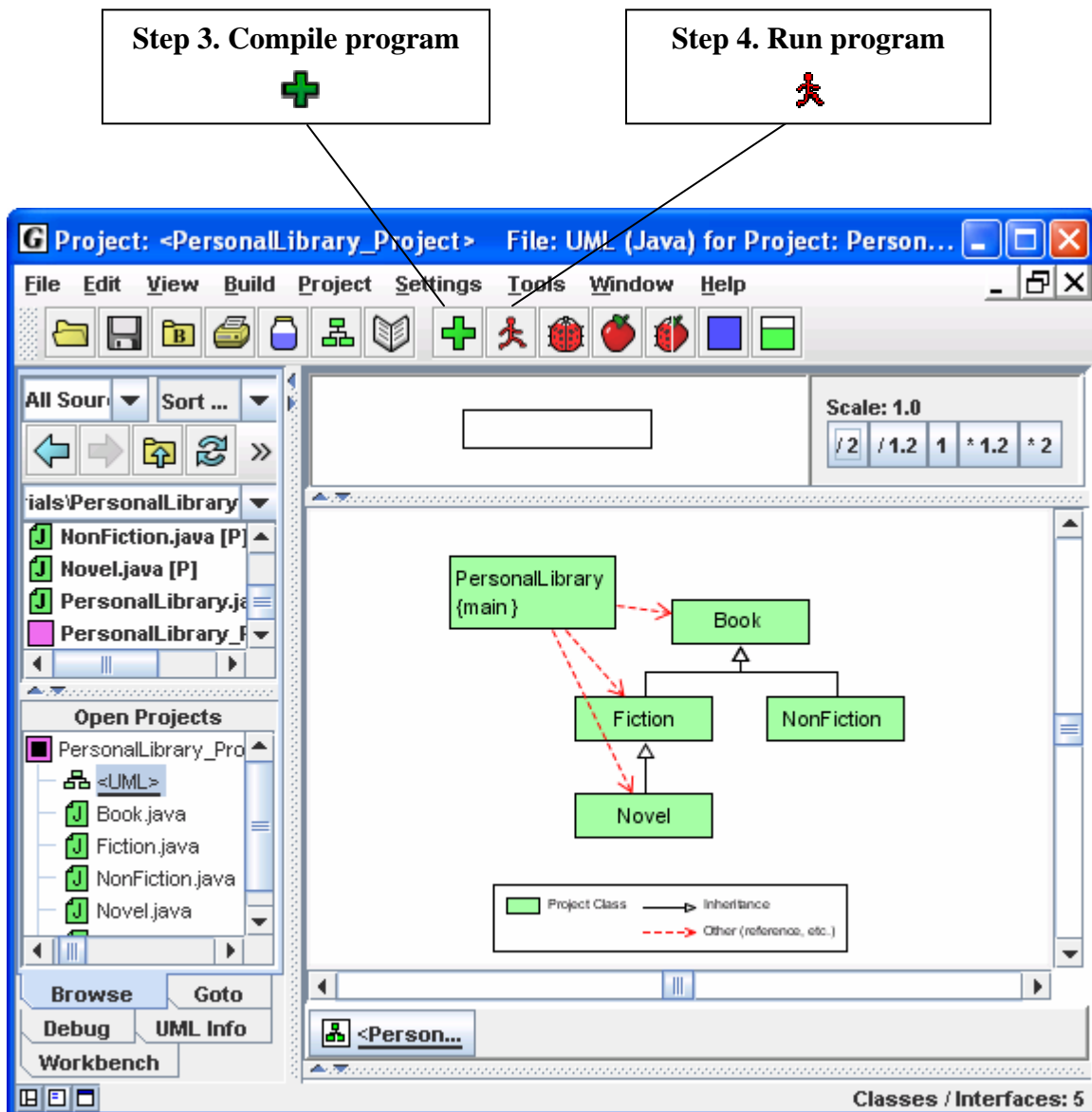
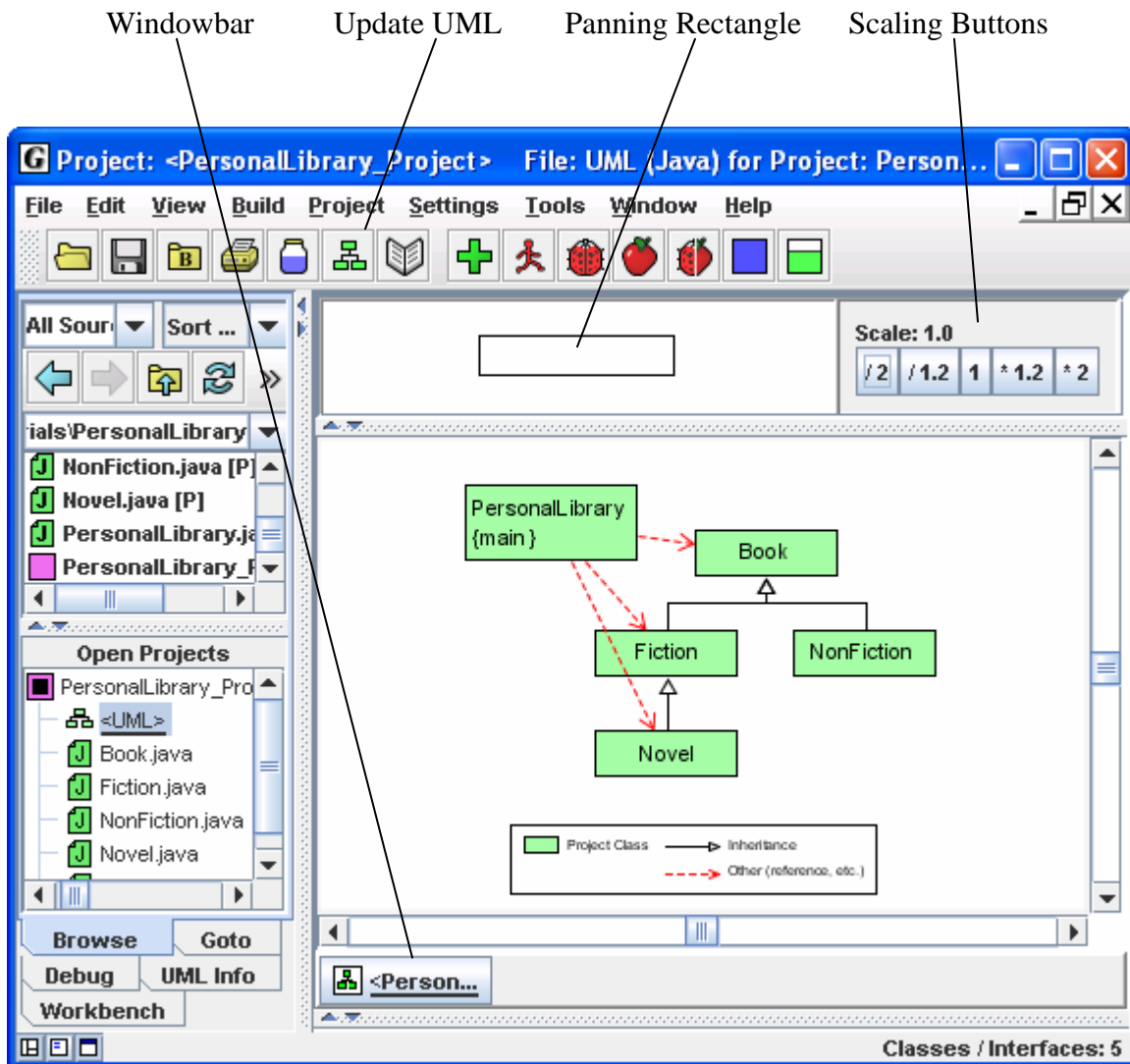


Figure 3-4. After loading file into CSD Window

3.5 Exploring the UML Window

In the figure below, the UML window has been opened for the PersonalLibraryProject 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” always 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.



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 (on/off), Compile, Run, Create Instance, and Invoke Method, as well as buttons for Create Instance and Invoke Method.

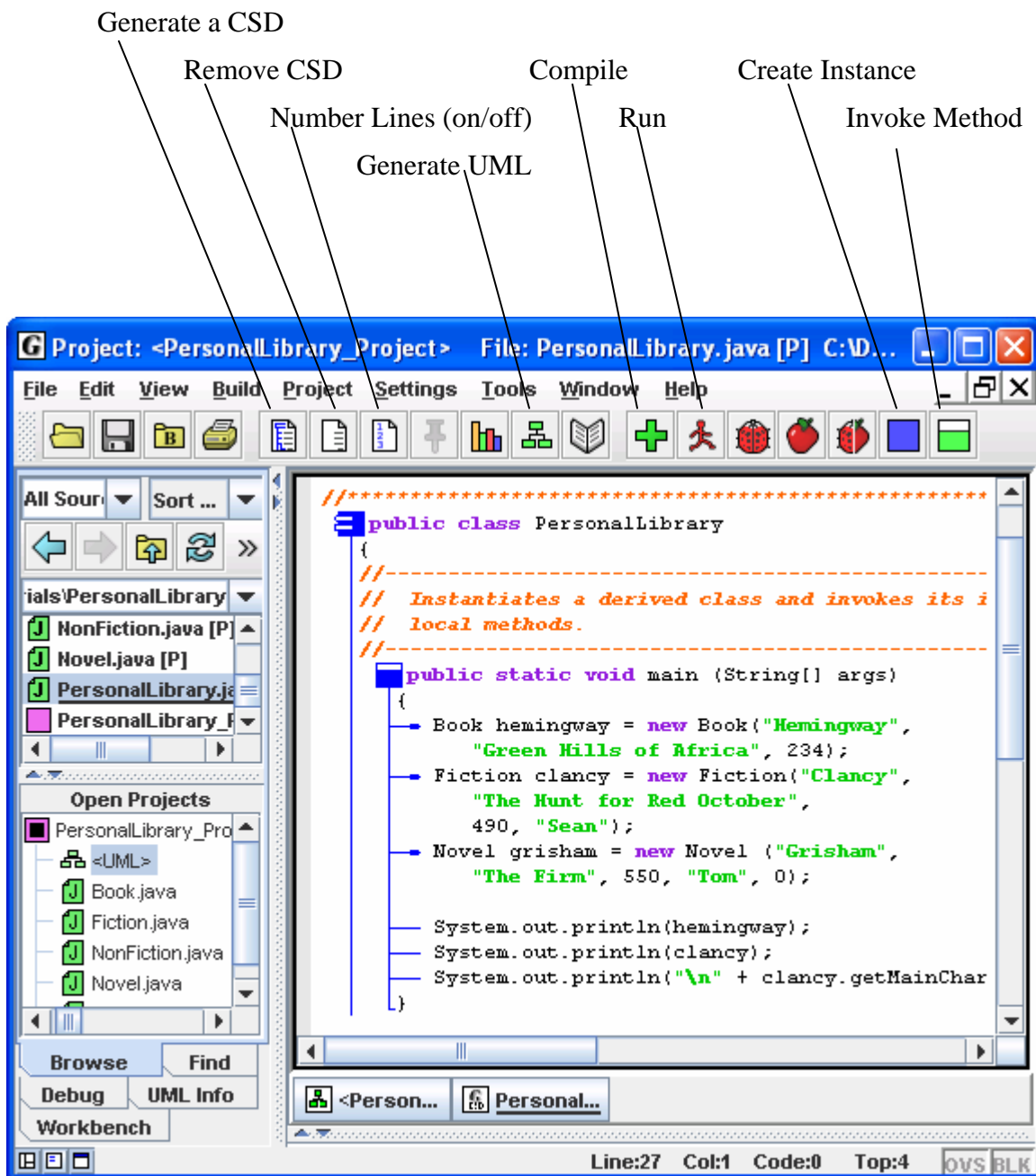


Figure 3-6. After the CSD is generated

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 next page or [Sec 2.5-2.7](#) 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, i.e., an application programmer interface (API). To begin the process of generating the documentation, click **Project – Generate Documentation**. Alternatively, 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.

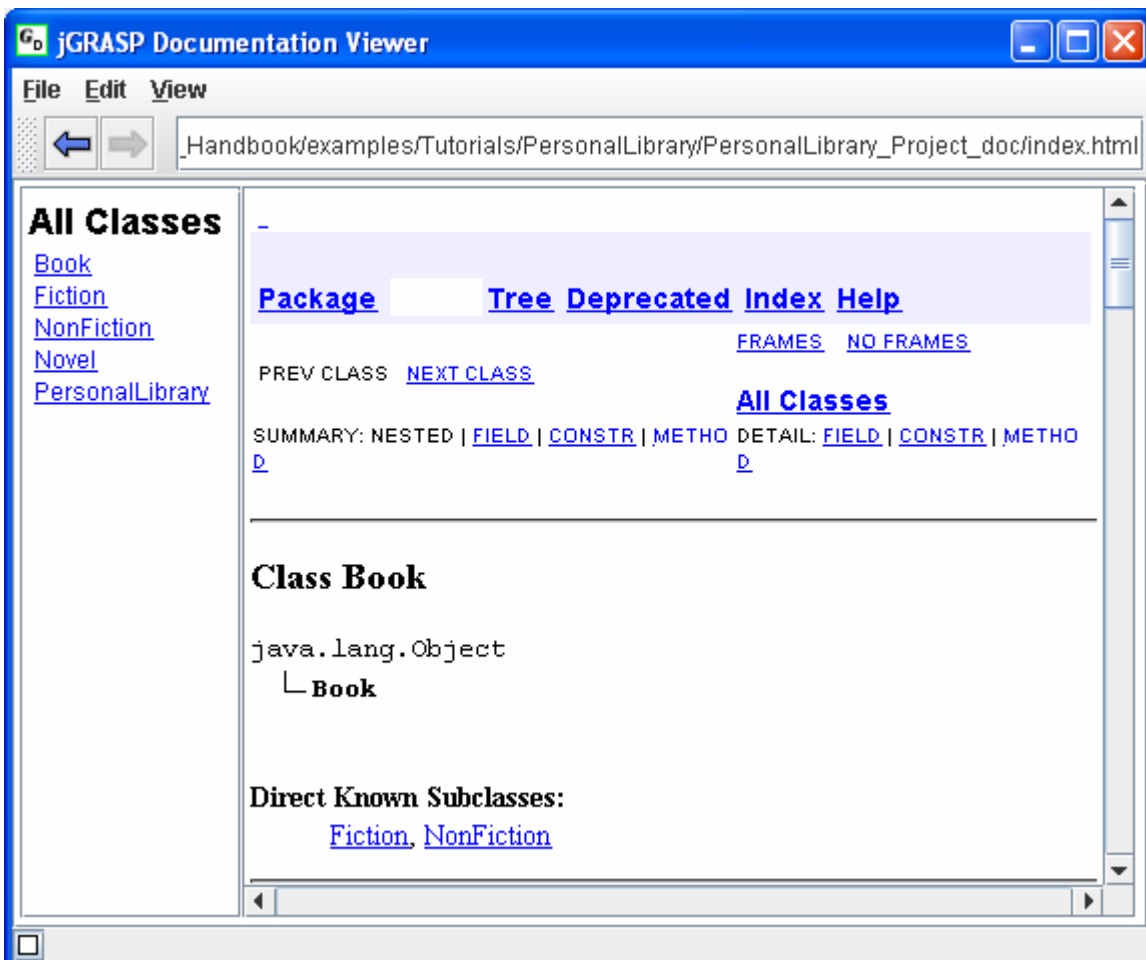





Figure 3-7. After generating documentation for PersonalLibraryProject

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.

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.

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

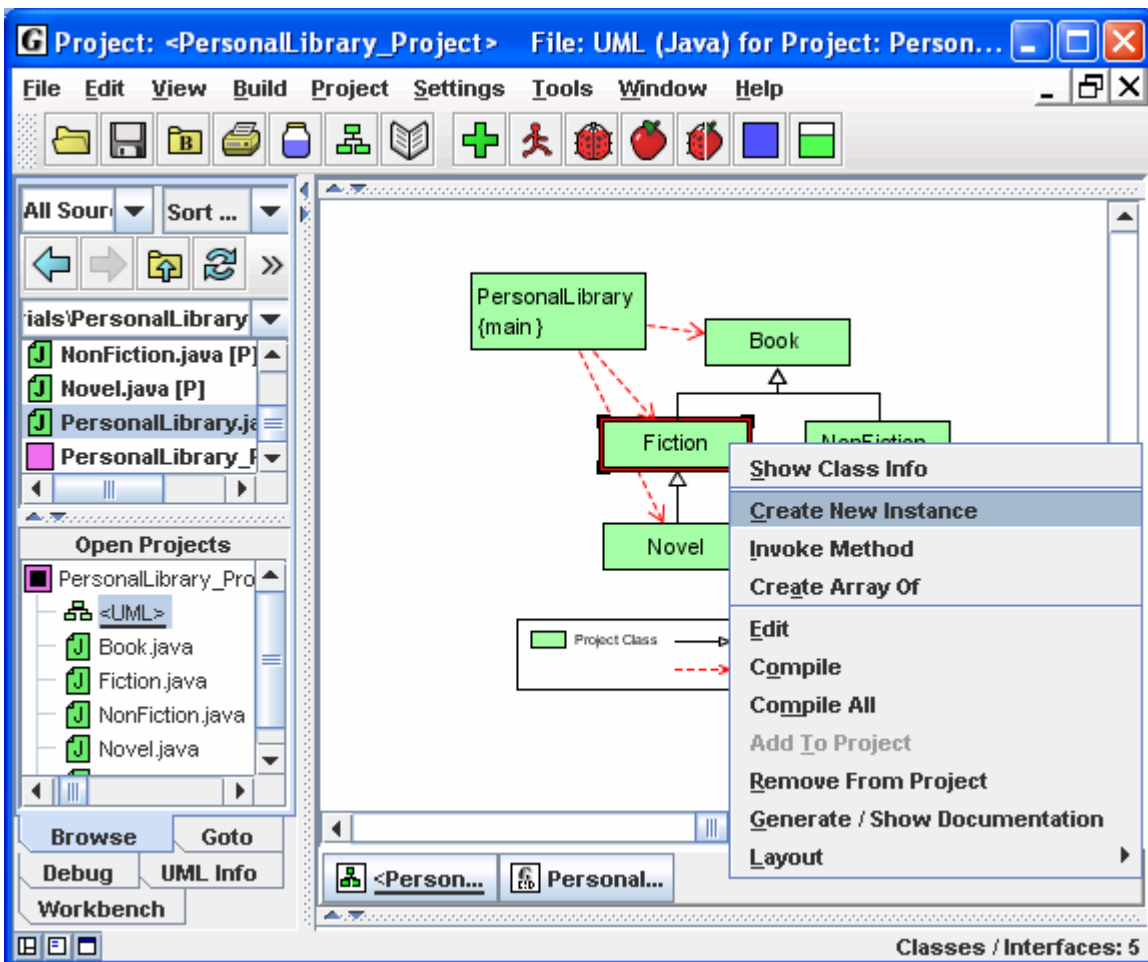


Figure 3-8. Creating an Object for the Workbench

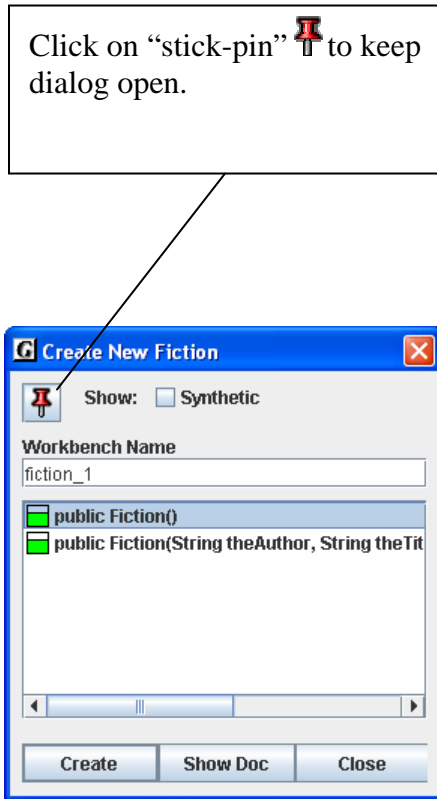


Figure 3-9. Selecting a constructor

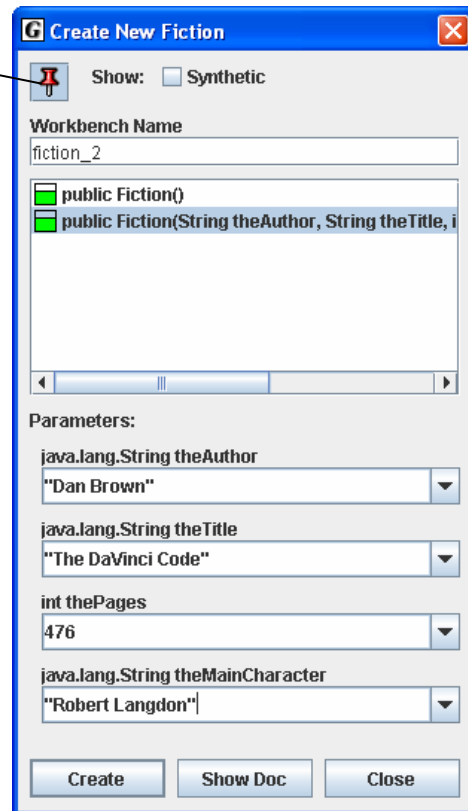



Figure 3-10. Constructor with parameters

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 double-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. 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., mainCharacter) 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 inherited from a super class (e.g., author inherited from Book). Finally, a red border on a symbol means the field is inaccessible outside the class (i.e., the object was declared as either private or protected).

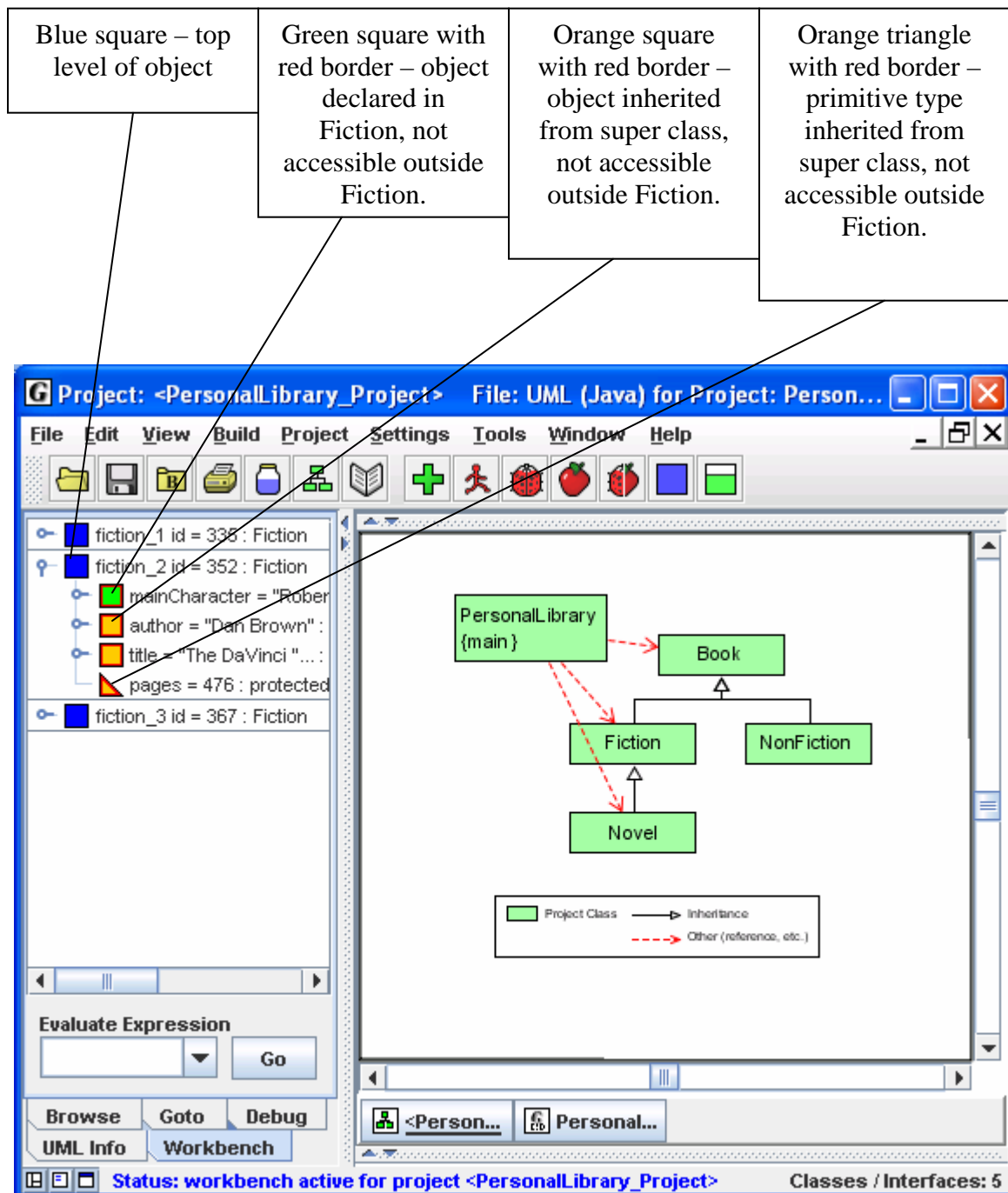


Figure 3-11. Workbench with three objects

3.10 Opening a Viewer Window

A separate *Viewer* window can be opened for any object or field of an object on the workbench. 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 always provides the same *basic* view shown on the workbench. 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 *title* field in *fiction_2*.

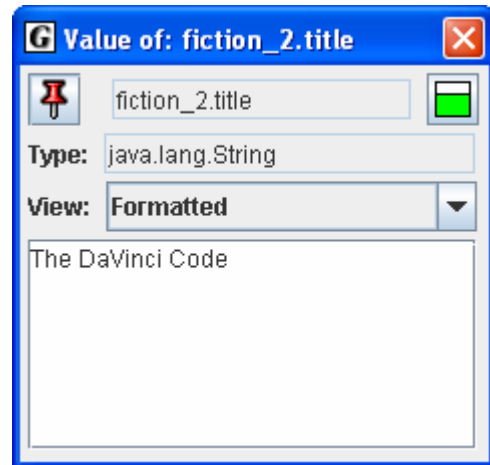



Figure 3-12. Viewer on String title field of *fiction_2* (*fiction_2.title*)

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.

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.

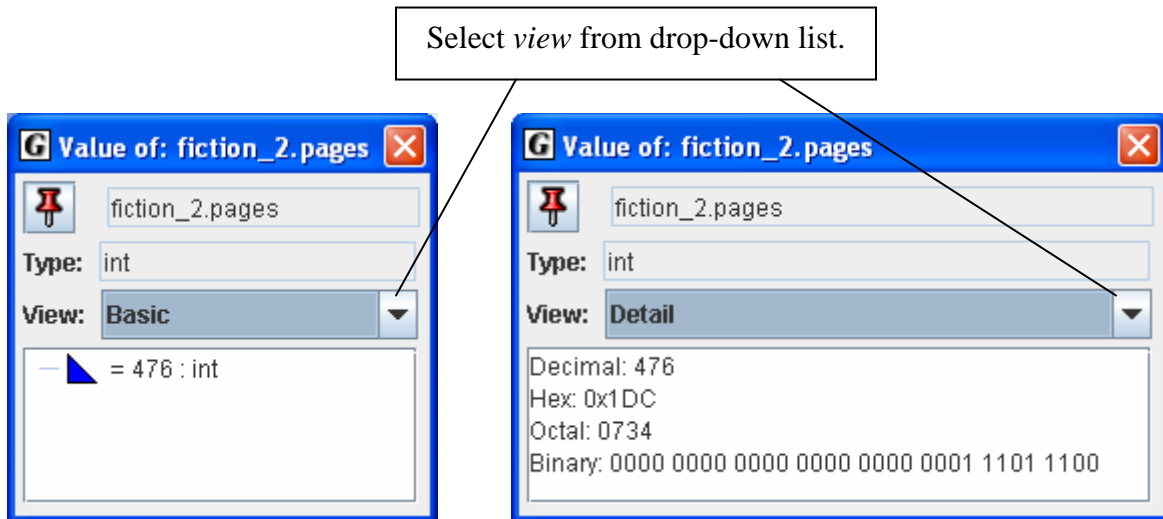


Figure 3-13 Viewer with Basic View of Primitive int

Figure 3-14 Viewer with Detail View of Primitive int

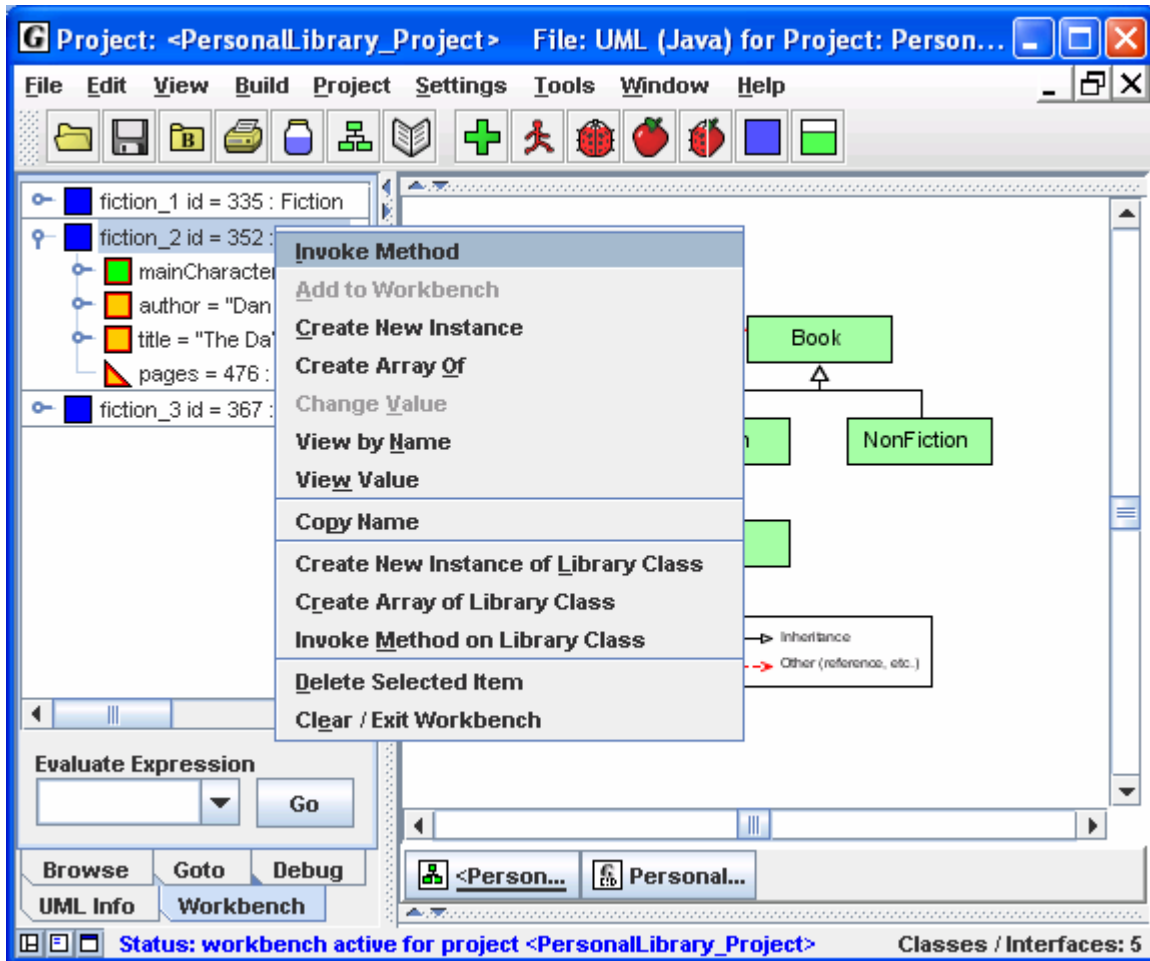



Figure 3-15. Workbench with two instances of Fiction

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, then click **Invoke**. This will execute the

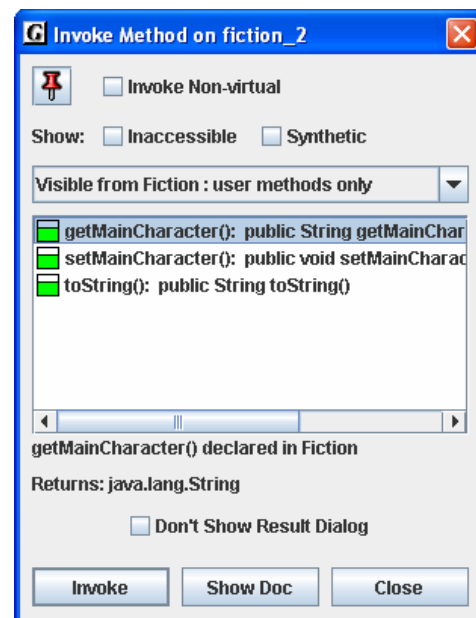


Figure 3-16. Selecting a method

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 been generated for the project.

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, we can invoke each of its three methods: `getMainCharacter()`, `setMainCharacter()`, and `toString()`. 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 Object Parameters

In the example above, we created two 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 for the workbench first. Then you can simply drag the object (actually a copy) from the workbench to the parameter field in the Invoke Method dialog. You can also use the `new` operator 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`, and `pages`). 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

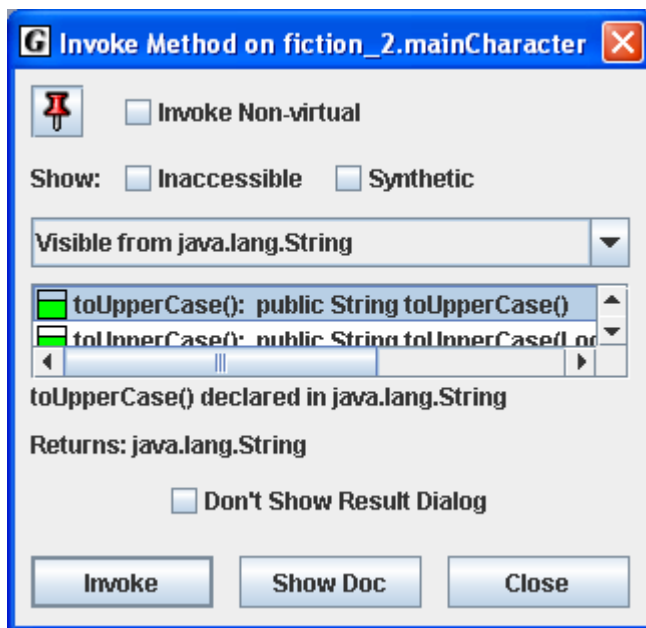


Figure 3-15. Invoking a String method

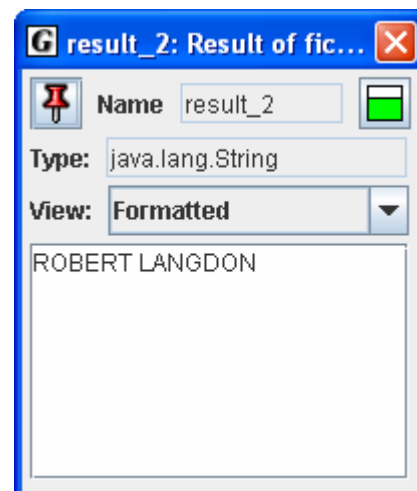


Figure3-16. Result of
fiction_1. mainCharacter.
toUpperCase()

mainCharacter, select **Invoke Method**. When the dialog pops up (Figure 3-15), scroll down and select the first *toUpperCase()* method and click **Invoke**. This should pop up the Result dialog with “NONE” as the return value (Figure 3-16). 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.

3.14 Invoking Inherited Methods

The methods we have invoked thus far were declared in the class from which we created the object. An object also inherits methods from its parents. We now consider an instance of the Novel class, which inherited several methods from the Book class in our example. If we right-click on the novel_1 in the Workbench tab pane (shown below fiction_2 in Figure 3-11) and select **Invoke Method**, the dialog in Figure 3-17 pops up. Since the default is to list visible user method, *toString()* method and the two inherited user methods are listed. Notice the orange color coding indicating “inherited” methods similar to the fields on the workbench. To view other lists of methods, find the drop-down list located above the list. A gray symbol in front of a method indicates it has been overridden by another method in the category.

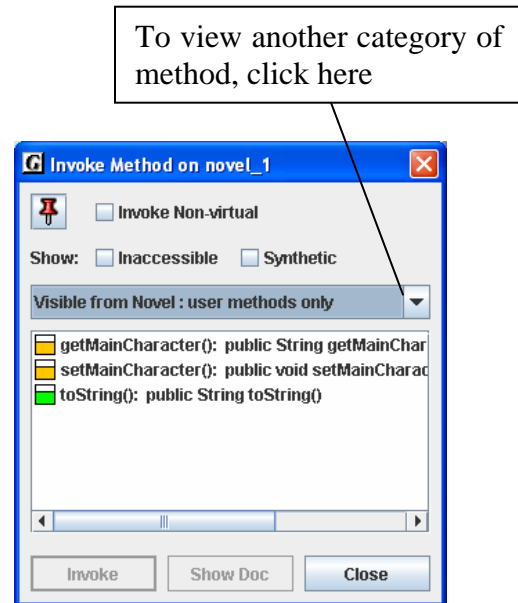





Figure 3-17. Invoking a method for novel_1

3.15 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 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.

3.16 Creating Objects from the CSD Window

In addition to creating instances of classes from the UML class diagram as described above, instances can be created directly from the CSD window for the class it contains. Figure 3-18 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 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. In Figure 3-18, the Fiction class has been opened in a CSD window without a project being opened, and two instances have been placed on the workbench. Figure 3-19 shows a viewer for fiction_2. Notice the viewer has its own button  for invoking the methods of fiction_2.

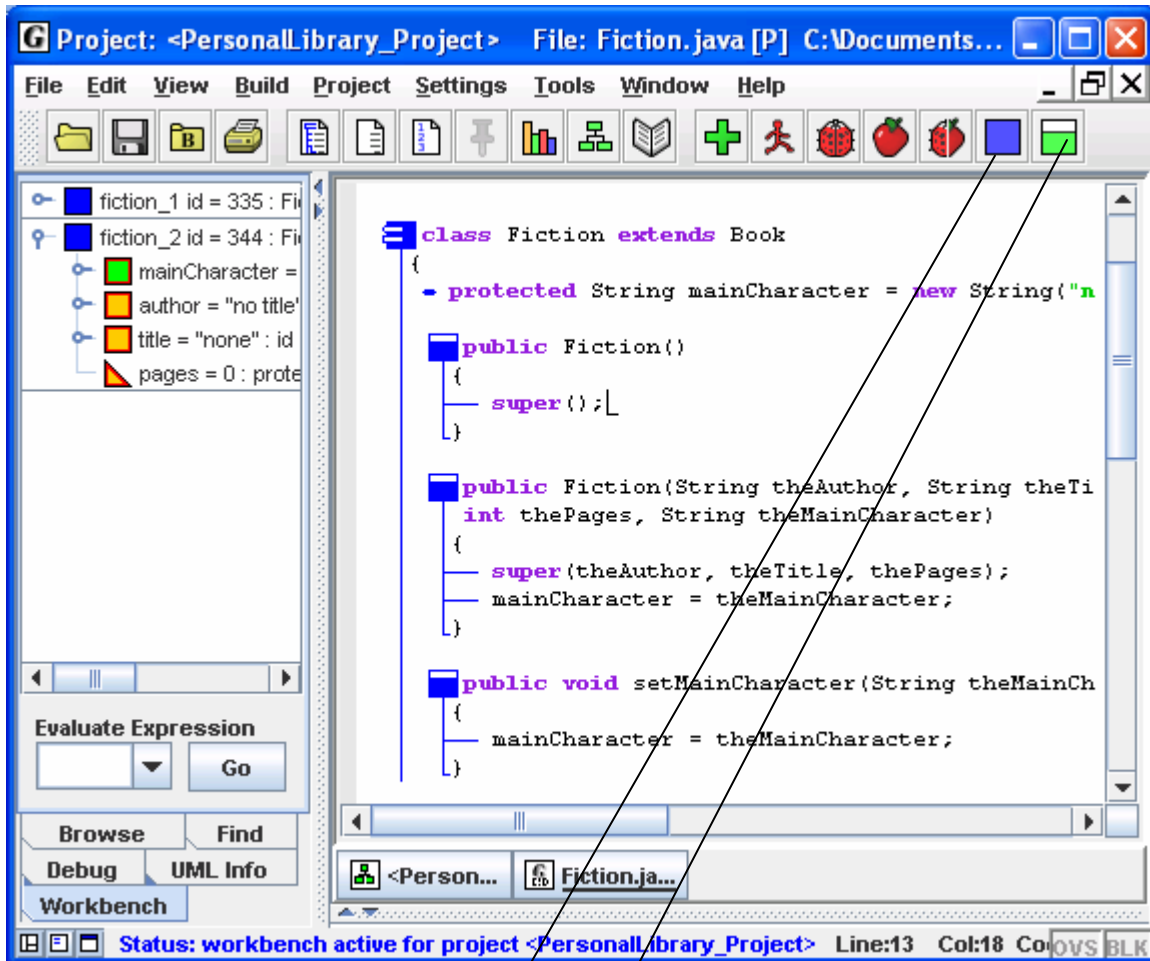





Figure 3-18. Creating an Instance from the CSD Window

Click  to create an instance of the Fiction class.

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).

Click  on a viewer opened for an instance of Fiction (e.g., fiction_2) to invoke a method for the instance.

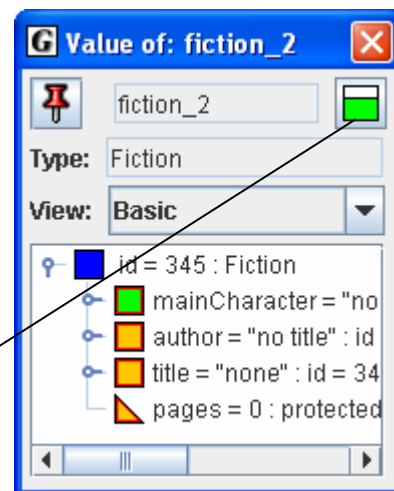


Figure 3-19. Viewer for Fiction instance

3.17 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, enter the name of a class such as `java.lang.String` 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 `String` on the workbench where you can invoke any of its methods as described earlier.

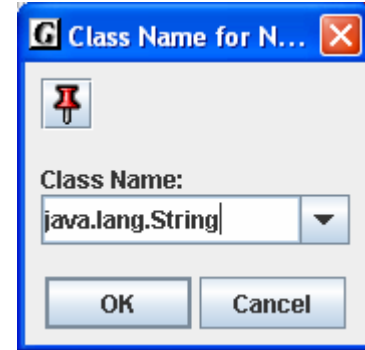


Figure 3-20. Creating an instance of `String`

3.18 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 (Figure 3-21). 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**.

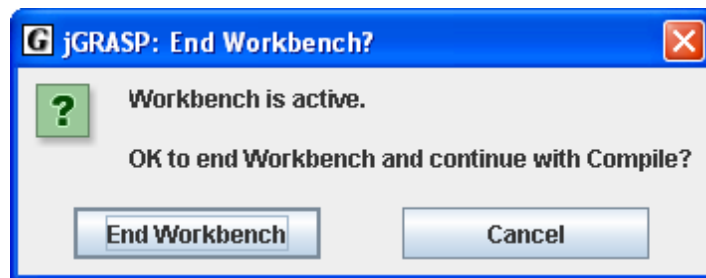


Figure 3-21. Making sure it is okay to exit the Workbench

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.

3.19 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 three 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.

3.20 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 to make sure it is okay to quit jGRASP.



Figure 3-22. Making sure it is okay to exit

3.21 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) Open the CSD window for PersonalLibrary.
 - 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.
- (7) If you have other Java programs available, repeat steps (1), (2), (5), and (6) above for each program.

4 Projects

A project in jGRASP is essentially one or more files which may be located in the same or different directories. When a “project” is created, all information about the project, including project settings and file locations, is stored in a *project file* with the .gpj extension.

Although projects are not required to do simple operations such as Compile and Run, to generate UML class diagrams and to use many of the Object Workbench features, you must organize your Java files in a Project. UML Class Diagrams and the Object Workbench are discussed in Sections 5 and 6. Many users will find projects useful independent of the UML and Object Workbench features.

Before doing this tutorial, be sure you have read the tutorial entitled *Getting Started with Objects* since the concept of a jGRASP project is first introduced there.

Objectives – When you have completed this tutorial, you should be able to create projects, add files to them, remove files from them, generate documentation, and close projects.

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

4.1 Creating a Project

4.2 Adding files to the Project

4.3 Removing files from the Project

4.4 Generating Documentation for the Project (Java only)

4.5 Jar File Creation and Extraction

4.6 Closing a Project

4.7 Exercises

4.1 Creating a Project

On the Desktop menu, click **Project – New – New Standard Project** (Figure 4-1) to open the **New Project** dialog. *Note that the “New J2ME Project” option should only be selected if you have installed the Java Wireless Tool Kit (WTK) and you plan to develop a project based on the Java 2 Micro Edition (J2ME).*

Within the New Project dialog (Figure 4-2), notice the two check boxes (*Add Files Now* and *Open UML Window*). Normally, you would want to have the *Add Files Now* checked ON so that as soon as you click the Create button, the Add Files dialog will pop up. If you are working in Java, you may also want to turn ON the *Open UML Window* option. This will generate the UML class diagram and open the UML window (see Section 5 for details).

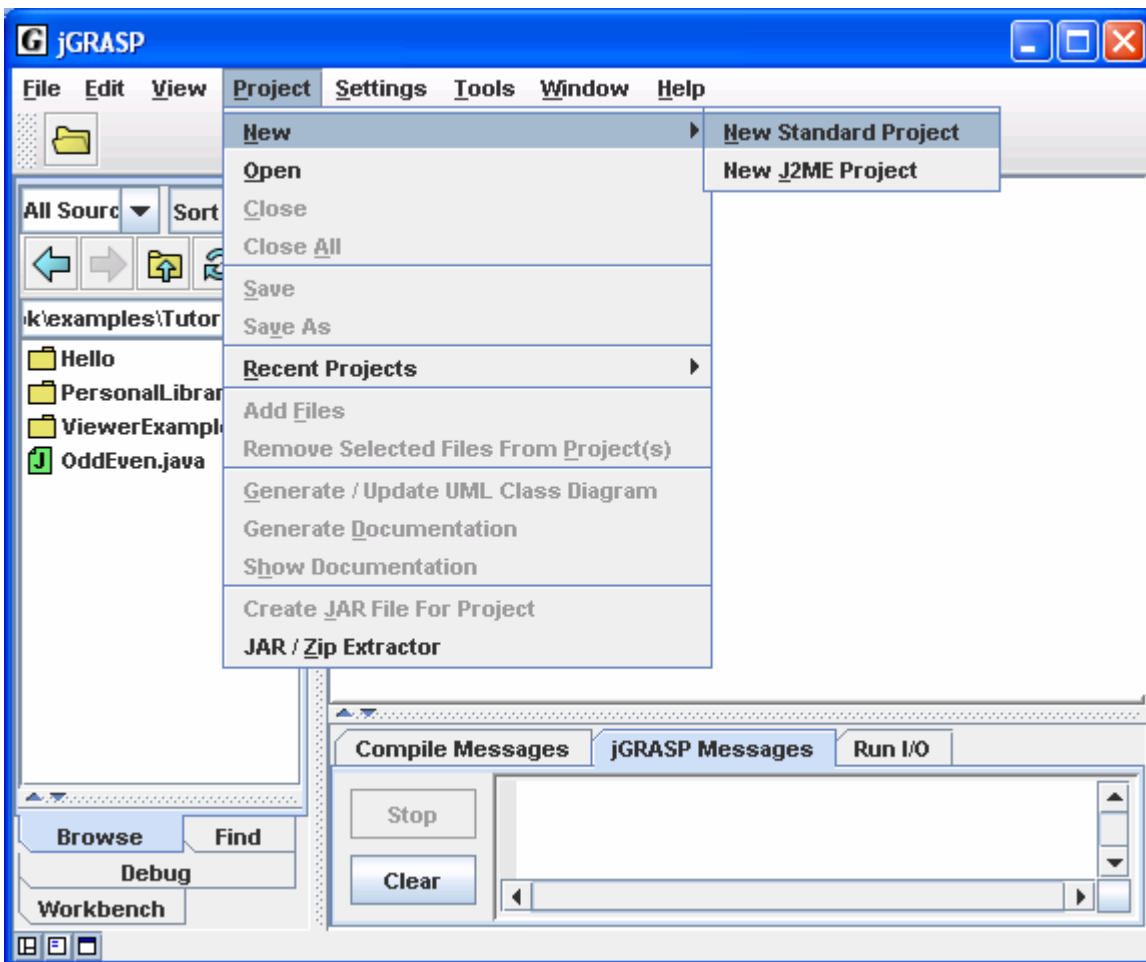


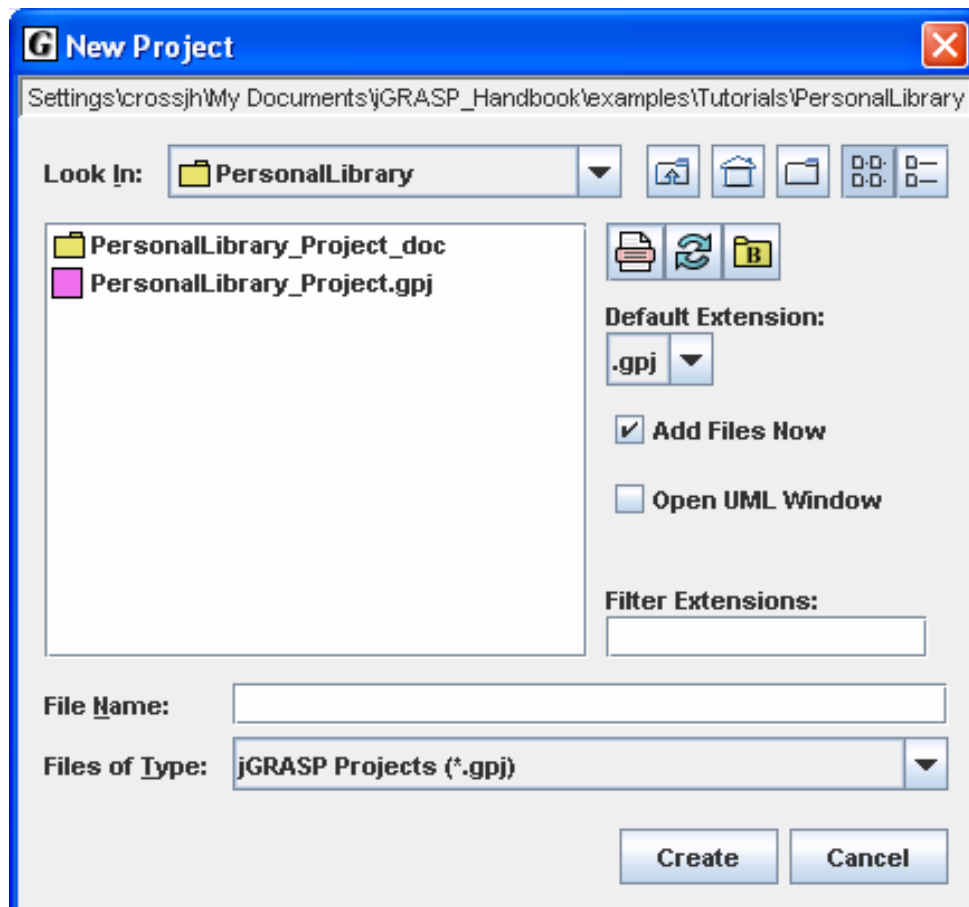
Figure 4-1. Creating a Project

Navigate to the directory where you want the project to reside and enter the project file name. It is recommended that the project file be stored in the same directory as the file containing *main*. A useful naming convention for a project is *ClassnameProject* where *Classname* is the name of the class that contains *main*. For example, since the *PersonalLibrary* class contains *main*, an appropriate name for the project file would be *PersonalLibraryProject*.

After entering the project file name, click **Create** to save the project file. Notice the new project file with *.gpj* extension is listed in the Files section of the Browse tab. The project is also listed in the Open Projects section of Browse tab.

If *Add Files Now* was checked ON when you created the project, the Add Files dialog will pop up. As files are added to the project, they will appear under the project name in the Open Projects section of the Browse tab. When you have finished adding files, click the *Close button* on the dialog. You can always add more files to a project later.

Note that when you have multiple projects open, these are all listed in the Open Projects section of the Browse tab. If you open a UML window for one or more projects and/or if you open one or more CSD windows for files in projects, then the UML or CSD window with focus will determine which open project has focus. The project with focus will have a black square in the project symbol and the project name will be displayed in the title bar of the jGRASP desktop.



4.2 Adding files to the Project

The Browse tab is split to show the current file directory in the top part and the open projects in the lower part as shown in Figure 4-3. After a project has been created and/or opened, there are several ways to add Java files to the project.

- (1) **From Browse Tab** - Drag the file (left click and hold) from the Files section to the project in the Open Projects section below.
- (2) **From Browse Tab** - Drag the file from the Files section to the UML Window.
- (3) **In Browse Tab** - Right click on the file and select **Add to Project**. (Figure 4-3).
- (4) **From CSD window** – Click **Project** – **Add files**.

You can also select multiple files (holding down the control or shift key), and add or drag the highlighted files all at once. The files in the project are shown beneath the project name in the Open Projects section of the Browse tab. Double-clicking on the project name (or single-clicking on the “handle” in front of the project name) will open or close the list of files in the project.

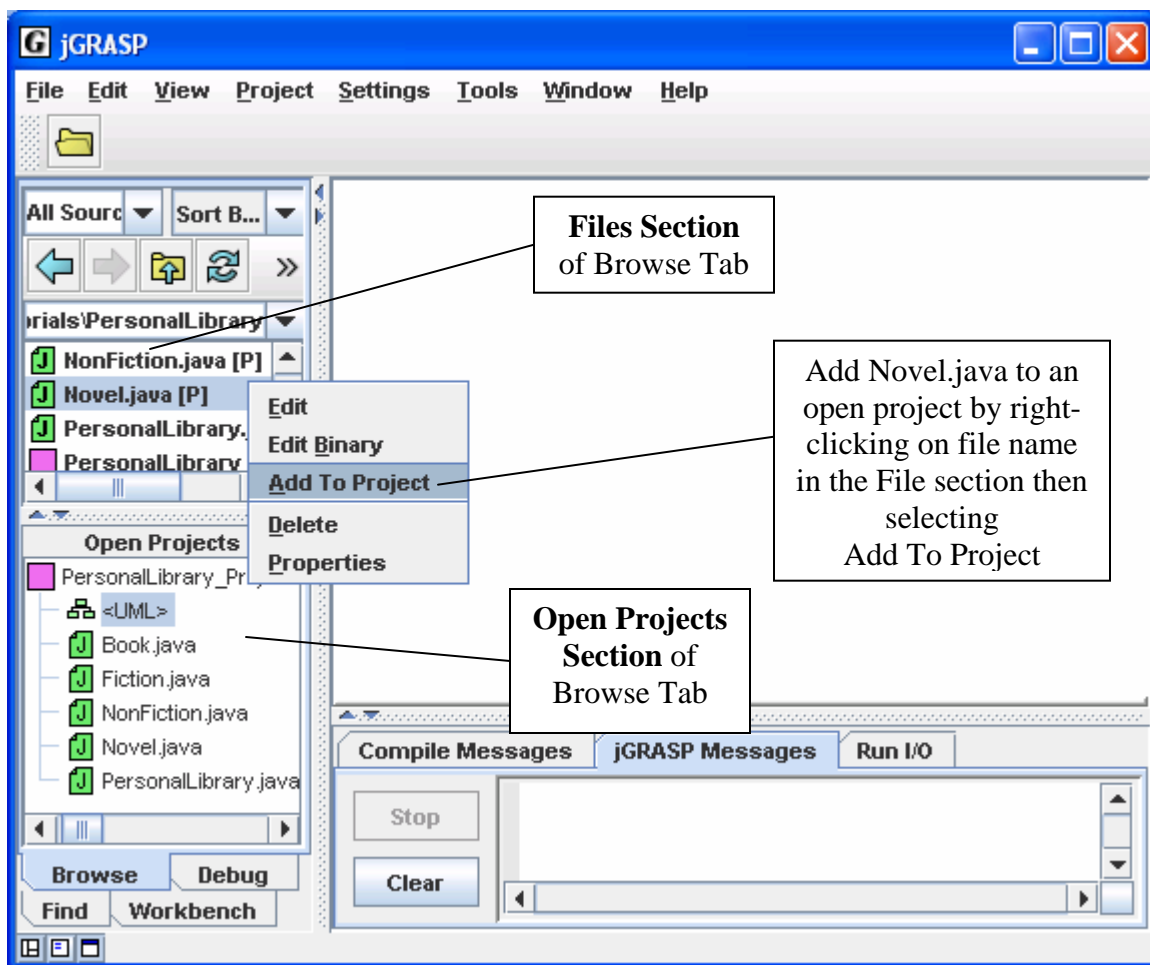


Figure 4-3. Adding a file to the Project

4.3 Removing files from the Project

You can remove files from the project by selecting one or more files in the Open Projects section of the Browse tab, then right clicking and selecting **Remove from Project(s)** as shown in Figure 4-4. You can also remove the selected file(s) by pressing *Delete* on the keyboard. Note that *removing* a file from a project does not delete the file from its directory, only from the project. However, you can delete a file by selecting it in the Files section of the Browse tab, then right-clicking and selecting **Delete** from the pop up menu or by pressing the *Delete* key. Note that deleting a file is a permanent operation, so jGRASP warns you accordingly.

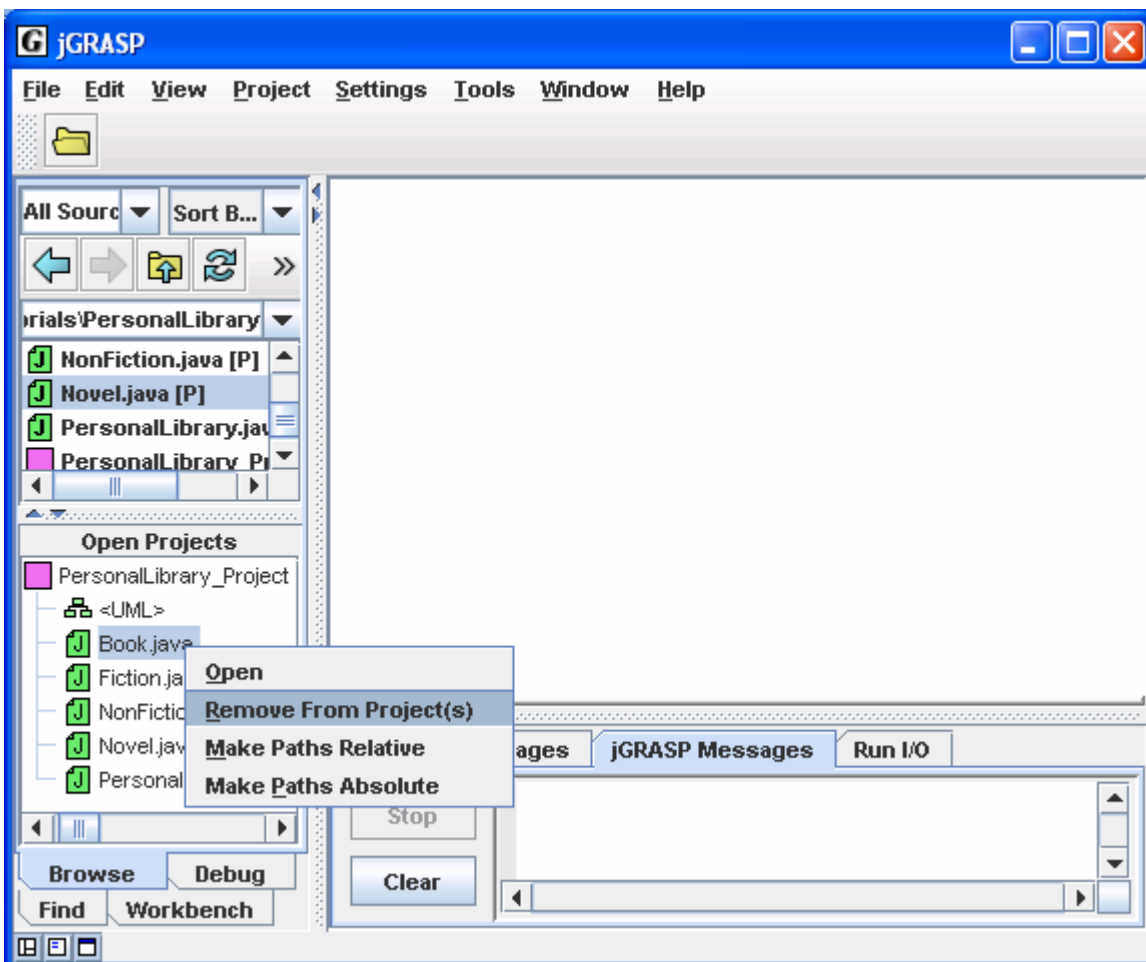


Figure 4-4. Removing a file from the Project

4.4 Generating Documentation for the Project (Java only)

Now that you have established a project, you have the option to generate project level documentation for your Java source code, i.e., an application programmer interface (API). To generate the documentation for the PersonalLibraryProject, select **Project – Generate Documentation – <PersonalLibraryProject>** as shown in the Figure 4-5. This will bring up the “Generate Documentation for Project” dialog which asks for the directory in which the generated HTML files are to be stored. The default directory name is the name of the project with “_doc” appended to it (e.g., PersonalLibraryProject_doc). Using the default name is recommended so that your documentation directories will have a standard naming convention. If the default directory is not indicated, click the **Default** button in the dialog. However, you are free to use any directory as the target. Click the **Generate** button on the dialog to start the process. jGRASP calls the javadoc utility, which is included with the JDK, to create a complete hyper-linked document within a few seconds.

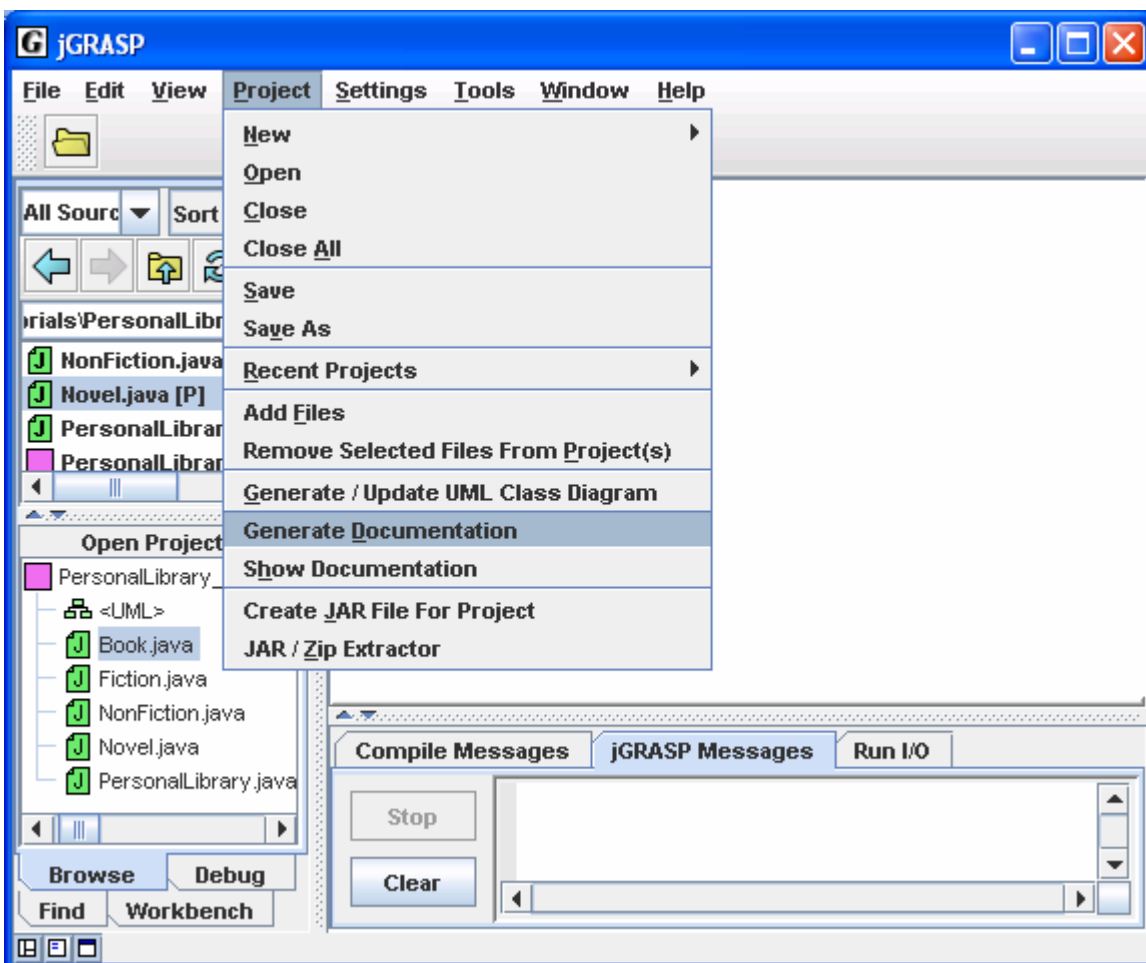


Figure 4-5. Generating Documentation for the Project

The documentation generated for PersonalLibraryProject is shown below in Figure 4-6. Note that in this example, even though no JavaDoc comments were included in the source file, the generated documentation is still quite useful. However, for even better documentation, JavaDoc formal comments should be included in the source code. When generated for a project, the documentation files are stored in a directory that becomes part of the project and, therefore, persists from one jGRASP session to the next. **Project – Show Documentation** can be used to display the documentation without regenerating it. However, if any changes have been made to a project source file and the file has been saved, jGRASP will indicate that the documentation needs to be regenerated. You may choose to view the documentation anyway or to regenerate the documentation.

Documentation generated for an individual file is stored in a temporary directory for the duration of the jGRASP session unless the individual file is part of a project for which documentation has already been generated. In this case, the Generate Documentation displays the existing documentation rather than generating a temporary documentation file.

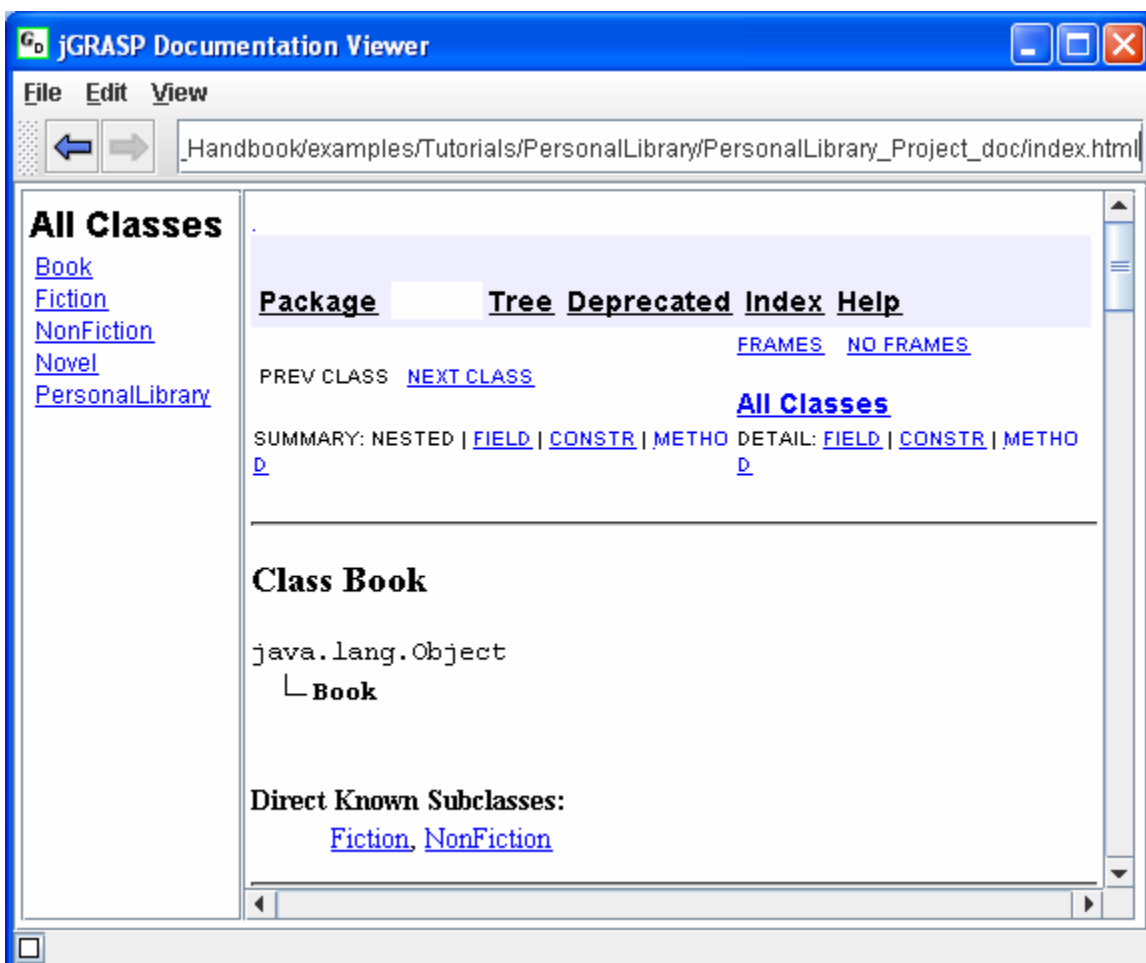


Figure 4-6. Project documentation

4.5 Jar File Creation and Extraction

jGRASP provides a utility for the creation and extraction of a Java Archive file (JAR) for your project. To create a JAR file, click **Project – Create Jar File for Project**. This will allow you to create a single compressed file containing your entire project.

The **Project – Jar/Zip Extractor** option enables you to extract the contents of a JAR or ZIP archive file.

These topics are described in more detail in jGRASP **Help** (find using the Index tab).

4.6 Closing a Project

When you exit jGRASP, the projects and files that are currently open on the desktop are remembered so that the next time you start jGRASP, you can pick up where you left off. However, to prevent clutter you should close the ones you are no longer using.

- (1) From the Desktop toolbar - Click **Project – Close All Projects**.
- (2) From the Desktop toolbar - Click **Project – Active Project < > – Close**.
- (3) From the Browse Tab – Right-click on the project file name in the Open Projects section of the Browse tab and select **Close**.

All project information is saved when you close the project as well as when you exit jGRASP. Note that closing a project does not close the files that are currently open. You can close these individually or all at once with **File – Close All Files**.

4.7 Exercises

- (1) Create a new project called PersonalLibraryProject2 in the same directory folder as the original PersonalLibraryProject. During the create step, add the file Book.java to the new project. Close the Add Files dialog.
- (2) Add the other Java files in the directory to the project by dragging each file from the Files section of the Browse tab and dropping the files in PersonalLibraryProject2 in the open projects section.
- (3) Remove a file from PersonalLibraryProject2. After verifying the file was removed, add it back to the project.
- (4) Generate the documentation for PersonalLibraryProject2. 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.
- (5) Close the project.
- (6) Open the project by double-clicking on the project file in the files section of the Browse tab.

5 UML Class Diagrams

Java programs usually involve multiple classes, and there can be many dependencies among these classes. To fully understand a multiple class program, it is necessary to understand the interclass dependencies. Although this can be done mentally for small programs, it is usually helpful to see these dependencies in a class diagram. jGRASP automatically generates a class diagram based on the Unified Modeling Language (UML). In addition to providing an architectural view of your program, the UML class diagram is also the basis for the Object Workbench which is described in a separate section.

Objectives – When you have completed this tutorial, you should be able to generate the UML class diagram for your project, display the members of a class as well as the dependencies between two classes, and navigate to the associated source code.

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

- 5.1 Opening the Project
- 5.2 Generating the UML
- 5.3 Compiling and Running from the UML Window
- 5.4 Determining the Contents of the UML Class Diagram
- 5.5 Laying Out the UML Diagram
- 5.6 Displaying the Members of a Class
- 5.7 Displaying Dependencies Between Two Classes
- 5.8 Navigating to Source Code via the Info Tab
- 5.9 Finding a Class in the UML Diagram
- 5.10 Opening Source Code from UML
- 5.11 Saving the UML Layout
- 5.12 Printing the UML Diagram

5.1 Opening the Project

The jGRASP project file is used to determine which user classes to include in the UML class diagram. The project should include all of your source files (.java), and you may optionally include other files (e.g., .class, .dat, .txt, etc.). You may create a new project file, then drag and drop files from the Browse tab pane to the UML window.

To generate the UML, jGRASP uses information from both the source (.java) and byte code (.class) files. Recall, .class files are generated when you compile your Java program files. Hence, you must compile your .java files in order to see the dependencies among the classes in the UML diagram. Note that the .class files do not have to be in the project file, but they should be in the same directory as the .java files.

If your project is not currently open, you need to open it by doing one of the following:



- (1) On the Desktop tool bar, click **Project – Open Project**, and then select the project from the list of project files displayed in the Open Project dialog and click the **Open** button.
- (2) Alternatively, in the files section of the Browse tab, double-click the project file.

When opened, the project and its contents appear in the open projects section of the Browse tab, and the project name is displayed at the top of the Desktop. If you need additional help with opening a project, review the previous tutorial on *Projects*.


The remainder of this section assumes you have created your own project file or that you will use PersonalLibraryProject from the examples that are included with jGRASP.

TIP: Remember your Java files must be compiled before you can see the dependencies among your classes in the UML diagram. When you recompile any file in a project, the UML diagram is automatically updated.

5.2 Generating the UML

In Figure 5-1 below, PersonalLibraryProject is shown in the Open Projects section of the Browse tab along with a **UML** symbol  and the list of files in the project. To generate the UML class diagram, double-click the UML symbol . Alternatively, on the Desktop menu, click on **Project – Generate/Update UML Class Diagram**.

The UML window should open with a diagram of all class files in the project as shown below. You can select one or more of the class symbols and drag them around in the diagram. In the figure, the class containing *main* has been dragged to the upper left of the diagram and the legend has been dragged to the lower center.

The UML window is divided into three panes. The top pane contains a **panning rectangle** that allows you to reposition the entire UML diagram by dragging the panning rectangle around. To the right of the panning rectangle are the buttons for scaling the UML: divide by 2 (/2), divide by 1.2 (/1.2), no scaling (1), multiply by 1.2 (*1.2), and multiply by 2 (*2). In general, the class diagram is automatically updated as required; however, the user can force an update by clicking the Update UML diagram button  on the desktop menu.

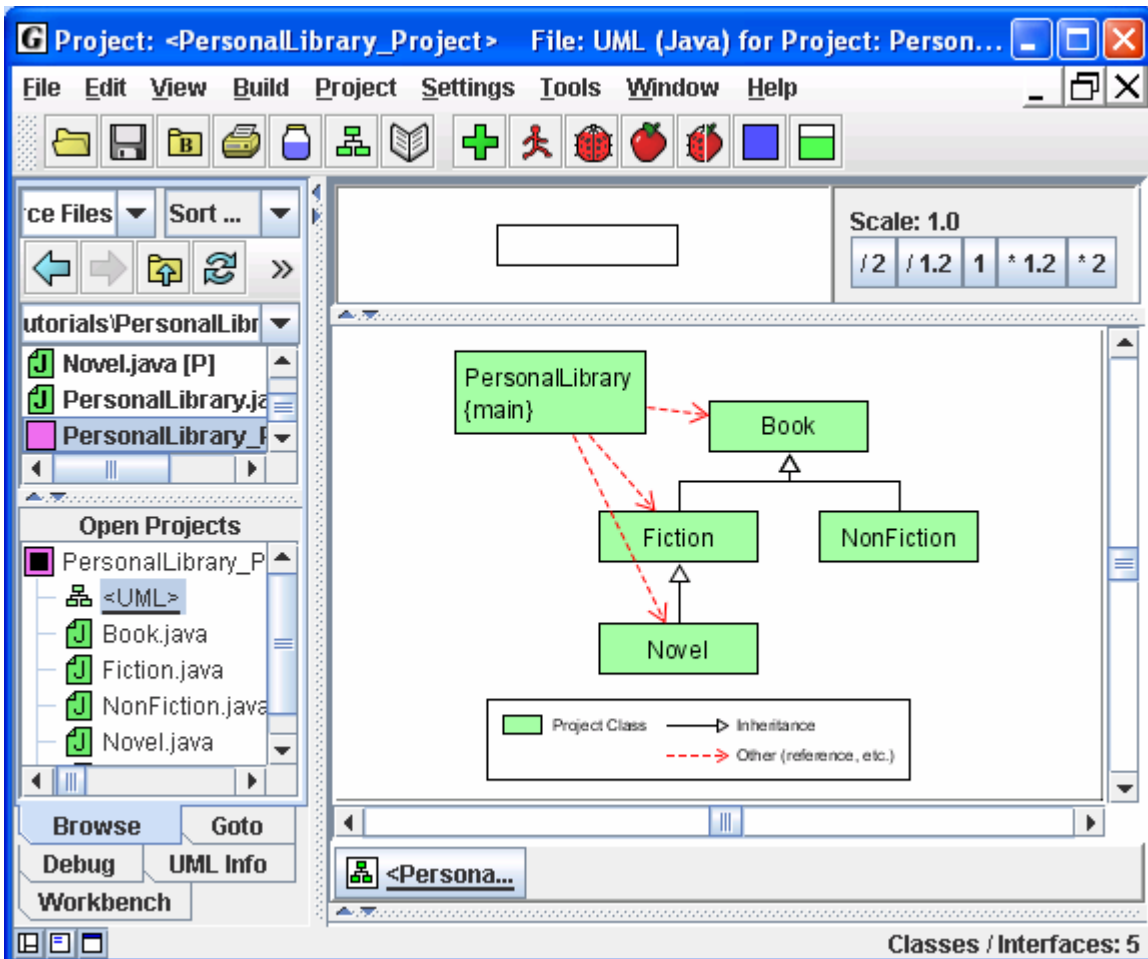







Figure 5-1. Generating the UML

If your project includes class inheritance hierarchies and/or other dependencies as in the example, then you should see the appropriate red dashed and solid black dependency lines. The meaning of these lines is annotated in the *legend* as appropriate.

5.3 Compiling and Running from the UML Window

The Build menu and buttons on the toolbar for the UML window are essentially the same as the ones for the CSD window. For example, clicking the Compile button  compiles all classes in the project (Figure 5-2). When a class needs to be recompiled due to edits, the class symbol in the UML diagram is marked with red crosshatches (double diagonal lines). During compilation, the files are marked and then unmarked when done. Single red diagonal lines in a class symbol indicate that another class upon which the first class depends has been modified. Clicking the **Run** button  on the toolbar will launch the program as an application assuming there is a *main()* method in one of the classes. Clicking on the **Run as Applet** button  as an applet will launch the program as an applet assuming one of the classes is an applet. Similarly, clicking the **Debug** button  or the **Debug Applet** button  will launch the program in debug mode. Note that for running in debug mode, you should have a breakpoint set somewhere in the program so that it will stop.

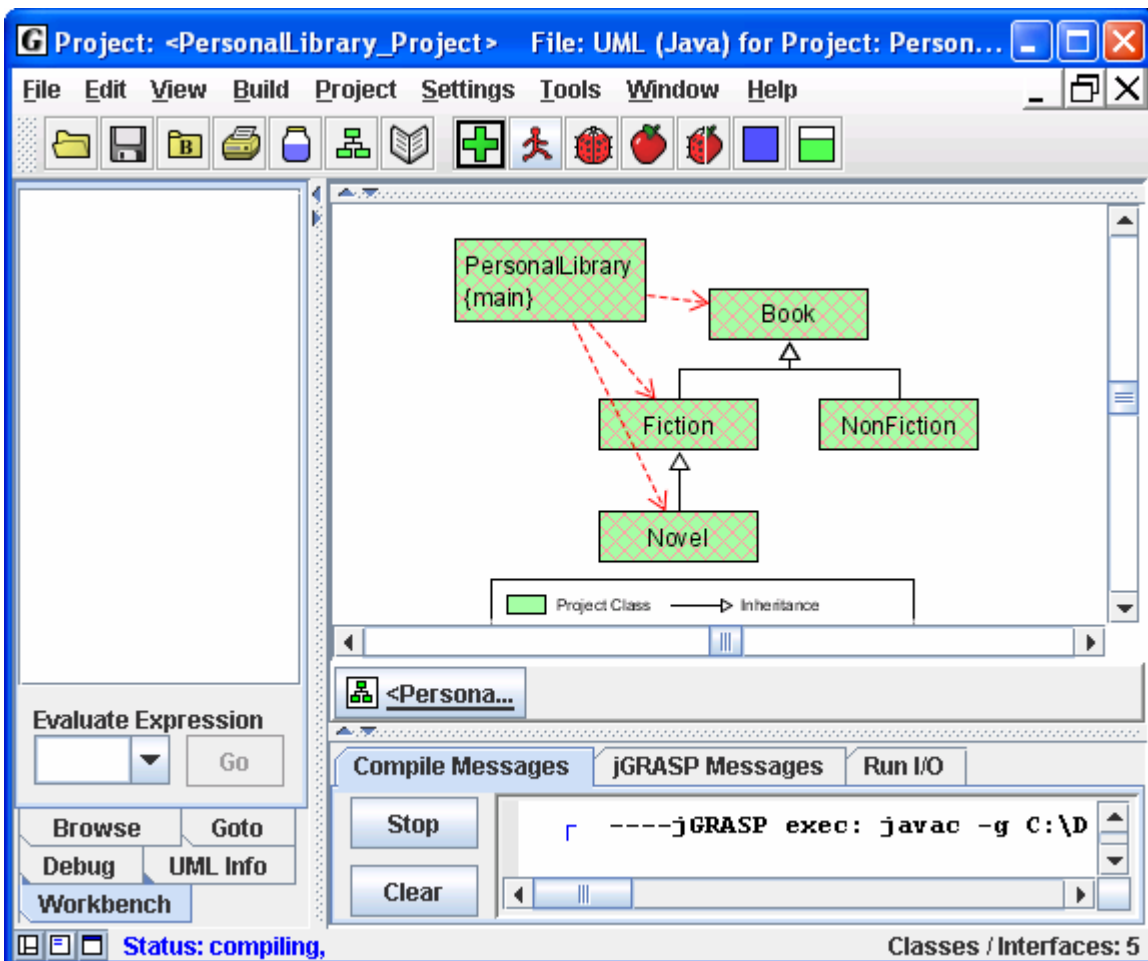


Figure 5-2. Compiling Your Program

5.4 Determining the Contents of the UML Class Diagram

jGRASP provides one option to control the contents of your UML diagram, and another option to determine which elements in the diagram are actually displayed. **Settings – UML Generation Settings** allows you to control the contents of the diagram by excluding certain categories of classes (e.g., external superclasses, external interfaces, and all other external references). **The View menu** allows you to make visible (or hide) certain categories of classes and dependencies that are actually in the UML diagram. Both options are described below.

Most programs depend on one or more JDK classes. Suppose you want to include these JDK classes in your UML diagram (the default is to exclude them). Then you will need to change the UML generation settings in order to not exclude these items from the diagram. Also, if you do not see the red and black dependency lines expected, then you may need to change the View settings. These are described below.

Excluding (or not) items from the diagram - On the UML window menu, click on **Settings – UML Generation Settings**, which will bring up the **UML Settings** dialog. Generally you should leave the top three items unchecked so that they are not excluded from the UML diagram. Now for our example of not excluding the JDK classes, under **Exclude by Type of Class**, uncheck (turn OFF) the checkbox that excludes **JDK Classes**, as shown in Figure 5-3. Note, synthetic classes are created by the Java compiler and are usually not included in the UML diagram. After checking (or unchecking) the items so that your dialog looks like the one in the figure, click the OK button. This should close the dialog and update the UML diagram. All JDK classes used by the project classes should now be visible in the diagram as gray boxes. This is shown in Figure 5-4 after the JDK classes have been dragged around. To remove them from the diagram, you will need to turn on the exclude option. If you want to leave them in the diagram but not always display them, see the next paragraph. For more information see UML Settings in jGRASP **Help**.

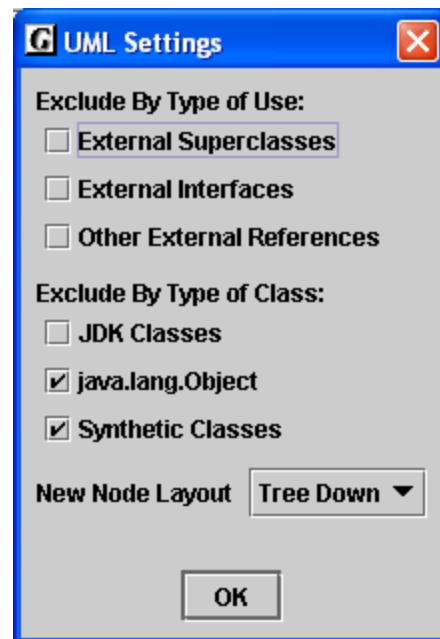


Figure 5-3. Changing the UML Settings

Making objects in the diagram visible (or not) - On the UML window menu, click on **View – Visible Objects**, then check or uncheck the items on the list as appropriate. In general, you will want all of the items on the list in **View – Visible Objects** checked ON as shown in Figure 5-4. For example, for the JDK classes and/or other classes outside the project to be visible, **External References** must be checked ON. Clicking (checking) ON or OFF any of the items on the Visible Objects list simply displays them or not, and their previous layout is retained when they are redisplayed. Note that if items have been *excluded* from the diagram via **Settings – UML Generation Settings**, as described above, then making them visible will have no effect since they are not part of the diagram. For more information see View Menu in jGRASP **Help**.

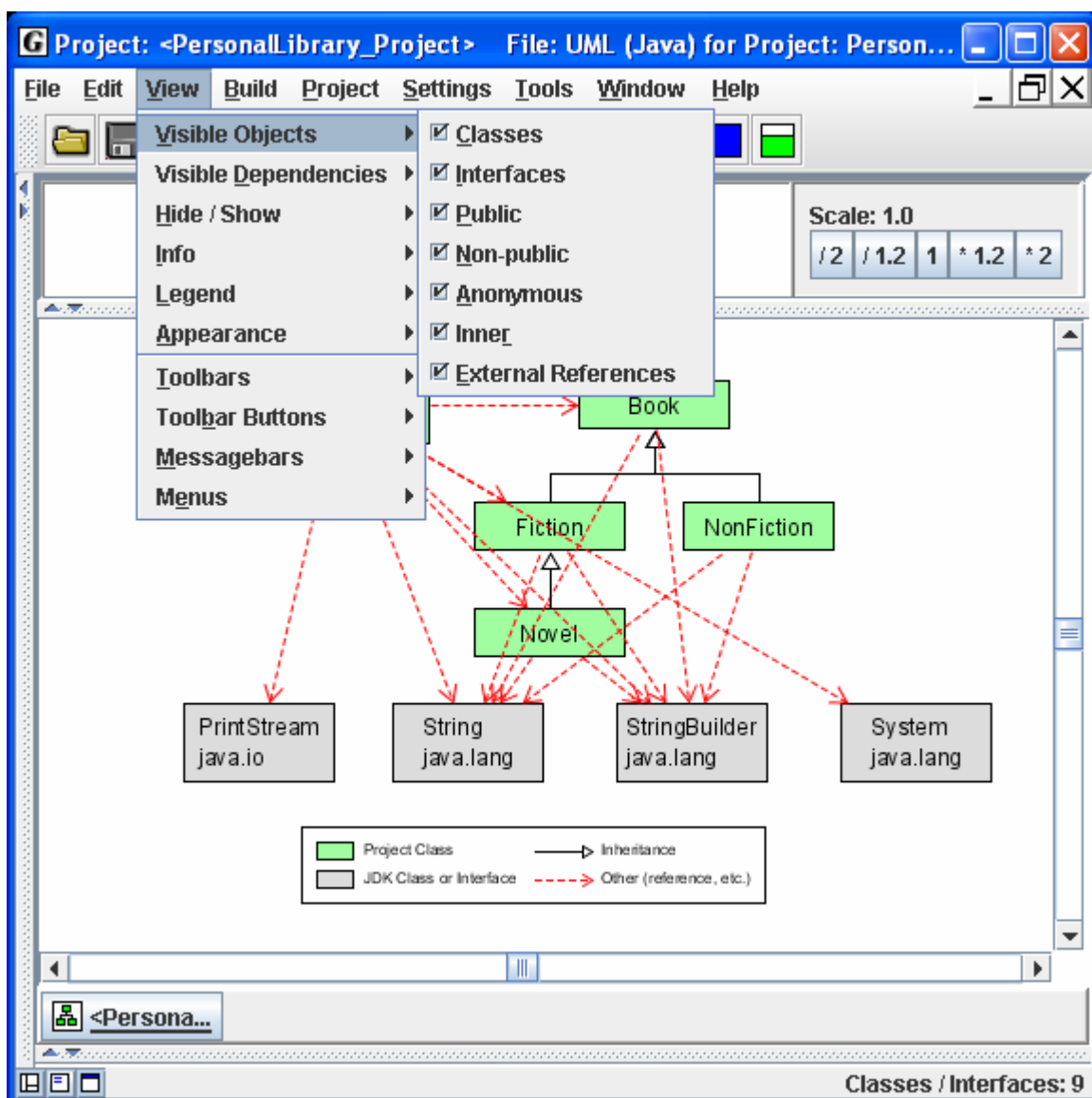


Figure 5-4. Making objects visible

Making dependencies visible - On the UML window menu, click on **View – Visible Dependencies**, then check or uncheck the items on the list as appropriate. The only two categories of dependencies in the example project are **Inheritance** and **Other**. **Inheritance dependencies** are indicated by black lines with closed arrowheads that point from a child to a parent to form an *is-a* relationship. Red dashed lines with open arrowheads indicate **other dependencies**. These include *has-a* relationships that indicate a class includes one or more instances of another class. Also a class may simply reference an instance variable or method of another class. The red dashed arrow is drawn from the class where an object is declared or referenced to the class where the item is actually defined. In general, you probably want to make all dependencies visible. as indicated in Figure 5-5.

Displaying the Legend - The legend has been visible in each of the UML diagrams (figures) in this tutorial. To set the options for displaying the legend, click **View –**

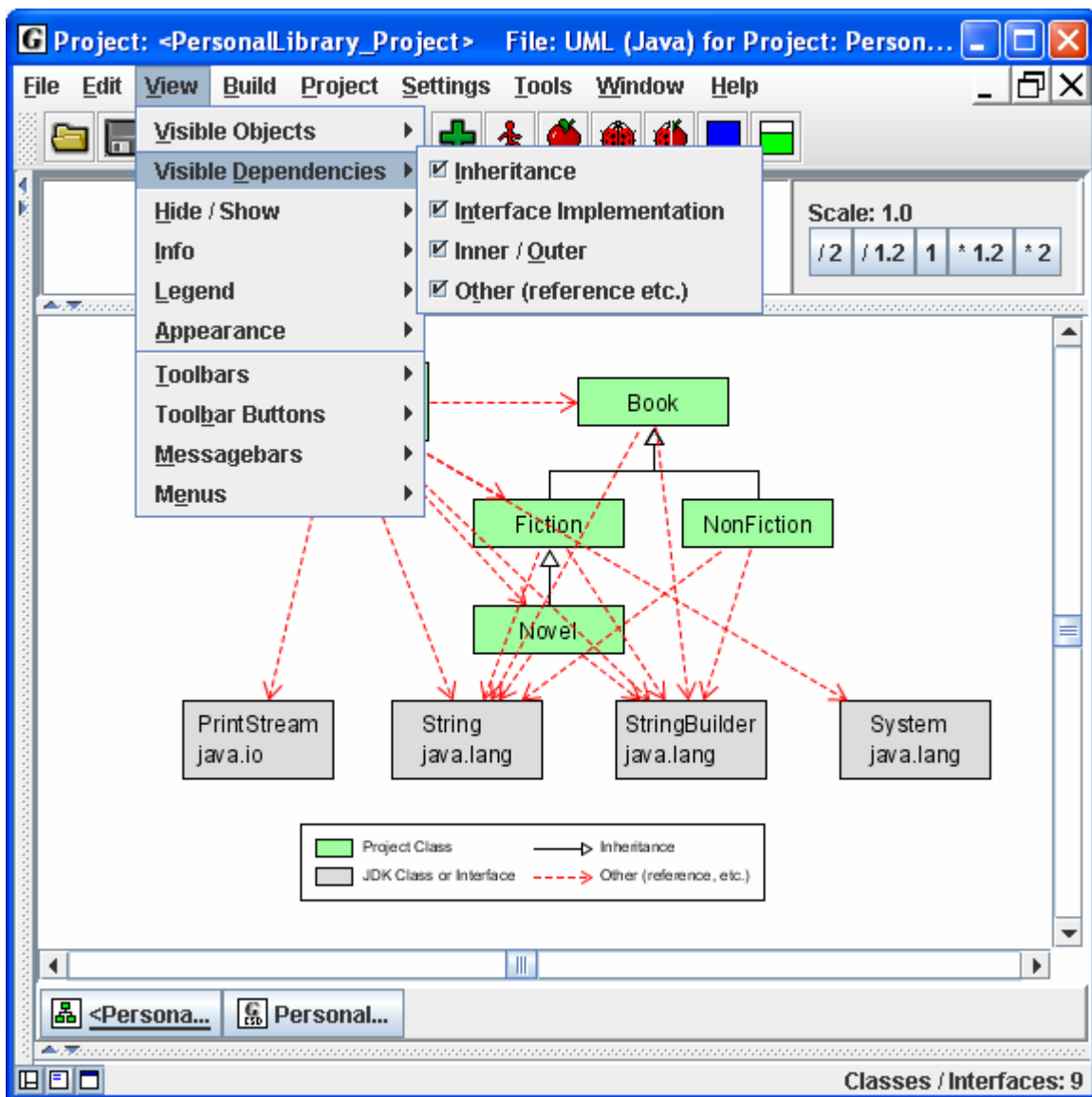


Figure 5-5. Making dependencies visible

Legend. Typically, you will want the following options checked ON: Show Legend, Visible Items Only, and Small Font. Notice that if “Visible Items Only” is checked ON, then an entry for JDK classes appears in the legend only if JDK classes are visible in the UML diagram. Experiment by turning on/off the options in **View – Legend**. When you initially generate your UML diagram, you may have to pan around it to locate the legend. Scaling the UML down (e.g., dividing by 2) may help. Once you locate it, just select it and drag to the location where you want it as described in the next section.

5.5 Laying Out the UML Diagram

Currently, jGRASP has limited automatic layout capabilities. However, manually arranging the class symbols in the diagram is straightforward, and once this is done, jGRASP remembers your layout from one generate/update to the next.

To begin, locate the class symbol that contains *main*. In our example, this would be the `PersonalLibrary` class. Remember the project name should reflect the name of this class. Generally, you want this class near the top of the diagram. Left click on the class symbol and then, while holding down the left mouse button, drag the symbol to the area of the diagram you want it, and then release the mouse button. Now repeat this for the other class symbols until you have the diagram looking like you want it. Keep in mind that class–subclass relationships are indicated by the *inheritance arrow* and that these should be laid out in a tree-down fashion. You can do this automatically by selecting all classes for a particular class–subclass hierarchy (hold down SHIFT and left-click each class). Then click **Edit – Layout – Tree Down** to perform the operation; alternatively, you can right-click on a selected class or group of classes, then on the pop up menu select **Layout – Tree Down**. Finally, right-clicking in the background of the UML window with no classes selected will allow you to lay out the entire diagram.

With a one or more classes selected, you can move them as a group. Figure 5-5 shows the UML diagram after the `PersonalLibrary` class has been repositioned to the top left and the JDK classes have been dragged as a group to the lower part of the diagram. You can experiment with making these external classes visible by going to **View – Visible Objects** – then uncheck **External References**.

Here are several heuristics for laying out your UML diagrams:

- (1) The class symbol that contains *main* should go near the top of the diagram.
- (2) Classes in an inheritance hierarchy should be laid out *tree-down*, and then moved as group.
- (3) Other dependencies should be laid out with the red dashed line pointing downward.
- (4) JDK classes, when included, should be toward the bottom of the diagram.
- (5) Line crossings should be minimized.
- (6) The legend is usually below the diagram.

5.6 Displaying the Members of a Class

To display the fields, constructors, and methods of a class, right-click on the class, then select **Show Class Info** which will pop the UML Info tab to the top in the left tab pane. Also, in the left tab pane, you can click on the **UML Info** tab to pop it to the top. Once the Info tab is on top, each time you select a class its members will be displayed.

In Figure 5-6, class Fiction has been selected and its fields, constructors, and methods are displayed in the left pane. This information is only available when the source code for a class is in the project. In the example below, the System class from package java.lang is an external class, so selecting it would result in a “no data” message. If the only field you are seeing is mainCharacter, click **View – Info – Show Inheritance within Project**. You should now see the fields that are inherited by Fiction (i.e., author, pages, and title).

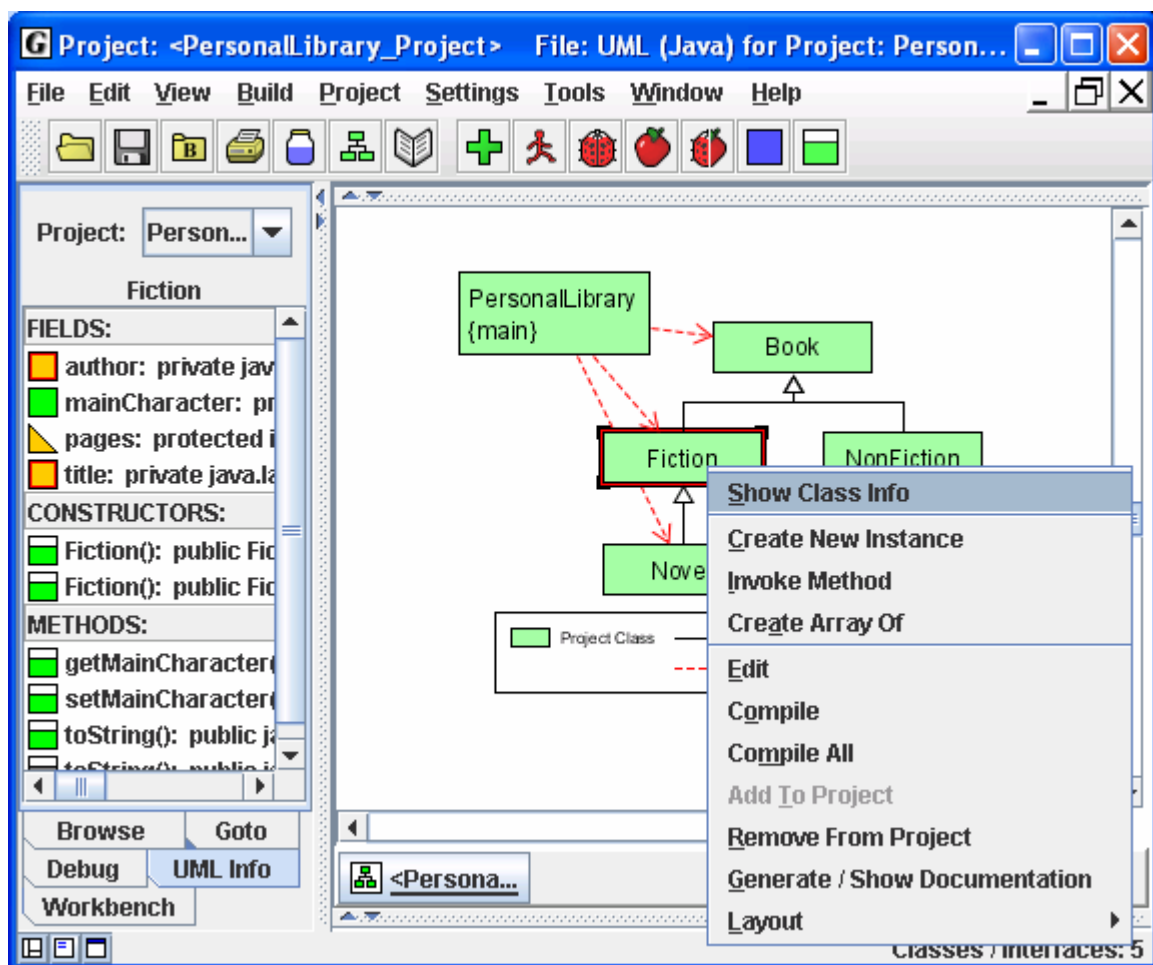


Figure 5-6. Displaying class members

5.7 Displaying Dependencies Between Two Classes

Let's reduce the number of classes in our UML diagram by not displaying the JDK classes. Click **View – Visible Objects** and uncheck **External References**. Now to display the dependencies between two classes, right-click on the arrow, then select **Show Dependency Info**. You can also click on the UML Info tab to pop it to the top. Once the Info tab is on top, each time you select an arrow, the associated dependencies will be displayed.

In Figure 5-7, the edge drawn from PersonalLibrary to Fiction has been selected as indicated by the large arrowhead. The list of dependencies in the Info tab includes one constructor (Fiction()) and one method (getMainCharacter()). These are the resources that PersonalLibrary uses from Fiction. Understanding the dependencies among the classes in your program should provide you with a more in-depth comprehension of the source code. Note that clicking on the arrow between PersonalLibrary and the PrintStream class in Figure 5-6 would show that PersonalLibrary is using two println() methods from the PrintStream class. Make the **External References** visible again and try this.

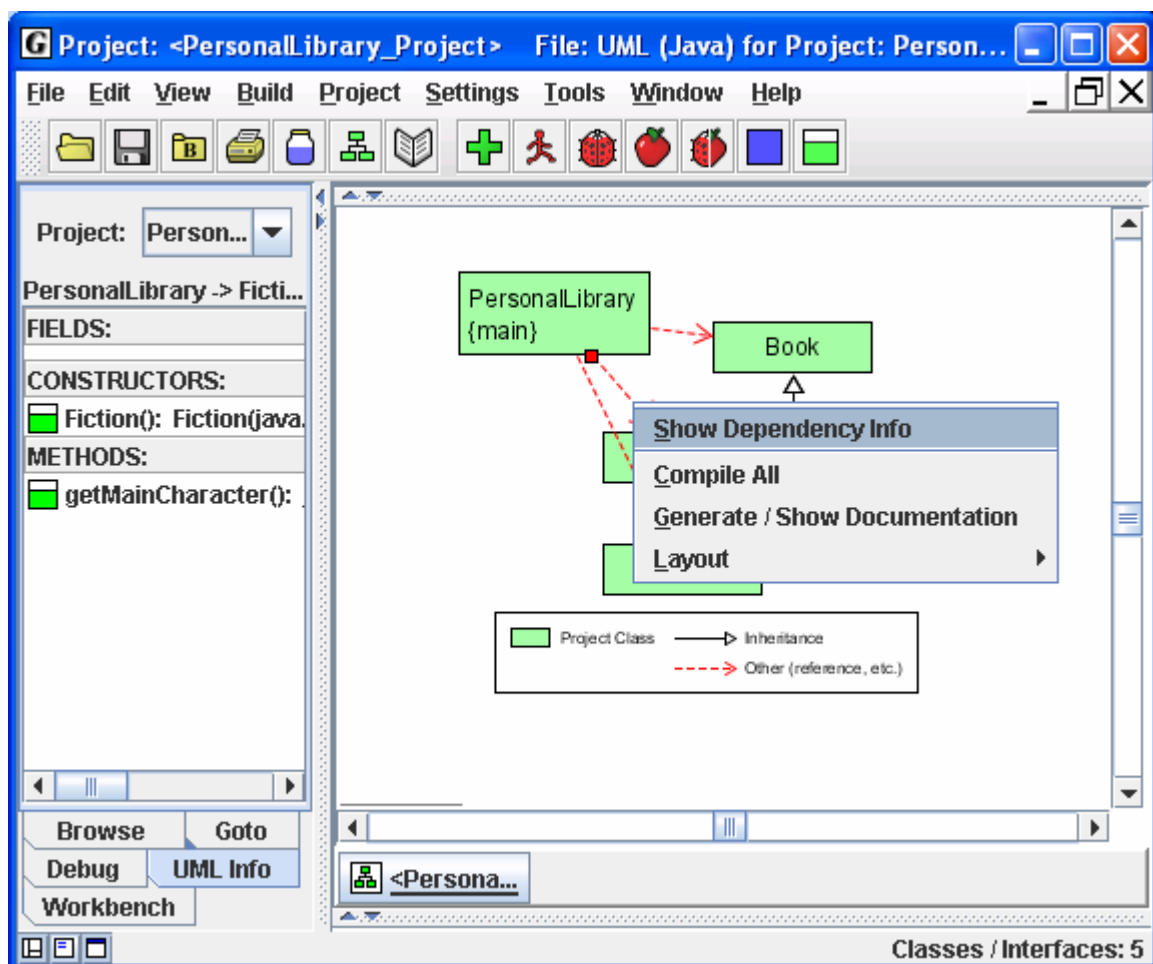


Figure 5-7. Displaying the dependencies between two classes

5.8 Navigating to Source Code via the Info Tab

In the Info tab, a green symbol indicates the item is defined or used in the class rather than inherited from a parent class. Double-clicking on a green item will take you to its definition or use in the source code. For example, clicking on `getMainCharacer()` in Figure 5.7 above will open a CSD window for `PersonalLibrary` with the line containing `getMainCharacter()` highlighted as shown in Figure 5-8 below.

5.9 Finding a Class in the UML Diagram

Since a UML diagram can contain many classes, it may be difficult to locate a particular class. In fact, the class may be off the screen. The **Goto** tab in the left pane provides the list of classes in the project. Clicking on a class in the list brings it to the center of the UML window.

5.10 Opening Source Code from UML

The UML diagram provides a convenient way to open source code files. Simply double-click on a class symbol, and the source code for the class is opened in a CSD window.

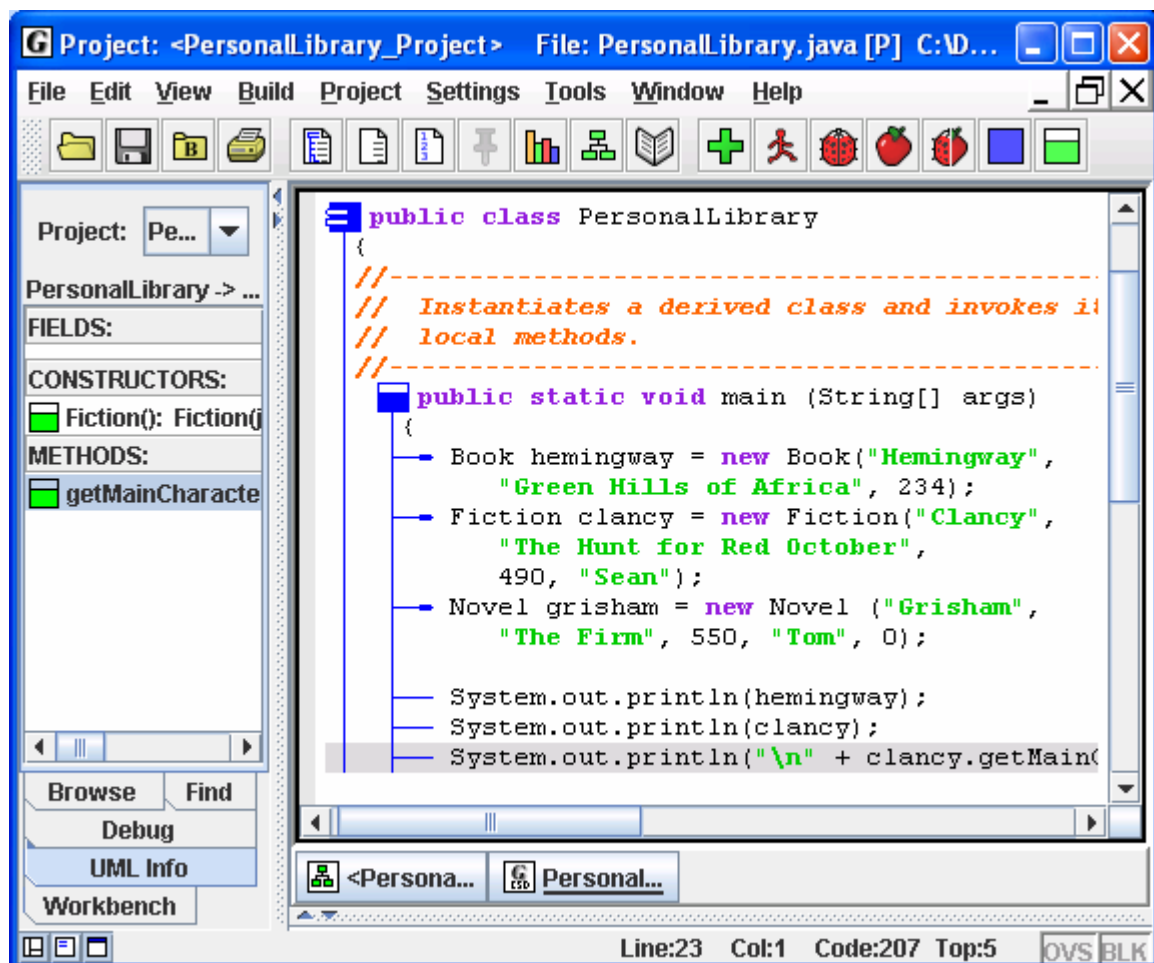


Figure 5-8. Navigating to where `getMainCharacer` is used in the CSD Window

5.11 Saving the UML Layout

When you close a project, change to another project, or simply exit jGRASP, your UML layout is automatically saved in the project file (.gpj). The next time you start jGRASP, open the project, and open the UML window, you should find your layout intact.

If the project file is created in the same directory as your program files (.java and .class files), and if you added the source files with *relative paths*, then you should be able to move, copy, or send the project and program files as a group (e.g., email them to your instructor) without losing any of your layout.

5.12 Printing the UML Diagram

With a UML window open, click on **File – UML Print Preview** to see how your diagram will look on the printed page. If okay, click the **Print** button in the lower left corner of the Print Preview window. Otherwise, if the diagram is too small or too large, you may want to go back and scale it using the scale factors near the top right of the UML window, and then preview it again.

For details see UML Class Diagrams in jGRASP **Help**.

6 The Object Workbench


The Object Workbench, which is tightly integrated with the CSD and UML windows, provides a useful approach for learning the fundamental concepts of classes and objects. The user can create instances of any class in the CSD window, the UML window, or the Java class libraries. When an object is created, it appears on the workbench where the user can select it and invoke any of its methods. The user can also invoke *static* (or class) methods directly from the class without creating an instance of the class. 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. That is, being able to invoke the methods without the need for a driver program. Some of the examples in this section were also presented in the section on Getting Started with Objects; however, more detail is included in this section.


Objectives – When you have completed this tutorial, you should be able to create objects for the workbench from classes in CSD or UML windows as well as directly from the Java libraries, invoke the methods for each of these objects, and display the dynamic states of these objects by opening object viewers for them.


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

- 6.1 Invoking Static Methods from the CSD Window
- 6.2 Invoking Static Methods from the UML Window
- 6.3 Creating an Object for the Workbench
- 6.4 Invoking a Method
- 6.5 Invoking Methods with Parameters Which Are Objects
- 6.6 Invoking Methods on Object Fields
- 6.7 Selecting Categories of Methods to Invoke
- 6.8 Opening Object Viewers
- 6.9 Running the Debugger on Invoked Methods
- 6.10 Exiting the Workbench

6.1 Invoking Static Methods from the CSD Window

In the tutorial *Getting Started*, we ran the Hello program as an application by clicking the Run button . Now let's see how we can invoke its *main* method directly by using the workbench. Since *main* is a static method, it is associated with the Hello class rather than an instance of the Hello class; therefore, we don't have to create an instance for the workbench. There are two ways to invoke a static method from the CSD window:

- a. Click **Build – Java Workbench – Invoke Static Method**.
- b. Click the Invoke Static Method button  on the toolbar.

The latter is the easiest way, so click the Invoke Static Method  button now. This pops up the Invoke Method dialog which lists the static method *main*. After selecting *main*, the dialog expands to show the available parameters (Figure 6-2). We can leave the *java.lang.String[] args* blank since our *main* method is not expecting command line arguments to be passed into it.

In Figure 6-2, notice the two check boxes below the String[] args field. The first, *Don't Show Result Dialog*, will be useful when you want to repeatedly invoke a method that has a void return type or one that you do not care about. When checked ON, all result dialogs (e.g., Figure 6-3) will be suppressed. The second check box, *Run Without Clearing the Workbench*, is a special case option for running a *main*. Normally it is okay to invoke a *main* method without clearing the workbench if you are sure this won't interfere with objects you previously placed on the workbench.

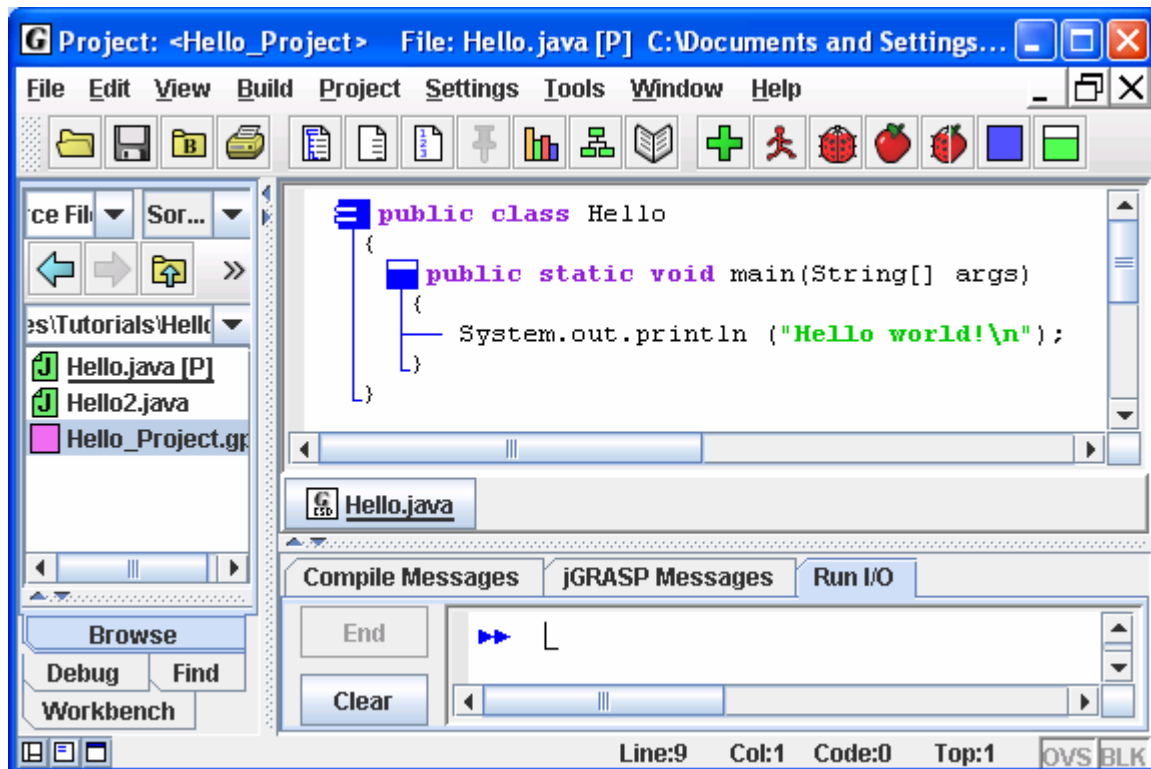



Figure 6-1. Invoking a static method from the Workbench

Finally, notice the “stick-pin”  in the upper left corner which is used to keep the dialog open until you close it. This will allow you to click the Invoke button multiple times.

Now you are ready to invoke the main method by clicking the **Invoke** button in the lower left corner of the dialog. Figure 6-3 shows the desktop and the dialog that pops up with the result: “Method invocation successful (void return type).” Recall that *main* has a “void” return type. The standard output from the program, “Hello World!” appears in the Run I/O tab pane. When the return type for a method is not void, the dialog in Figure 6-3 will contain the value of the return type.

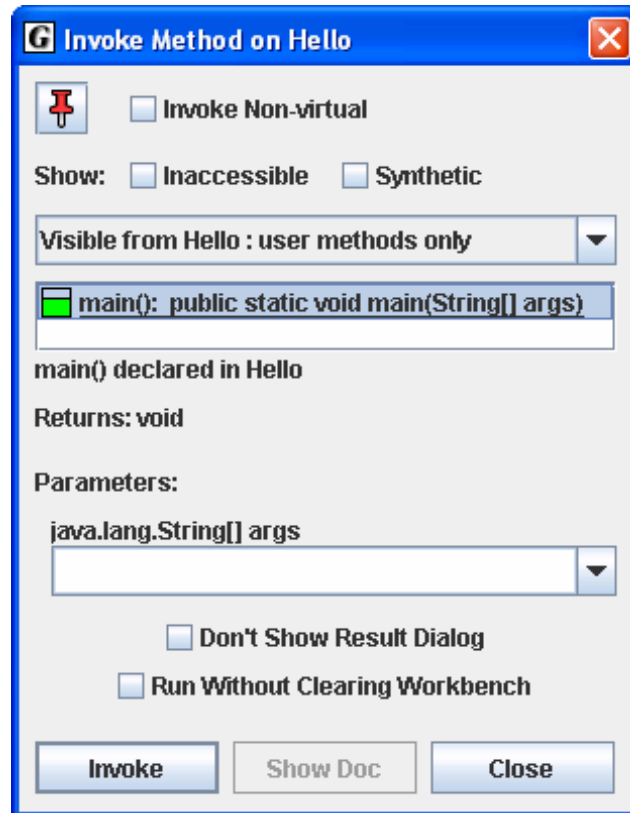


Figure 6-2. Invoking *main*

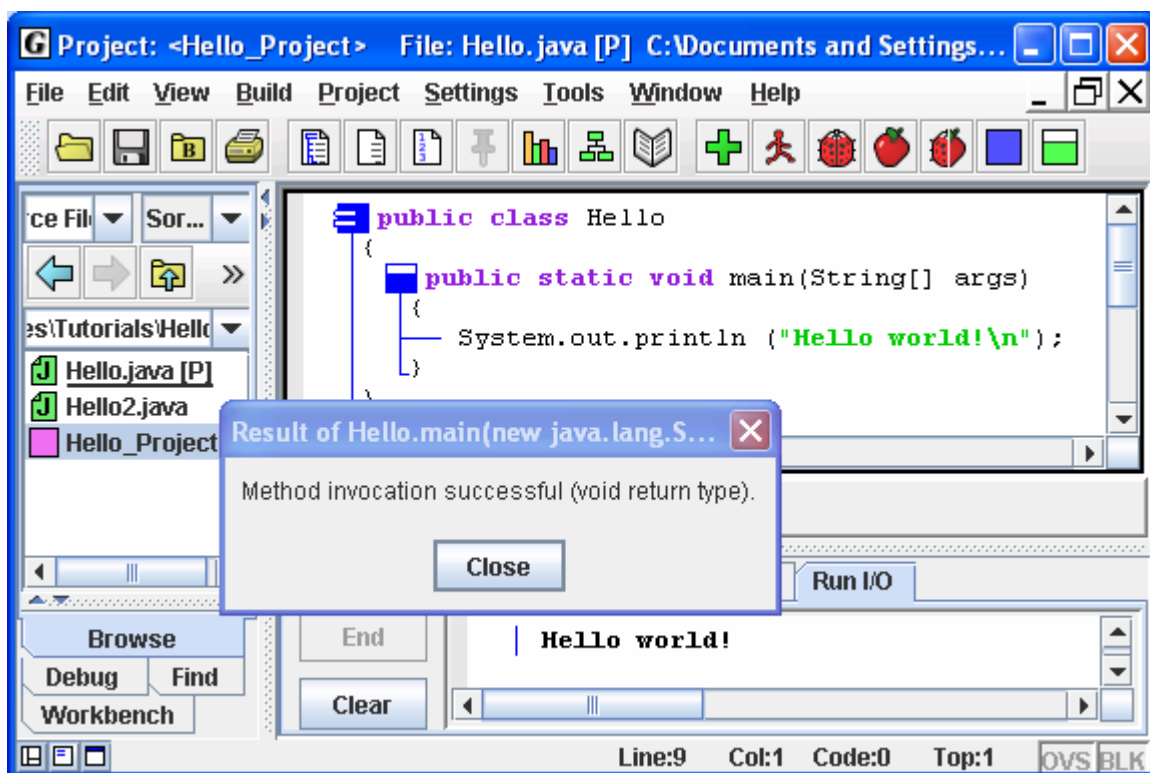



Figure 6-3. The Result dialog from invoking a method

6.2 Invoking Static Methods from the UML Window

Figure 6-4 shows that we have created a project file, Hello_Project, added Hello.java to the project, and then generated the UML class diagram. To make the class diagram more interesting, we have elected to display the Java library classes used by the Hello class. We did this by selecting **Settings – UML Generation Settings** – then in the dialog, we unchecked **JDK classes** under the **Exclude by Type** section. As always, feel free to substitute your own examples in the discussion below.

Since *main* is a static method associated with the class rather than an instance of the class, it can be invoked by selecting the Hello class in the UML diagram, then right-clicking and selecting **Invoke Method**. This pops up the Invoke Method dialog which lists the static method *main* as described in the section above. After selecting *main*, leave the parameters blank, and then click the **Invoke** button. The “Result” dialog should pop up and you should see the output “Hello World!” in the Run I/O tab as shown in Figure 6-5.

You can also invoke the static methods of a class in the UML window by using the Workbench menu or by clicking the Invoke Static method button  on the toolbar.

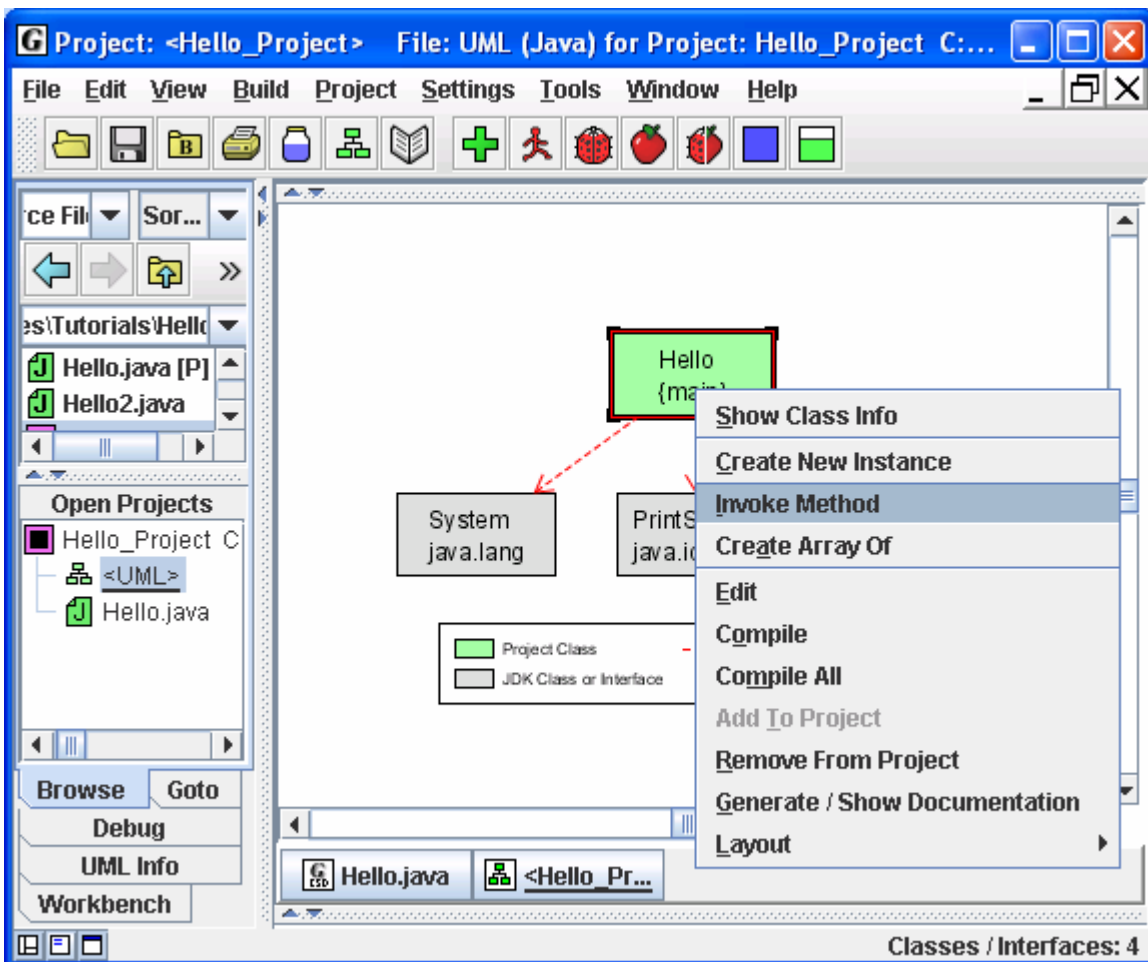


Figure 6-4. Invoking a static method from a class

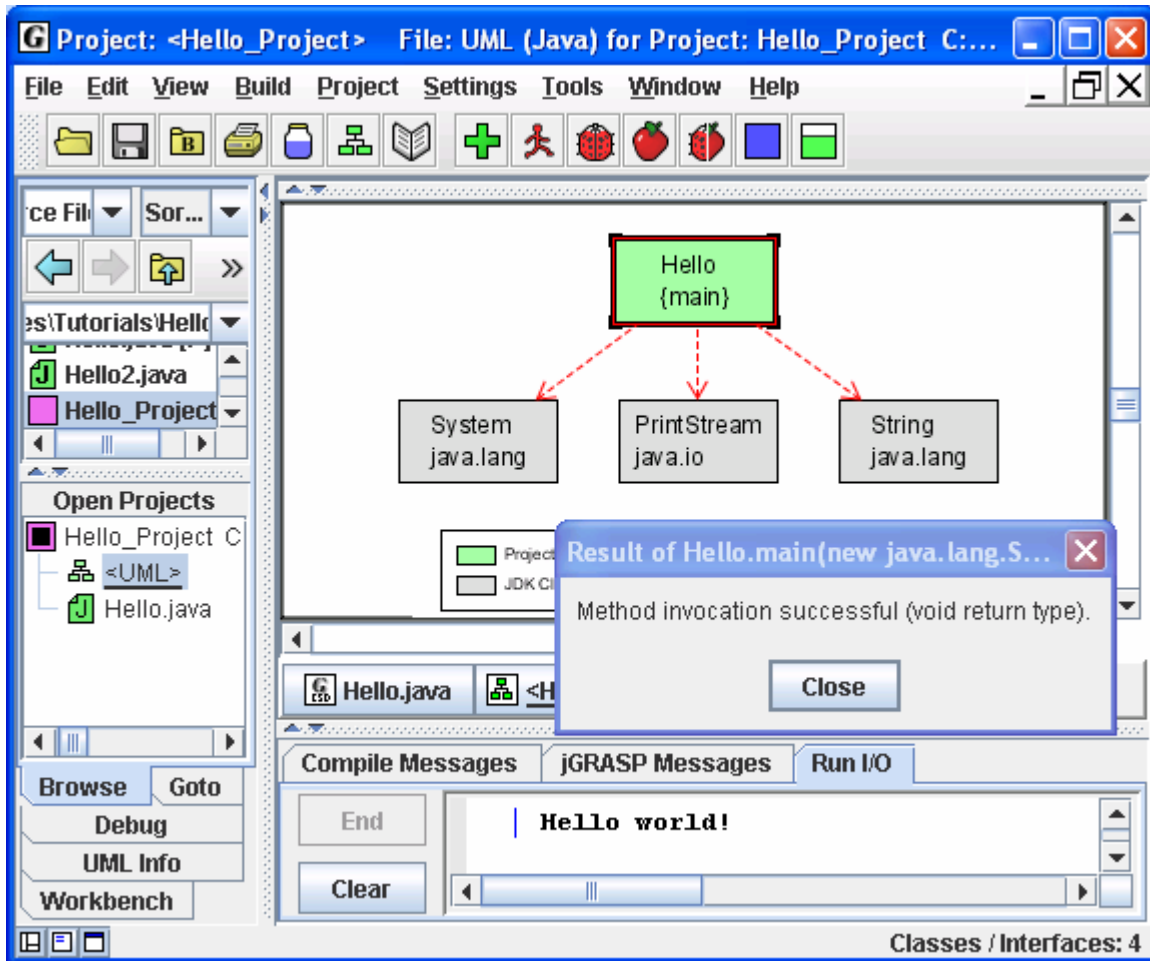



Figure 6-5. Invoking a static method from a class

6.3 Creating an Object for the Workbench

Now we move to a more interesting example which contains multiple classes. Figure 6-5 shows the PersonalLibraryProject loaded in the UML window. We could invoke main by following the procedure described in the preceding section (i.e., right-clicking on PersonalLibrary and selecting **Invoke Method** or by clicking the Invoke Static method button  on the toolbar. However, in this section we want to create objects and place them on the workbench. In the next section, we'll see how to invoke the instance (or non-static) methods of the objects we've placed on the workbench.

So we begin by right clicking on the Fiction class in the UML diagram, and then selecting **Create New Instance**, as shown in Figure 6-6. A list of constructors will be displayed in a dialog box.

If a parameterless constructor is selected as shown in Figure 6-7, 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 6-8. The arguments (values of the parameters) should be filled in prior to clicking **Create**. Remember to enclose String arguments in double quotes.

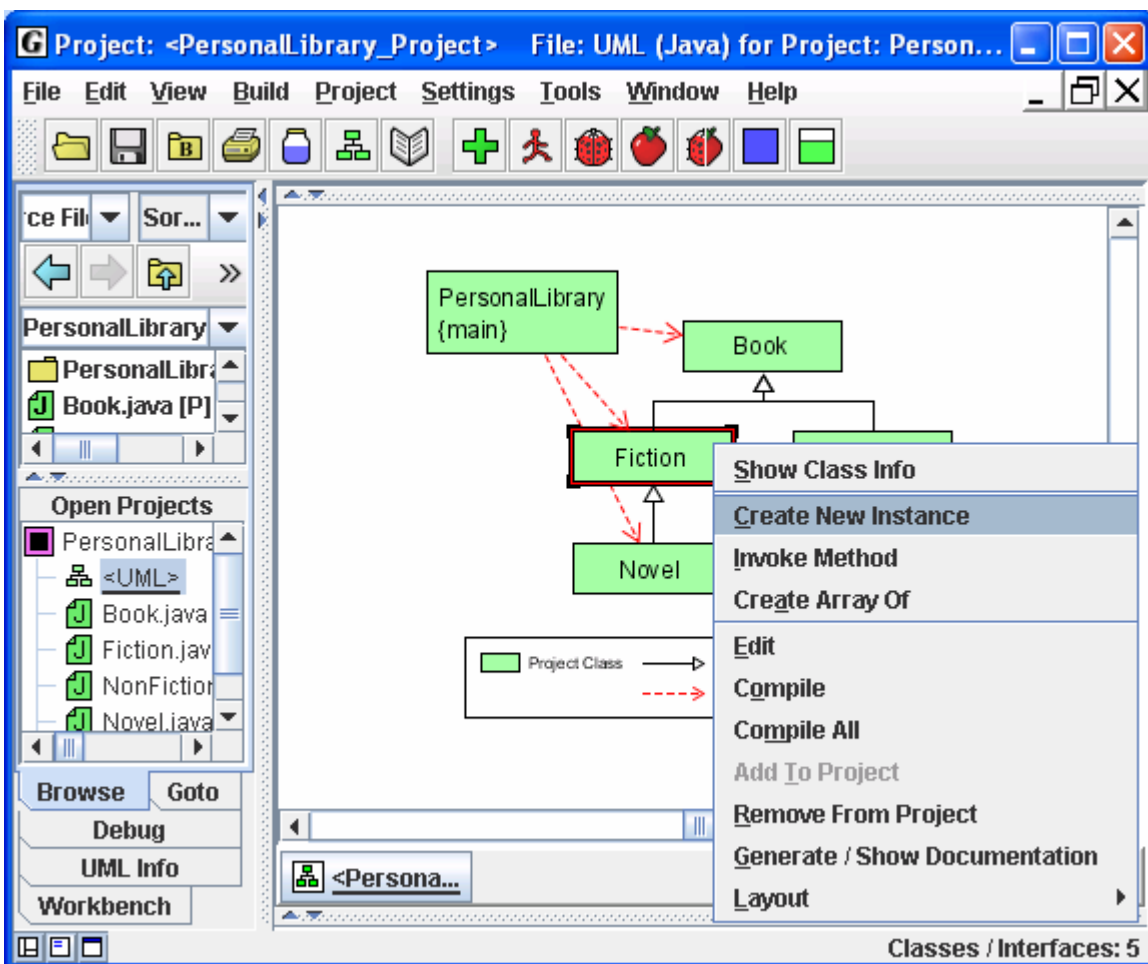


Figure 6-6. Creating an Object for the Workbench

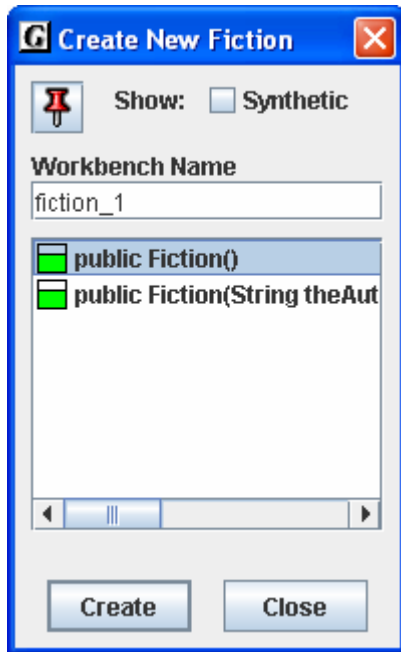




Figure 6-7. Selecting a constructor

In either case above, 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 “stay up” after you create an instance. This is handy for creating multiple instances of the same class. Click on the “stick-pin”  (it should turn darker), then click the Create button three times and you should see three new instances appear on the workbench.

In Figure 6-9, the Workbench tab is shown after four instances (or objects) of Fiction have been created. Notice fiction_2 and fiction_3 have been expanded so that their respective fields (mainCharacter, author, title, and pages) can be viewed. Since the first three fields are instances of the String class, they too can also be expanded. You should also note that mainCharacter is color coded green since it is the only field actually declared in Fiction. The other fields are color coded orange to indicate they are inherited from a parent, which in this case is Book. The placement of these fields in Book vs. Fiction was a design decision. Since not all books have a mainCharacter (e.g., a math book) but works of fiction almost certainly do, mainCharacter was defined in Fiction. Notice that Novel, a subclass (or child) of Fiction, appropriately inherits mainCharacter.

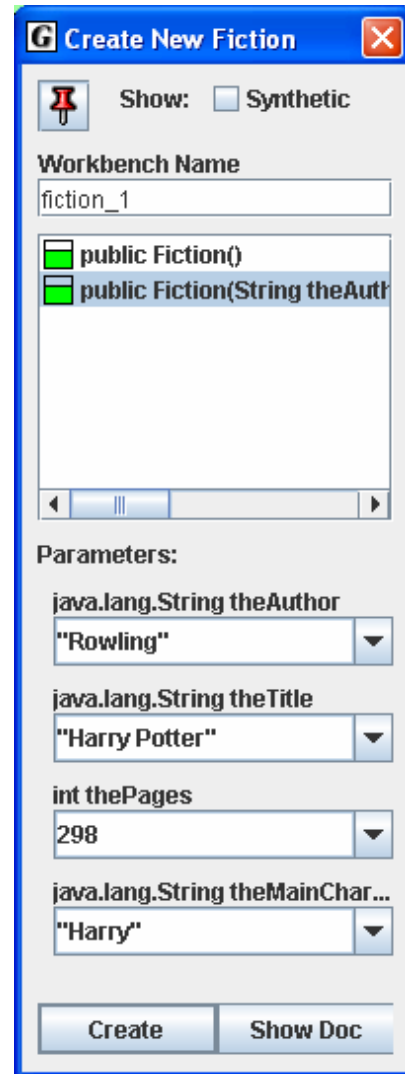


Figure 6-8. Constructor with parameters

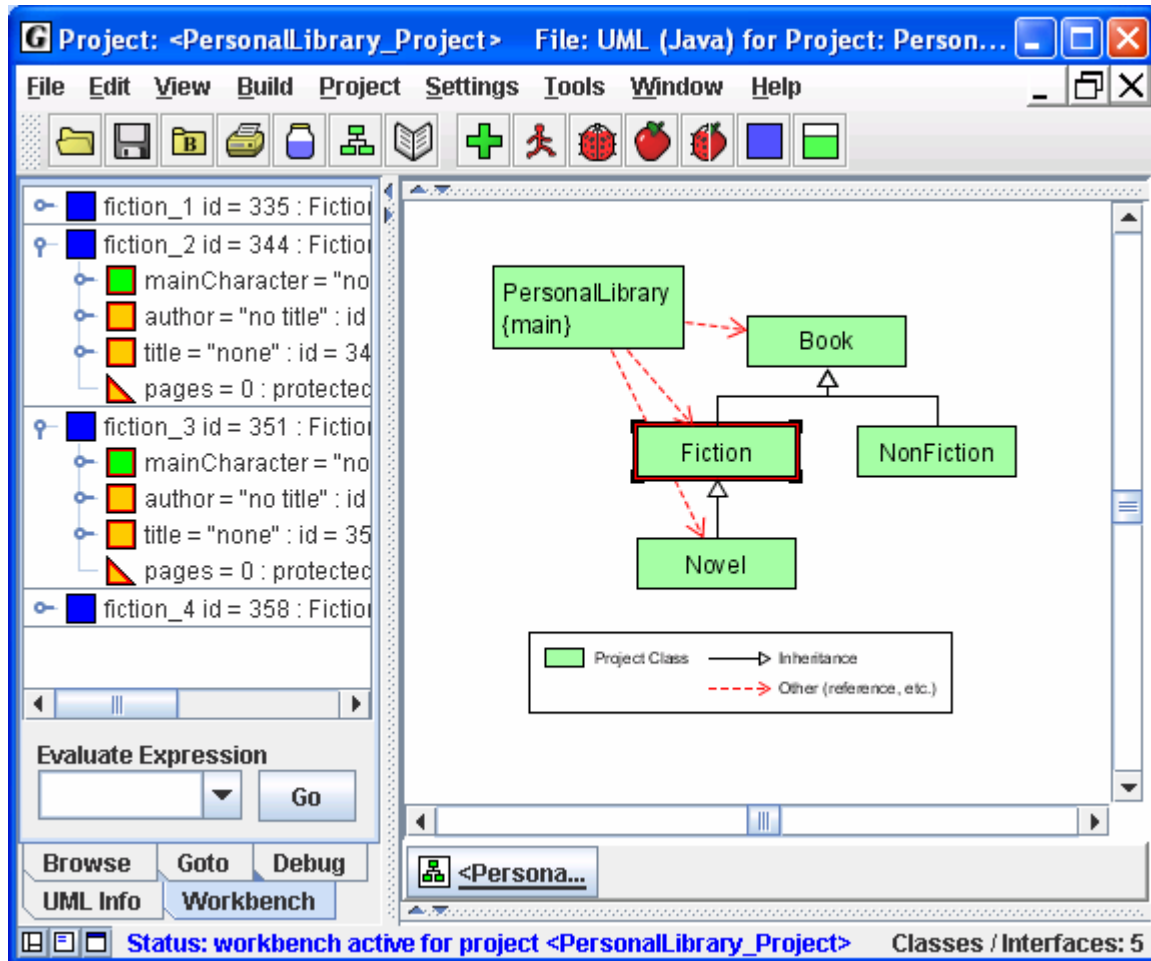


Figure 6-9. Workbench with four instances of Fiction

6.4 Invoking a Method

To invoke a method for an object on the workbench, select the object, right click, and then select **Invoke Method**. In Figure 6-10, fiction_2 has been selected, followed by a right mouse click, and then Invoke Method has been selected. A list of user methods visible from Fiction will be displayed in a dialog box as shown in Figure 6-11. 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. Other output, if any, is handled in the usual way. If a method updates a field (e.g., setMainCharacter()), the

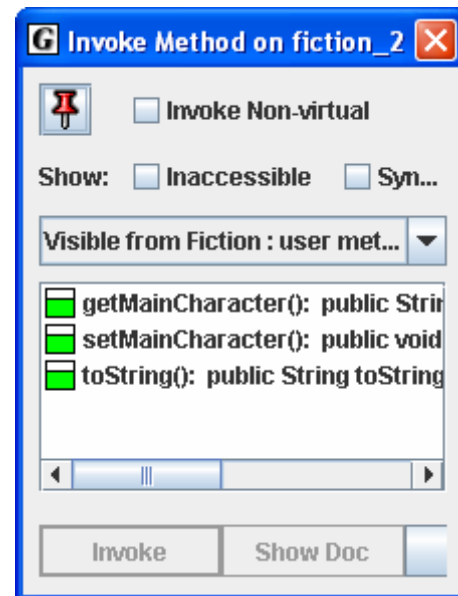



Figure 6-10. Selecting a method

effect of the invocation is seen in the 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 stay up. This is useful for invoking multiple methods for the same object. For example, in a graphics program a “move” method could be clicked repeatedly to see an object move across the display.

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, we can invoke each of its three methods: `getMainCharacter()`, `setMainCharacter()`, and `toString()`. By reviewing the results of the method invocations, we are essentially testing our class without a driver program.

6.5 Invoking Methods with Parameters Which Are Objects

If a method (or constructor) requires parameters that are primitive types and/or strings, these can be entered directly. However, if a parameter requires an object, then you must create an object instance for the workbench first. Then you can simply drag the object (actually a copy) from the workbench to the parameter field in the Invoke Method dialog.

6.6 Invoking Methods on Object Fields

If you have an object in the Workbench tab, you can expand it to reveal its fields. In Figure 6-9, `fiction_2` and `fiction_3` are expanded to show their fields (`mainCharacter`, `author`, `title`, and `pages`). Since the field `mainCharacter` is itself an object of the class `String`, any of the `String` methods can be invoked on it. For example, right-click on `mainCharacter` in `fiction_2`, then select **Invoke Method**. When the dialog pops up (Figure 6-10), you’ll see a rather lengthy list of all the methods visible to `String` objects. Scroll down the list and select the first `toUpperCase()` method, and then click **Invoke**. This should pop up the Result dialog with “HARRY” as the return value (Figure 6-11). 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.

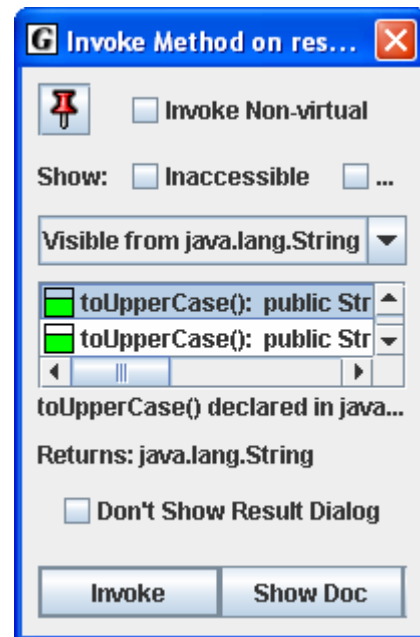


Figure 6-10. Invoking a String method

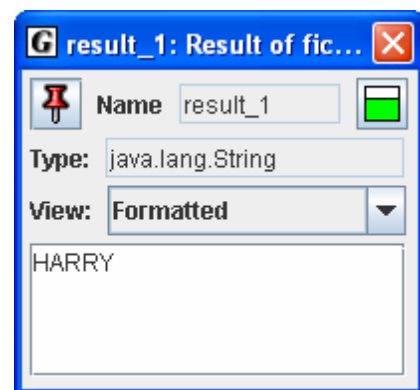


Figure 6-11. Result of `fiction_2.mainCharacter.toUpperCase()`

6.7 Selecting Categories of Methods to Invoke

The Invoke Method dialog provides a list of categories of method on a drop-down list. The default category is “Visible from *object name* – user methods only.” As the category name suggests, this list includes methods defined in the object’s class as well as those inherited from parent classes. This category was selected as the default so that the *all* user defined methods could be conveniently viewed. In this section, we’ll explore the various categories of methods.

Let’s create an instance of Novel by right-clicking on Novel in the UML window and then selecting **Create New Instance**. On the Create dialog, choose the parameterless constructor and click **Create**. Now you should see novel_1 on the workbench. Right-clicking on novel_1 and then selecting the Invoke Method will open the Invoke Method dialog as shown in Figure 6-12. Notice the first two methods are inherited (gold method symbols) and the third is defined in Novel (green method symbol). Now look back at the Invoke Method dialog for fiction_2 in Figure 6-10. The same methods are listed, but all are marked with green method symbols since those are defined in the Fiction class. One should surmise from this that both Fiction and Novel must have their own *toString* method.

Now let’s look at another category of method on the Invoke Method dialog for novel_1. Click the drop-down list on the dialog (see info box for Figure 6-12) and select “Declared in superclass Fiction”. Notice that the *toString* method in figure 6-13 has a gray bar through its gold method symbol to indicate that it has been

Click pull-down list to select a **category** of methods.

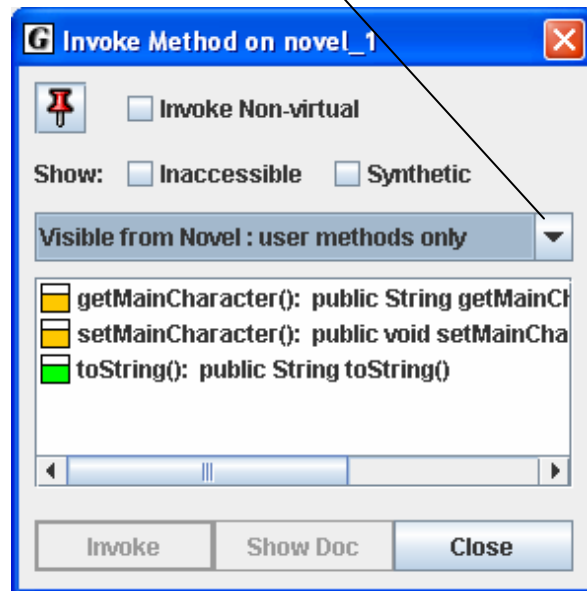


Figure 6-12. Invoking a method for novel_1

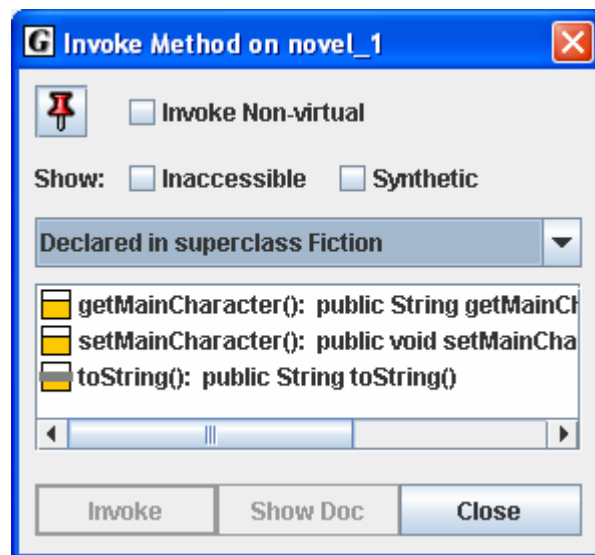


Figure 6-13 Methods declared in superclass Fiction

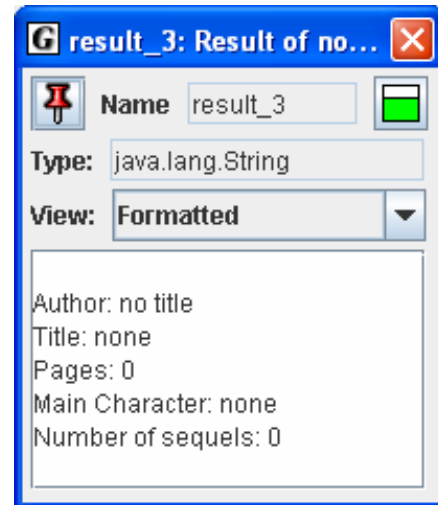
overridden by the *toString* method defined for Novel. This means that if you select and invoke the *toString* method listed in Figure 6-13, the *toString* defined in Novel will be the one that gets called. Remember, it is the object itself that determines which method is actually called. In your Java program, if you wanted to call an overridden method for an object, you would need to cast the object to the superclass and then call the method. jGRASP provides a short cut for doing this on the workbench with the “Invoke Non-virtual” check box on the dialog. In the example in 6-13, if you invoke the *toString* method without the checking the box for Invoke Non-virtual, Novel’s *toString* method is called, and you get the result shown in Figure 6-14. However, if you invoke the method with the box checked, Fiction’s *toString* method is called, and you get the result in Figure 6-15. Notice the only difference is that Novel’s *toString* method includes one more line of text (“Number of sequels: 0”) than Fiction’s *toString* method.

The other two check boxes “Inaccessible Methods” and “Synthetic Methods” are primarily for advanced users. The first can be used to display inaccessible methods such as inherited private methods. The second provides a list of synthetic methods created by the compiler such as access methods for fields of inner classes.

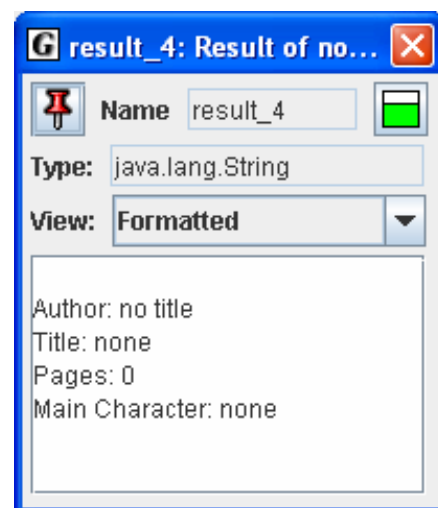
To wrap up this section, you are invited to select among the other **categories** of methods that can be displayed on the Invoke Method dialog for novel_1:

- Declared in superclass Book
- Declared in superclass java.lang.Object
- Visible from Novel

Notice that novel_1 inherits a large number of methods from java.lang.Object. The most inclusive category is “Visible from Novel” which includes all available methods. Perhaps now you see why the default category is “Visible from Novel – user methods only.”



**Figure 6-14. Viewing
superclasses for novel_1**




**Figure 6-15. Viewing
superclasses for novel_1**

6.8 Opening Object Viewers

A separate *Viewer* window can be opened for any object (or field of an object) on the workbench. All objects have a *basic* view which is the view shown in the workbench and debug tabs. However, some objects will have additional views.

The easiest way to open a viewer is to left-click on an object and drag it from the workbench to the location where you want the viewer to open. This will open a “view by name” viewer. You can also open a viewer by right-clicking on the object and selecting either **View by Value** or **View by Name**.

Figure 6-16 shows an object viewer for the *title* field of *fiction_3* which is a String object in an instance of *Novel*. *Formatted* is the default “view” for a String object which is especially useful when viewing a String object with a large value (e.g., a page of text). In Figure 6-17, the *Basic* view has been selected and expanded to show the gory details of the String object. Notice the first field is *value[12]* which is a character array holding the actual value of the string. If we open a separate viewer on value, we have a nice *Presentation* view of the array. You can also select the *Char Array* view for String as shown in Figure 6-18. In tutorial, *Viewers for Objects and Primitives*, additional Presentation views will be discussed. You are encouraged to open separate viewers for the objects on the workbench. In addition to providing multiple views of the object, each viewer includes an Invoke Method button  for the object being viewed.

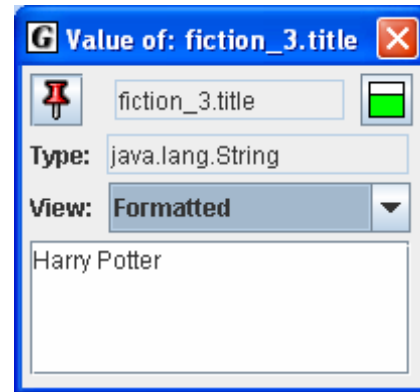


Figure 6-16. Viewing a String Object

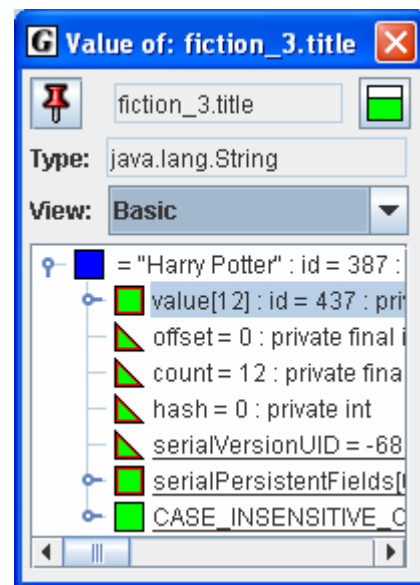


Figure 6-17. Basic view of a string (expanded to see fields)

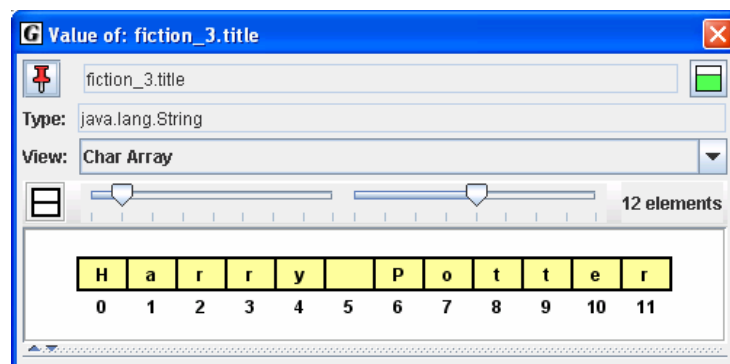


Figure 6-18. Char Array view of "Harry Potter"

6.9 Running the Debugger on Invoked Methods

When objects are on the workbench, the workbench is actually running the Java Virtual Machine (JVM) in debug mode. Thus, if you have a class open in a CSD window and set a breakpoint in one of its methods and then invoke the method from the workbench, the CSD window will pop to the top when the breakpoint is reached. At this time, you can single step through the program, examine fields, resume, etc. in the usual way. See the tutorial on “*The Integrated Debugger*” for more details.

6.10 Exiting the Workbench

The workbench is *running* whenever you have objects on it. If you attempt to do an operation that conflicts with workbench (e.g., recompile a class, switch projects, etc., jGRASP will prompt you with a message indicating that the workbench process is active and ask you if it is OK to end the process (Figure 6-19). When you try to exit jGRASP, you will get a similar message (Figure 6-12). These prompts are 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**.

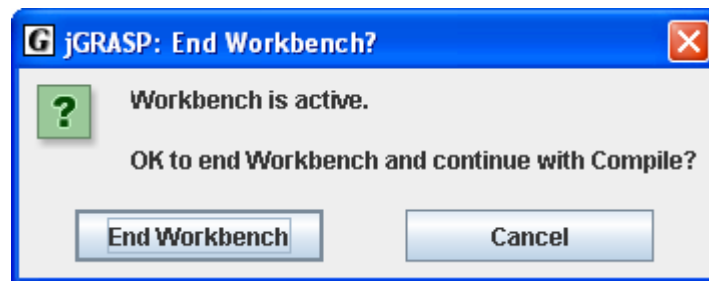


Figure 6-19. Making sure it is okay to exit the Workbench

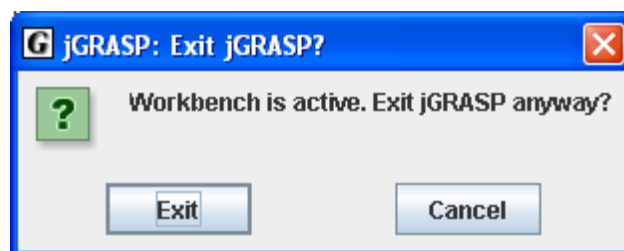


Figure 6-17. Making sure it is okay to exit

7 The Integrated Debugger

Your skill set for writing programs would not be complete without knowing how to use a debugger. While a debugger is traditionally associated with finding bugs, it can also be used as a general aid for understanding your program as you develop it. jGRASP provides a highly visual debugger for Java, which is tightly integrated with the CSD and UML windows, the Workbench, and the Viewers. The jGRASP debugger includes all of the traditional features expected in a debugger.

If the example program used in this section is not available to you, or if you do not understand it, simply substitute your own program in the discussion.

Objectives – When you have completed this tutorial, you should be able to set breakpoints and step through the program, either by single stepping or auto stepping. You should also be able to display the dynamic state of objects created by the program using the appropriate Object Viewer.

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

7.1 Preparing to Run the Debugger

7.2 Setting a Breakpoint

7.3 Running a Program in Debug Mode

7.4 Stepping Through a Program – the Debug Buttons

7.5 Stepping Through a Program – without *Stepping In*

7.6 Stepping Through a Program – and *Stepping In*

7.7 Opening Object Viewers

7.8 Debugging a Program

7.1 Preparing to Run the Debugger

In preparation to use the debugger, we need to make sure that programs are being compiled in debug mode. This is the default, so this option is probably already turned on. With a CSD or UML window in focus, click **Build** on the menu and make sure **Debug Mode** is checked. If the box in front of Debug Mode is not checked, click on the box. When you click on Build again, you should see that Debug Mode is checked. When you compile your program in Debug Mode, information about the program is included in the .class file that would normally be omitted. This allows the debugger to display useful details as you execute the program. If your program has not been compiled with Debug Mode checked, you should recompile it before proceeding.

7.2 Setting a Breakpoint

In order to examine the state of your program at a particular statement, you need to set a breakpoint. The statement you select must be “executable” rather than a simple declaration. To set a breakpoint in a program, move the mouse to the line of code and left-click the mouse to move the cursor there. Now right-click on the line to display a set of options that includes **Toggle Breakpoint**. For example, in Figure 7-1 the cursor is on

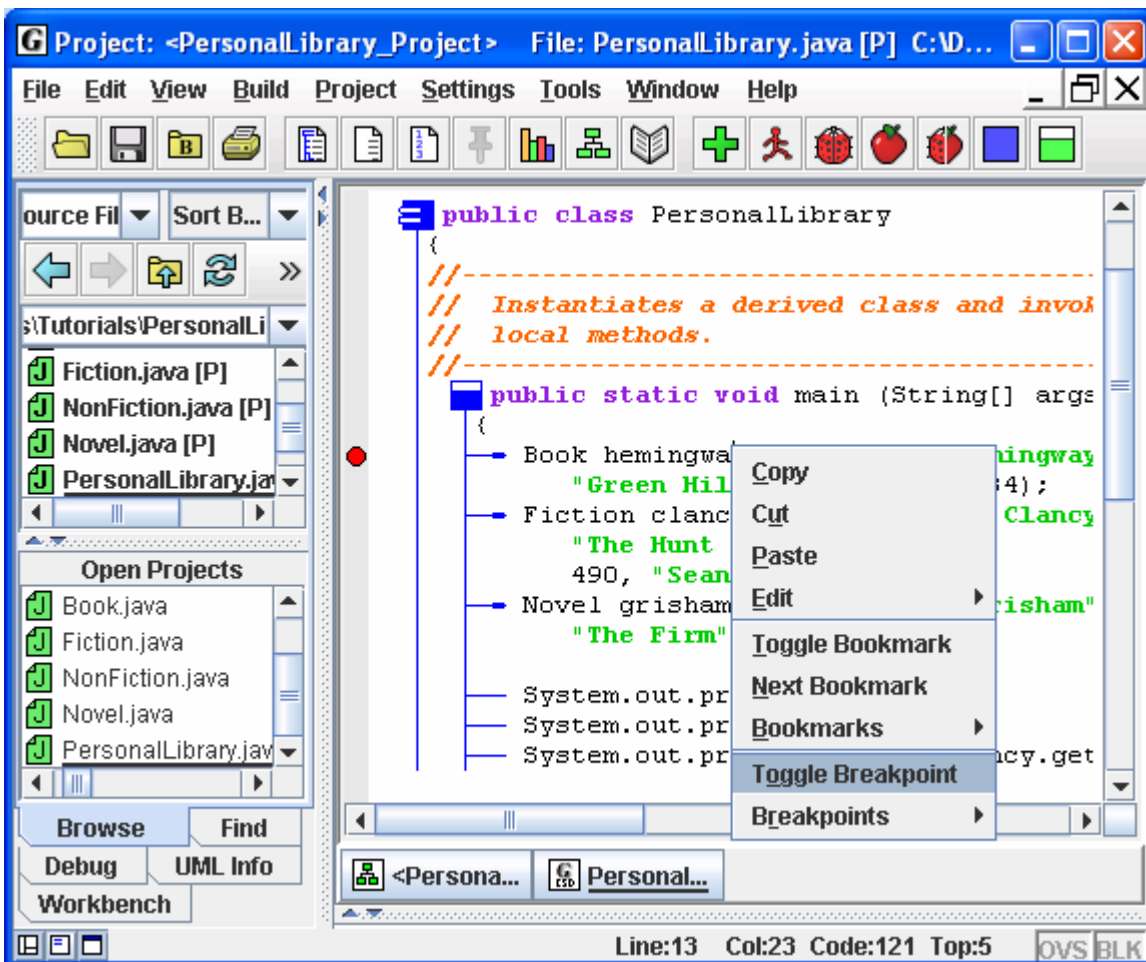





Figure 7-1. Setting a breakpoint

the first executable line in *main* (which declares *Book hemingway ...*), and after Toggle Breakpoint is selected in the options popup menu, a small red stop sign symbol  appears in the left margin of the line to indicate that a breakpoint has been set. To remove a breakpoint, you repeat the process since this is a toggle action. You may set as many breakpoints as needed.


You can also set a breakpoint by hovering the mouse over the leftmost column of the line where you want to set the breakpoint. When you see the red octagonal breakpoint symbol , you just left-click the mouse to set the breakpoint. You can remove a breakpoint by clicking on the red octagonal. This second approach is the one most commonly used for setting and removing breakpoints.

7.3 Running a Program in Debug Mode

After compiling your program in Debug Mode and setting one or more breakpoints, you are ready to run your program with the debugger. You can start the debugger in one of two ways:

- (1) Click **Build – Debug** on the CSD window menu, or
- (2) Click the Debug button  on the toolbar.

After you start the debug session, several things happen. In the Run window near the bottom of the Desktop, you should see a message indicating the debugger has been launched. In the CSD window, the line with the breakpoint set is eventually highlighted, indicating that the program will execute this statement next. On the left side of the jGRASP desktop, the Debug tab is popped to the top. Each of these can be seen in Figure 7-2. Notice the Debug tab pane is further divided into three sub-panes or sections labeled **Threads**, **Call Stack**, and **Variables/Eval**. Each of these sections can be resized by selecting and dragging one of the horizontal partitions.

The **Threads** section lists all of the active threads running in the program. In the example, the red thread symbol  indicates the program is stopped in *main*, and green indicates a thread is running. Advanced users should find this feature quite useful for starting and stopping individual threads in their programs. However, since beginners and intermediate users rarely use multi-threading, the thread section is closed when the debugger is initially started. Once the Threads section is dragged open, it remains open for the duration of the jGRASP session.

The **Call Stack** section is useful to all levels of users since it shows the current call stack and allows the user to switch from one level to another in the call stack. When this occurs, the CSD window that contains the source code associated with a particular call is popped to the top of the desktop.

The **Variables/Eval** section shows the details of the current state of the program in the *Variables* tab and provides an easy way to evaluate expressions involving these variables in the *Eval* tab. Most of your attention will be focused on the *Variables* tab where you can monitor all current values in the program. From the *Variables* tab, you can also launch separate viewers on any primitives or objects as well as fields of objects.

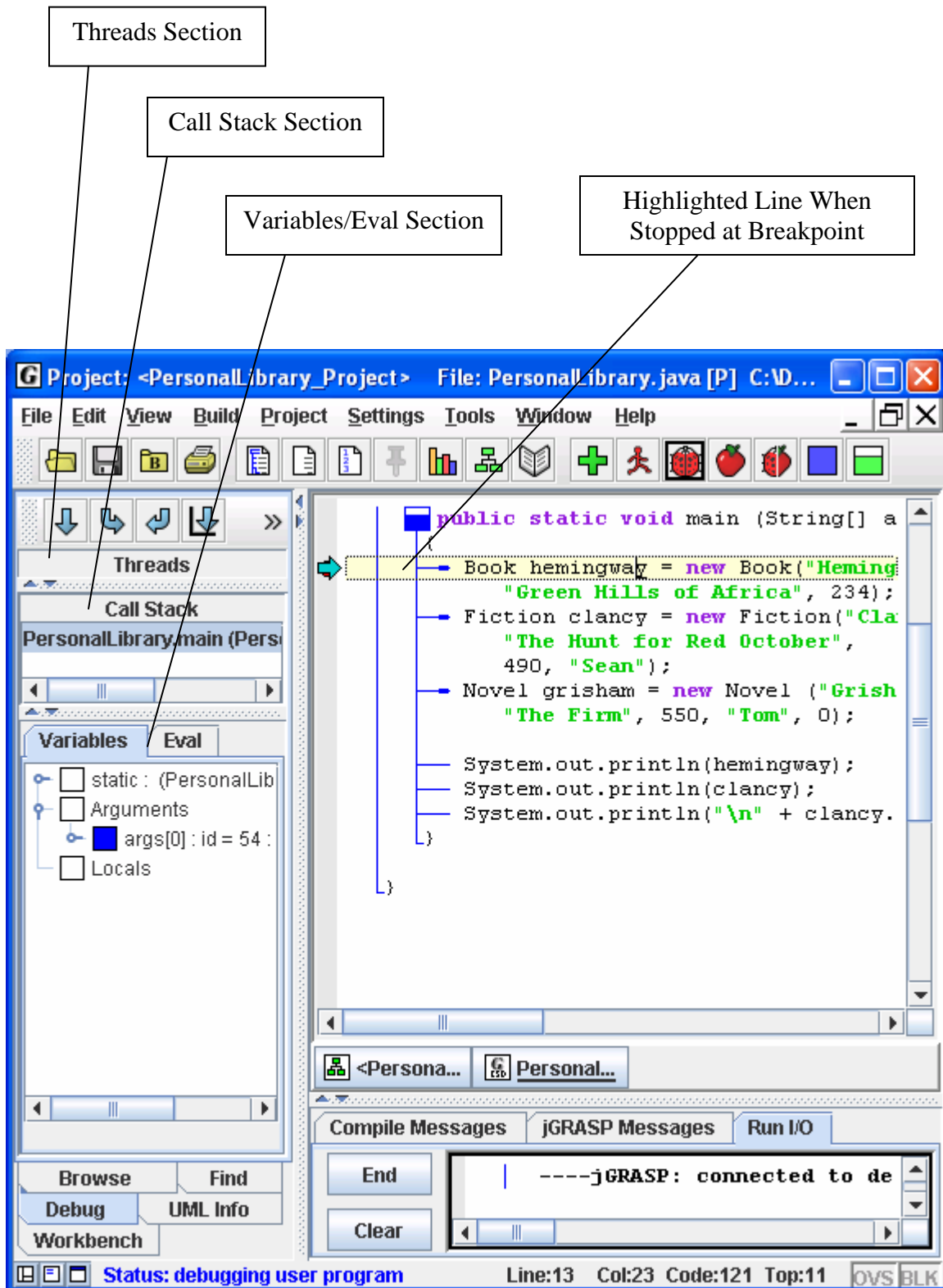














Figure 7-2. Desktop after debugger is started

7.4 Stepping Through a Program – the Debug Buttons




After the program stops at the breakpoint (Figure 7-2), you can use the buttons at the top of the Debug tab to *step*, *step into* a method call, *step out* of a method, *run to the cursor*, *pause* the current thread, *resume*, turn on/off *auto step* mode, turn on/off *auto resume* mode, and *suspend* new thread. Note that an application program begins at the first executable statement in the *main* method. The sequence of statements that is executed when you run your program is called the *control path* (or simply *path*). If your program includes statements with conditions (e.g., *if* or *while* statements), the control path will reflect the *true* or *false* state of the conditions in these statements.

- ↓ Clicking the *Step* button will single step to the next statement. The highlighted line in the CSD window indicates the statement that's about to be executed. When the Step button is clicked, that statement is executed and the "highlighting" is moved to the next statement along the control path.
- ↙ Clicking the *Step in* button for a statement with a method call that's part of the user's source code will open the new file, if it's not already open, and pop its CSD window to the top with the current statement highlighted. The top entry in the *Call Stack* indicates where you are in the program. Note that clicking the *Step in* button for a statement without a method call is equivalent to clicking *Step*.
- ↘ Clicking the *Step out* button will take to the statement in the CSD window from which you previously stepped in. The *Call Stack* will be updated accordingly.
- ↙ Clicking the *Run to Cursor* button will cause your program to step automatically until the statement with the cursor L is reached. If the cursor is not on a statement along the control path, the program will stop at the next breakpoint it encounters or at the end of the program. The *Run to Cursor* button is convenient since placing the cursor on a statement is like setting "temporary" breakpoint.
- || Clicking the *Pause* button suspend the program running in debug mode. Note that if you didn't have a breakpoint set in your code, you may have to select the main thread in the *Threads* section before the *Pause* button is available. After the program has halted, refer to the *Call Stack* and select the last method in your source code that was invoked. This should open the CSD window containing the method with the current line highlighted. Place a breakpoint on the next line and click the *step* ↓ button to advance through the code.
- Clicking the *Resume* button advances the program along the control path to the next breakpoint or to the end of the program. If you have set a breakpoint in a CSD window containing another file and this breakpoint is on the control path (i.e., in a method that gets called), then this CSD window will pop to the top when the breakpoint is reached.

-  The *Auto Step* button is used to toggle off and on a mode which allows you to step repeatedly after clicking the *step*  button only once. This is an extremely useful feature in that it essentially let's you watch your program run. Notice that with this feature turned on, a *Delay* slider bar appears beneath the Debug controls. This allows you to set the delay between steps from 0 to 26 seconds (default is .5 seconds). While the program is auto stepping, you can stop the program by clicking the *Pause*  button. Clicking the *Step*  button again continues the auto stepping. Remember after turning on Auto Step , you always have to click the *step*  button once to get things rolling.
-  The *Auto Resume* button is used to toggle off and on a mode which allows you to resume repeatedly after clicking the *Resume*  button only once. The effect is that your program moves from breakpoint to breakpoint using the delay indicated on the *delay* slider bar. As with auto step above, you can click the *Pause*  button to interrupt the auto resume; then clicking the *Resume*  button again continues the auto resume.
-  The *Use Byte Code Size Steps* button toggles on and off the mode that allows you to step through a program in the smallest increments possible. With this feature off, the step size is approximately one source code statement, which is what most users want to see. This feature is seldom needed by beginning and intermediate programmers.
-  The *Suspend New Threads* button toggles on and off the mode that will immediately suspend any new threads that start. With this feature on when the debugging process is started, all startup threads are suspended as soon as is possible. Unless you are writing programs with multiple threads, you should leave the feature turned off.

As you move through the program, you can watch the call stack and contents of variables change dynamically with each step. The integrated debugger is especially useful for watching the creation of objects as the user steps through various levels of constructors. The jGRASP debugger can be used very effectively to explain programs, since a major part of understanding a program is keeping track (mentally or otherwise) of the state of the program as one reads from line to line. We will make two passes through the example program as we explain it. During the first pass, we will “step” through the program without “stepping into” any of the method calls, and we will concentrate on the Variable section.

7.5 Stepping Through a Program – without *Stepping In*

After initially arriving at the breakpoint in Figure 7-2, the **Variables/ Settings** section indicates no local variables have been declared. Figure 7-3 shows the results of clicking the *Step*  button to move to the next statement. Notice that under Locals in the **Variables/Eval** section, we now have an instance of *Book* called *hemingway*. Objects, represented by a colored square, can be opened and closed by clicking the “handle” in front of the square object. Primitives, like the integer *pages*, are represented by colored triangles. In Figure 7-3, *hemingway* has been opened to show the author, title, and pages fields. Each of the String instances (e.g., author) can be opened to show the details of a String object, including the character array that holds the actual value of the string.

Since *hemingway* is an instance of *Book*, the fields in *hemingway* are marked with green object or primitive symbols to indicate that they were declared in *Book*. Notice that the symbols for *author* and *title* have red borders since they were declared to be *private* in *Book* to indicate they are inaccessible from the current context of *main* in *PersonalLibrary*. The field *pages*, which was declared to be *protected* in *Book*, has a symbol without a red border. The reason for this is somewhat subtle. The *protected* field *pages* is accessible in all subclasses of *Book* as well as in any class contained the Java package containing *Book*. Since the *PersonalLibrary* program is not in a package, the directory containing it is considered the “package.” Thus, since *Book* is in the same directory as *PersonalLibrary*, the *protected* field *pages* is accessible to *PersonalLibrary*.

After executing the next statement in Figure 7-3, an instance of the *Fiction* class called *clancy* is created as shown in Figure 7-4. In the figure, *clancy* has been opened to reveal its fields. The field “*mainCharacter*” is green, indicating it is defined in *Fiction*. The

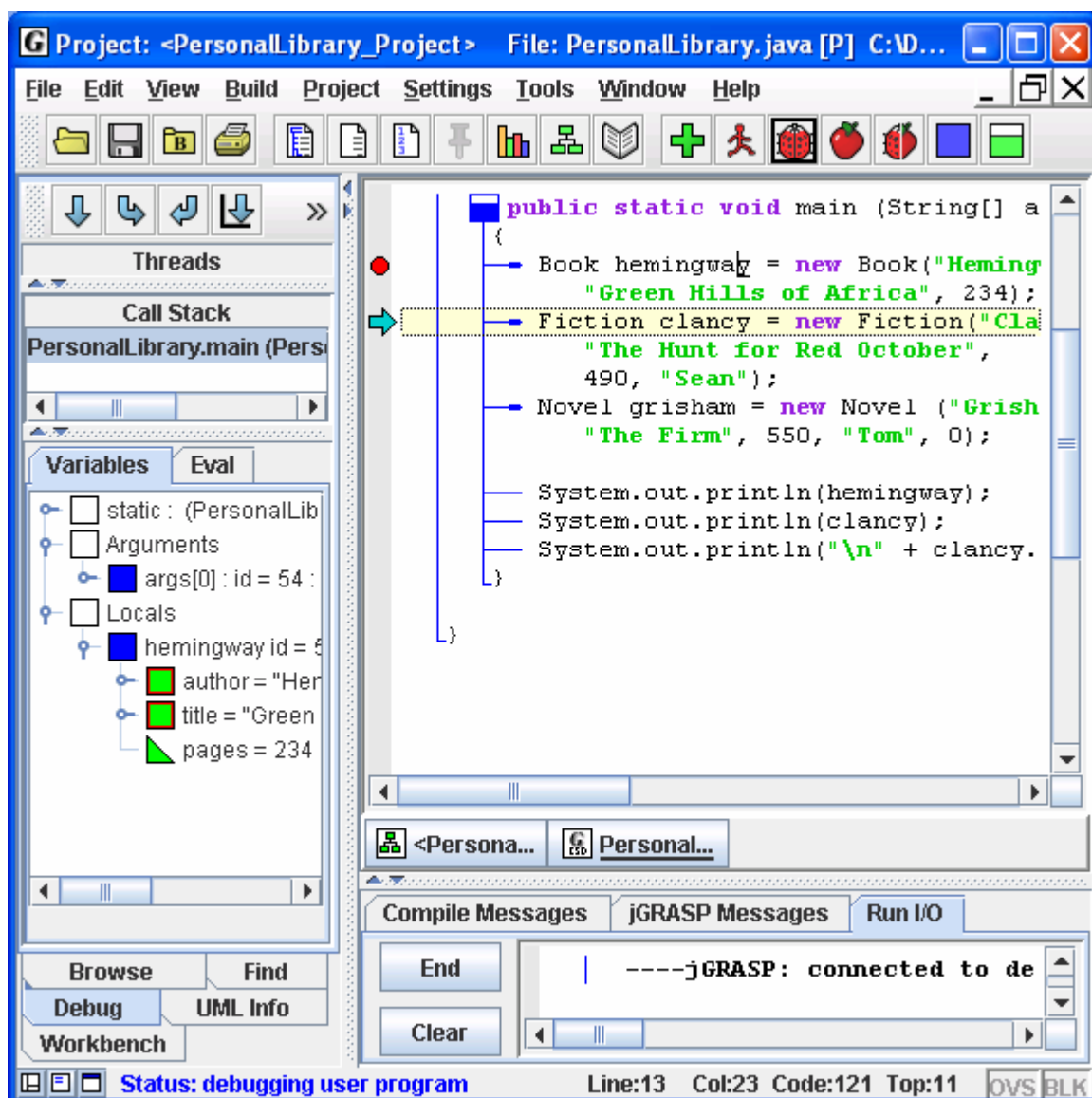


Figure 7-3. Desktop after hemingway (book) is created

other fields (author, title, and pages) are orange, which indicates these fields were inherited from Book.

As you continue to step through your program, you should see output of the program displayed in the Run I/O window in the lower half of the Desktop. Eventually, you should reach the end of the program and see it terminate. When this occurs, the Debug tab should become blank, indicating that the program is no longer running.

7.6 Stepping Through a Program – and *Stepping In*

Now we are ready to make a second pass and “step in” to the methods called. Tracing through a program by following the calls to methods can be quite instructive in the obvious way. In the object-oriented paradigm, it is quite useful for illustrating the concept of constructors. As before, we need to run the example program in the debugger

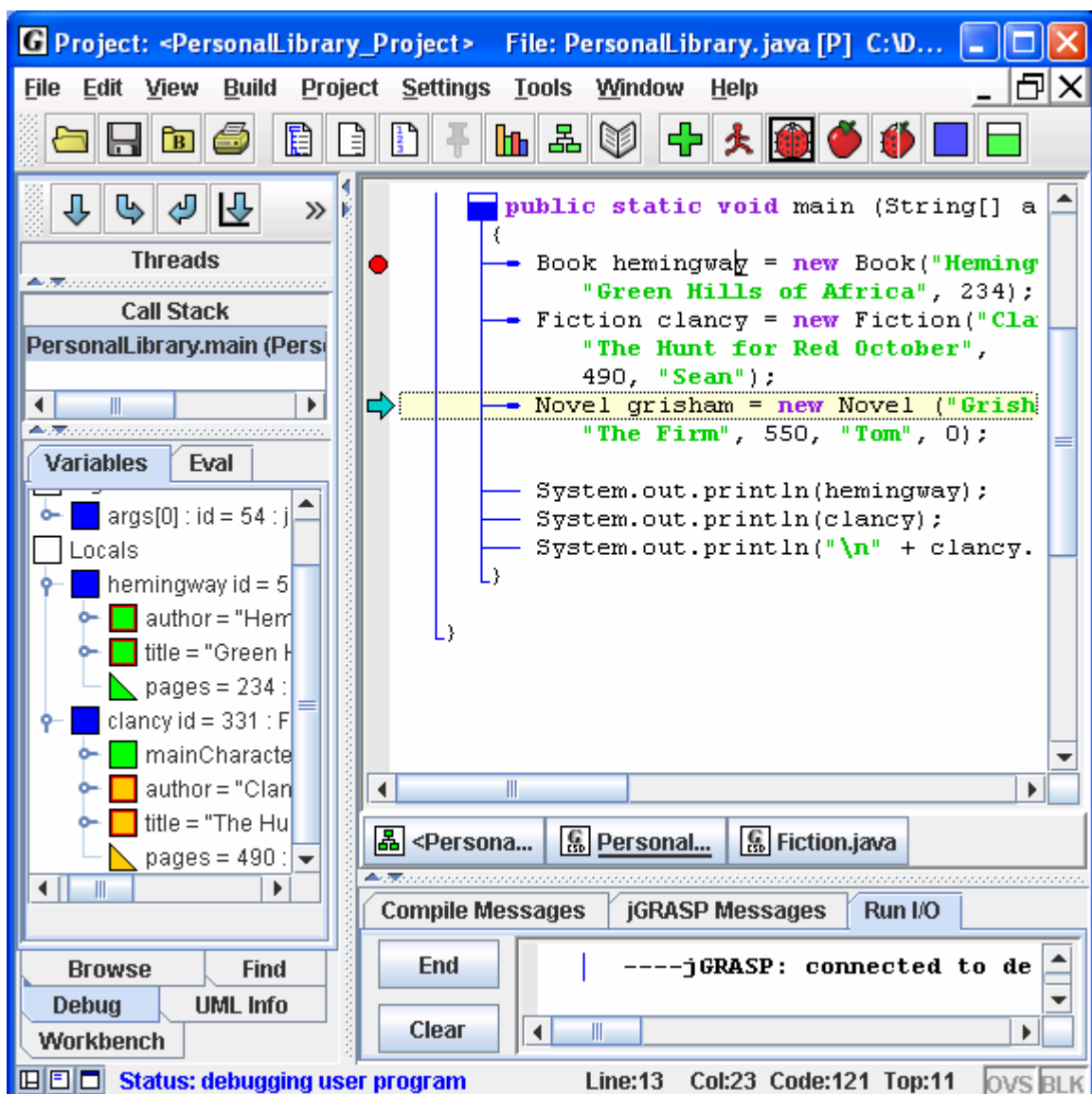




Figure 7-4. After next step and "clancy" created

by clicking **Build – Debug** on the CSD window menu or by clicking the debug button  on the toolbar. After arriving at the breakpoint, we click the *Step in* button  and the constructor for class `Book` pops up in the CSD window (Figure 7-5). Notice the *Call Stack* in the Debug tab indicates you have moved into `Book` from `PersonalLibrary` (i.e., the entry for `Book` is listed above `PersonalLibrary` in the *call stack*). If you click on the `PersonalLibrary` entry in the call stack, the associated CSD window will pop to the top and you see the variables associated with it. If you then click the `Book` entry, its CSD window pops to the top and you see the variables associated with the call to `Book`'s constructor. In Figure 7-5, the entry for *this* has been expanded in the *Variables* section. The *this* object represents the object that is being constructed. Notice none of the fields have a red border since we are inside the `Book` class. As you step through the constructor, you should see the fields in this get initialized to the values passed in as arguments. Also, note the *id* for *this* (it is 325 in our example debug session; it may be a different number in your session). You can then step through the constructor in the usual way, eventually returning to the statement in the main program that called the constructor. One more step should finally get you to the next statement, and you should see *hemingway* in the *Variables* section with the same *id* as you saw in the constructor as it was being built. If you expand *hemingway*, you should see the red borders are back on *author* and *title* since we're no longer in `book` class.

There are many other scenarios where this approach of tracing through the process of object construction is useful and instructive. For example, consider the case where the `Fiction` constructor for "clancy" is called and it in turn calls the super constructor located in `Book`. By stepping into each call, you can see not only how the program proceeds through the constructor's code, but also how fields are initialized.

Another even more common example is when the *toString* method of an object is invoked indirectly in a print statement (`System.out.println`). The debugger actually takes the user to the object's respective *toString* method.

7.7 Opening Object Viewers

A separate **Viewer** window can be opened for any primitive or object (or field of an object) displayed in *Variables* section of the Debug tab. All objects have a *basic* view which is the view shown in the Debug tab. However, when a separate viewer window is opened for an entry, some objects will have additional views.


The easiest way to open a viewer is to left-click on an object and drag it from the workbench to the location where you want the viewer to open. This will open a "view by name" viewer. You can also open a viewer by right-clicking on the object and selecting either **View by Value** or **View by Name**.

Figure 7-6 shows an object viewer for the *title* field of *hemingway* in Figure 7-4, which is a `String` object in an instance of `Book`. *Formatted* is the default "view" for a `String` object which is especially useful when viewing a `String` object with a large value (e.g., a



Figure 7-6. Viewing a String Object

page of text). In Figure 7-7, the *Basic* view has been selected and expanded to show the details of the String object. Notice the first field is value[21] which is a character array holding the actual value of the string. If we open a separate viewer on value, we have a *Presentation* view of the array as shown in Figure 7-8. Notice that the first element ('G') in the array has been selected and this opened a subview of type character. The subview displays the 'G' and its integral value of 71. If our example had been an array of strings (e.g., a list of words) then selecting an array element would have displayed the formatted view of a String object in the subview. Presentation view is the default for arrays. There is also a view called *Array Elements* which is quite useful for large arrays.

You are encouraged to open separate viewers for any of the primitives and objects in the Variables section of the Debug tab. In addition to providing multiple views of the object, each viewer includes an Invoke Method button  for the object being viewed. In the tutorial *Viewers for Objects and Primitives*, many other examples are presented along with a more detailed description of viewers in general.

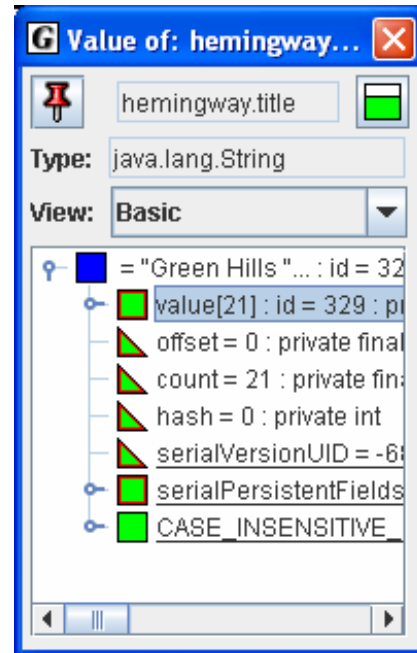


Figure 7-7. Basic view of a string (expanded to see fields)

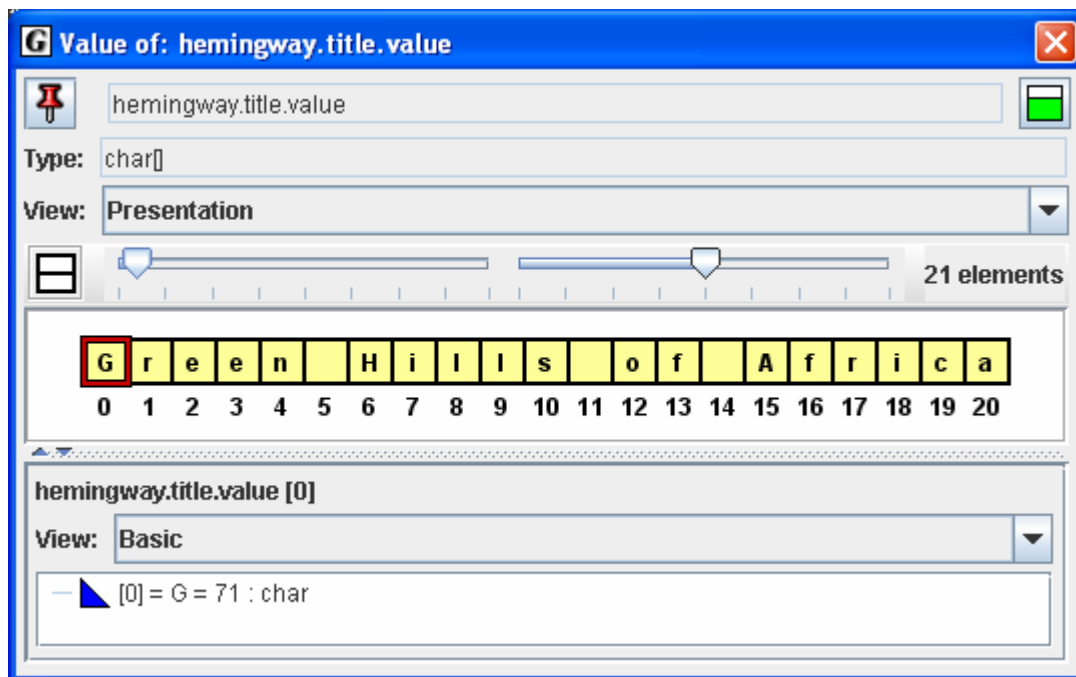



Figure 7-8. Presentation View of hemingway.title.value

7.8 Debugging a Program

You have, no doubt, noticed that the previous discussion was only indirectly related to the activity of finding and removing *bugs* from your program. It was intended to show you how to set and unset breakpoints and how to step through your program. Typically, to find a bug in your program, you need to have an idea where in the program things are going wrong. The strategy is to set a breakpoint on a line of code prior to the line where you think the problem occurs. When the program gets to the breakpoint, you can inspect the variables of interest to ensure that they have the correct values. Assuming the values are okay, you can begin stepping through the program, watching for the error to occur. Of course, if the value of one or more of the variables was wrong at the breakpoint, you will need to set the breakpoint earlier in the program.

You can also set several types of “watches” on a field of an object. In Figure 7-9, a *Watch for Access* has been set on the *title* in *hemingway* just after it was created. If you click the Resume button  at this point, with no breakpoints set before the end of the program, the next place the program should stop is in the *toString* method of *Book* in conjunction with the *println* statement for *hemingway*. This is because the *title* field of *hemingway* is accessed in the statement:

```
return( "\nAuthor: " + author +  
        "\nTitle: " + title +  
        "\nPages: " + pages );
```

Note that setting *Watch All for Access* on the *title* field of *hemingway* sets the *watch* on all occurrences of the *title* field (i.e., in all instances of *Book*, *Fiction*, and *Novel*).

As your programs become more complex, the debugger can be an extremely useful for both understanding your program and isolating bugs. For additional details, see *Integrated Java Debugger* in **jGRASP Help**.

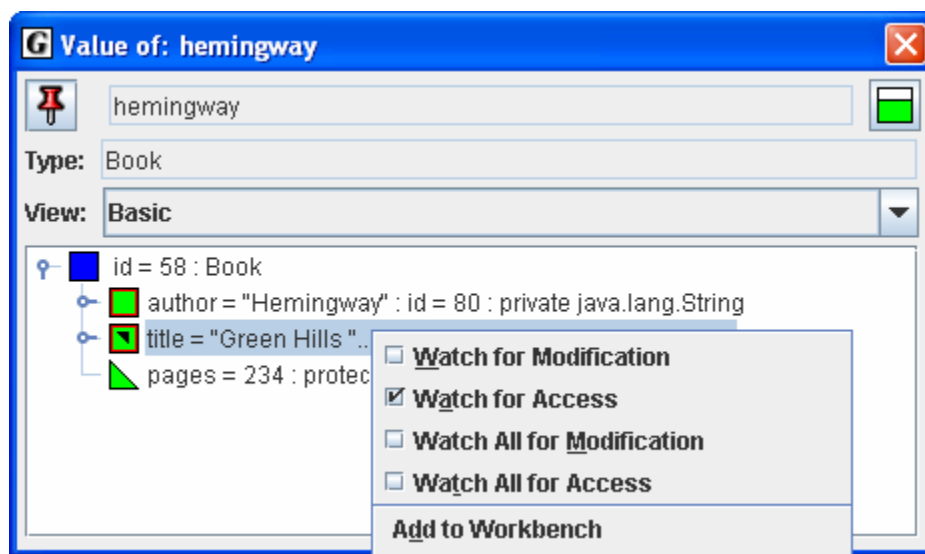


Figure 7-9. Setting a Watch for Access

8 The Control Structure Diagram (CSD)

The Control Structure Diagram (CSD) is an algorithmic level diagram intended to improve the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit. The CSD is an alternative to flow charts and other graphical representations of algorithms. The major goal behind its creation was that it be an intuitive and compact graphical notation that was easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and class diagrams.

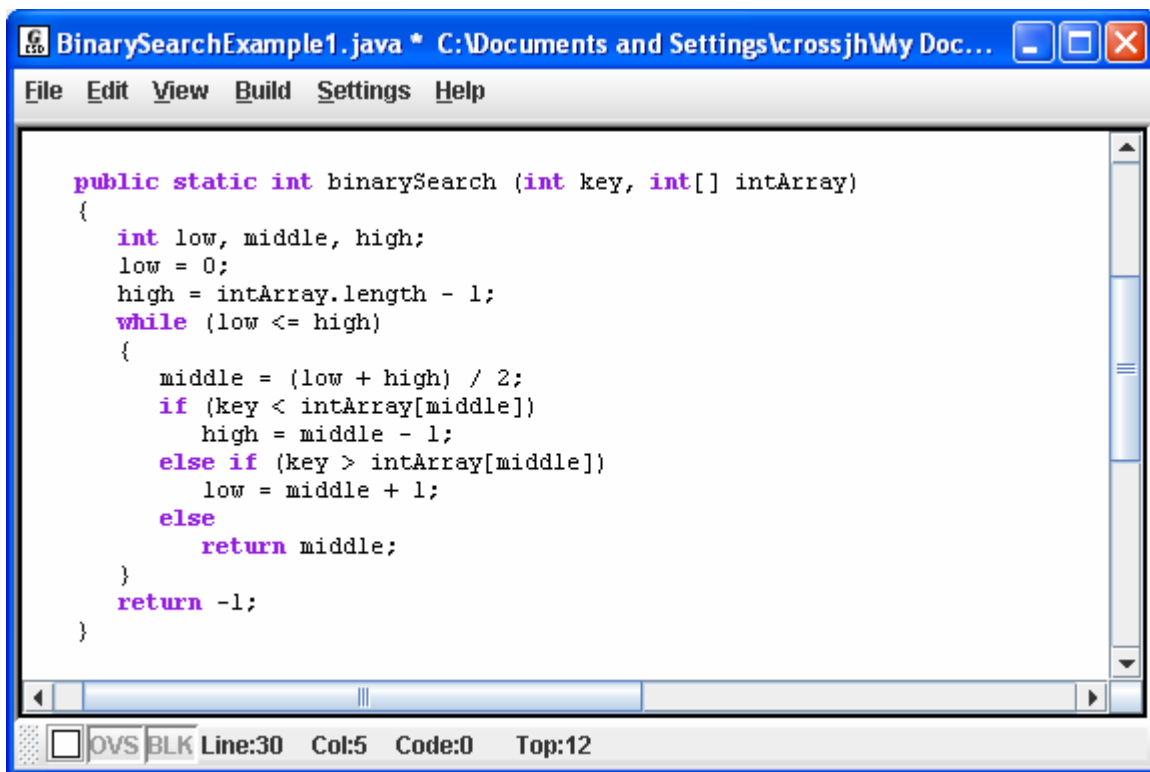
Objectives – When you have completed this tutorial, you should be able to use and understand the graphical notations used in the **CSD** for basic control constructs of modern programming language, including sequence, selection, iteration, exits, and exception handling.

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

- 8.1 An Example to Illustrate the CSD
- 8.2 CSD Program Components/Units
- 8.3 CSD Control Constructs
- 8.4 CSD Templates
- 8.5 Hints on Working with the CSD
- 8.6 Reading Source Code with the CSD
- 8.7 References

8.1 An Example to Illustrate the CSD

Figure 8-1 shows the source code for a Java method called `binarySearch`. The method implements a binary search algorithm by using a *while* loop with an *if..else..if* statement nested within the loop. Even though this is a simple method, displayed with colored keywords and traditional indentation, its readability can be improved by adding the CSD. In addition to the *while* and *if* statements, we see the method includes the declaration of primitive data (`int`) and two points of exit. The CSD provides visual cues for each of these constructs.



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

Figure 8-1. `binarySearch` method without CSD

Figure 8-2 shows the `binarySearch` method after the CSD has been generated. Although all necessary control information is in the source text, the CSD provides additional visual stimuli by highlighting the sequence, selection, and iteration in the code. The CSD notation begins with symbol for the method itself followed by the individual statements coming off the stem as it extends downward. The declaration of primitive data is highlighted with special symbol appended to the statement stem. The CSD constructs for the *while* statement is represented by the double line “loop” (with break at the top), and the *if* statement uses the familiar diamond symbol from traditional flowcharts. Finally, the two ways to exit from this method are shown explicitly with an arrow drawn from inside the method through the method stem to the outside.

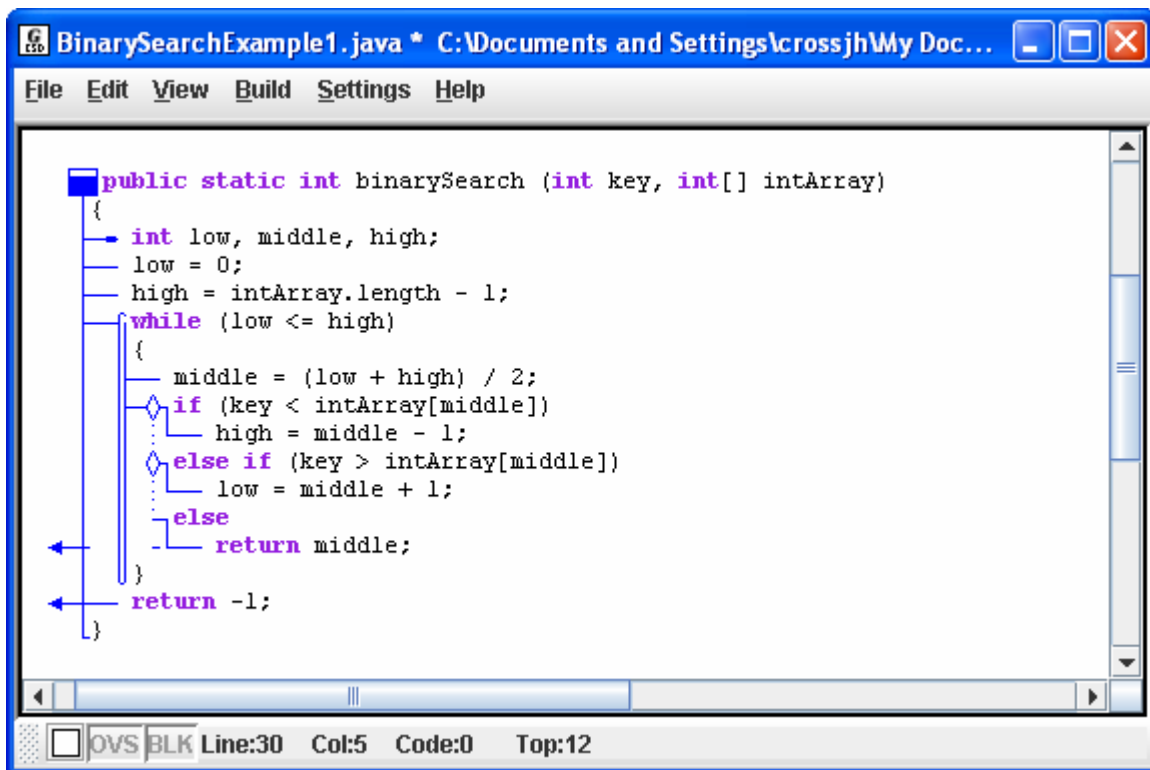


Figure 8-2. `binarySearch` with CSD

While this is a small piece of code, it does illustrate the basic CSD constructs. However, the true utility of the CSD can be realized best when reading or writing larger, more complex programs, especially when control constructs become deeply nested. A number of studies involving the CSD have been done and others are in progress. In one of these, CSD was shown to be preferred significantly over four other notations: flowchart, Nasir-Schneiderman chart, Warnier-Orr diagram, and the action diagram [Cross 1998]. In several later studies, empirical experiments were done in which source code with the CSD was compared to source code without the CSD. In each of these studies, the CSD was shown provide significant advantages in numerous code reading activities [Hendrix 2002]. In the following sections, the CSD notation is described in more detail.

8.2 CSD Program Components/Units

The CSD includes graphical constructs for the following components or program units: class, abstract class, method, and abstract method. The construct for each component includes a unit symbol, a box notation, and a combination of the symbol and box notation. The symbol notation provides a visual cue as to the specific type of program component. It has the most compact vertical spacing in that it retains the line spacing of source code without the CSD. The box notation provides a useful amount of vertical separation similar to skipping lines between components. The symbol and box notation is simply a combination of the first two. Most of the examples in this handbook use the symbol notation because of its compactness. CSD notation for program components is illustrated in the table below.

Component	Symbol Notation	Box Notation	Symbol and Box Notation
class or Ada package	{ }	{ }	{ }
abstract class	{ }	{ }	{ }
method or function or procedure	{ ; }	{ ; }	{ ; }
abstract method			

8.3 CSD Control Constructs

The basic CSD control constructs for Java are grouped in the following categories: sequence, selection, iteration, and exception handling, as described in the table below. Note, the semi-colons in the examples are placeholders for statements the language.

Sequence	<pre> ; ; ; </pre>	Sequential flow is represented in the CSD by a vertical stem with a small horizontal stem for each individual statement on a particular level of control.
Selection		
if	<pre> if (cond) ; </pre>	For selection statements, the True/False condition itself is marked with a small diamond, just as in a flow chart. The statements to be executed if the condition is true are marked by a solid line leading from the right of the decision diamond.
if..else	<pre> if (cond) ; else ; </pre>	The control path for a false condition is marked with a dotted line leading from the bottom of the diamond to another decision diamond, an else clause, a default clause, or the end of the decision statement.
if..else..if	<pre> if (cond) ; else if (cond) ; else ; </pre>	By placing the second if on the same line with the first else , the unnecessary indentation of nested if statements is avoided. However, if the deep nesting effect is desired, the second if can be placed on the line after the else.

<p>Selection (cont'd)</p> <p>switch</p>	<pre>switch(item) { case a: ; break; case b: ; break; default: ; }</pre>	<p>The semantics of the switch statement are different from those of if statements. The <i>expr</i> (of integral type: int, char) is evaluated, and then control is transferred to the case label matching the result or to the default label if there is no match. If a break statement is placed at the end of the sequence within a case, control passes “out” (as indicated by the arrow) and to the end of the switch statement after the sequence is executed. Notice the similarity of the CSD notation for the switch and if statements when the break is used in this conventional way. The reason for this is that, although different semantically, we humans tend to process them the same way (e.g., if <i>expr</i> is not equal to case 1, then take the false path to case 2 and see if they are equal, and so on). However, the break statement can be omitted as illustrated next.</p>
<p>switch (when break is omitted)</p>	<pre>switch (expr) { case 1: ; break; case 2: ; ; case 3: ; ; case 4: ; ; }</pre>	<p>When the break statement is omitted from end of the sequence within a case, control falls through to the next case. In the example at left, case 1 has a break statement at the end of its sequence, which will pass control to the end of the switch (as indicated by the arrow).</p> <p>However, case 2, case 3, and case 4 do not use the break statement. The CSD notation clearly indicates that once the flow of control reaches case 2, it will also execute the sequences in case 3 and case 4. The diamonds in front of case 3 and case 4 have arrows pointing to each case to remind the user that these are entry points for the switch. When the break statement precedes the next case (as in case 1), the arrows are unnecessary.</p>

Iteration		
while loop (pre-test)	<pre>while (cond) { ; }</pre>	<p>The CSD notation for the <i>while</i> statement is a loop construct represented by the double line, which is continuous except for the small gap on the line with the <i>while</i>. The gap indicates the control flow can exit the loop at that point or continue, depending on the value of Boolean condition. The sequence within the <i>while</i> will be executed zero or more times.</p>
for loop (discrete)	<pre>for (i=0 ; i<j ; i++) { ; }</pre>	<p>The <i>for</i> statement is represented in a similar way. The <i>for</i> statement is designed to iterate a discrete number of times based on an index, test expression, and index increment. In the example at left, the <i>for index</i> is initialized to 0, the <i>condition</i> is $I < j$, and the <i>index increment</i> is $i++$. The sequence within the <i>if</i> will be executed zero or more times.</p>
do loop (post-test)	<pre>do { ; } while (cond) ;</pre>	<p>The <i>do</i> statement is similar to the while except that the loop condition is at the end of the loop instead of the beginning. As such, the body of the loop is guaranteed to execute at least once.</p>
break in loop	<pre>while (cond) { ; if (cond) break ; ; }</pre>	<p>The <i>break</i> statement can be used to transfer control flow out of any loop (<i>while</i>, <i>for</i>, <i>do</i>) body, as indicated by the arrow, and down to the statement past the end of the loop. Typically, this would be done in conjunction with an <i>if</i> statement. If the <i>break</i> is used alone (e.g., without the <i>if</i> statement), the statements in the loop body beyond the <i>break</i> will never be executed.</p>

Iteration (cont'd) continue	<pre>do { ; if (cond) continue; ; } while (cond);</pre>	The <i>continue</i> statement is similar to the break statement, but the loop condition is evaluated and if true, the body of the loop body is executed again. Hence, as indicated by the arrow, control is not transferred out of the loop, but rather to top or bottom of the loop (<i>while</i> , <i>for</i> , <i>do</i>).
Exception Handling	<pre>try { ; } catch(E) { ; } finally { ; }</pre>	In Java, the control construct for exception handling is the <i>try..catch</i> statement with optional <i>finally</i> clause. In the example at left, if stmt1 generates an exception E, then control is transferred to the corresponding <i>catch</i> clause. After the catch body is executed, the <i>finally</i> clause (if present) is executed. If no exception occurs in the try block, when it completes, the <i>finally</i> clause (if present) is executed.
With a return	<pre>try { ; ; return; } catch(E) { ; } finally { ; }</pre>	<p>The <i>try..catch</i> statement can have multiple <i>catch</i> clauses, one for each exception to be handled.</p> <p>By definition, the <i>finally</i> clause is always executed not matter how the <i>try</i> block is exited. In the example at left, a <i>return</i> statement causes flow of control to leave the try block. The CSD indicates that flow of control passes to the finally clause, which is executed prior to leaving the <i>try</i> block. The CSD uses this same convention for <i>break</i> and <i>continue</i> when these cause a <i>try</i> block to exited.</p> <p>When try blocks are nested and <i>break</i>, <i>continue</i>, and <i>return</i> statements occur at the different levels of the nesting, the actual control flow can become quite counterintuitive. The CSD can be used to clarify the control flow.</p>

8.4 CSD Templates

In Figure 8-3, the basic CSD control constructs, described above, are shown in the CSD window. These are generated automatically based on the text in the window. In addition to being typed or read from a file, the text can be inserted from a list of templates by selecting **Templates** on the CSD window tool bar.

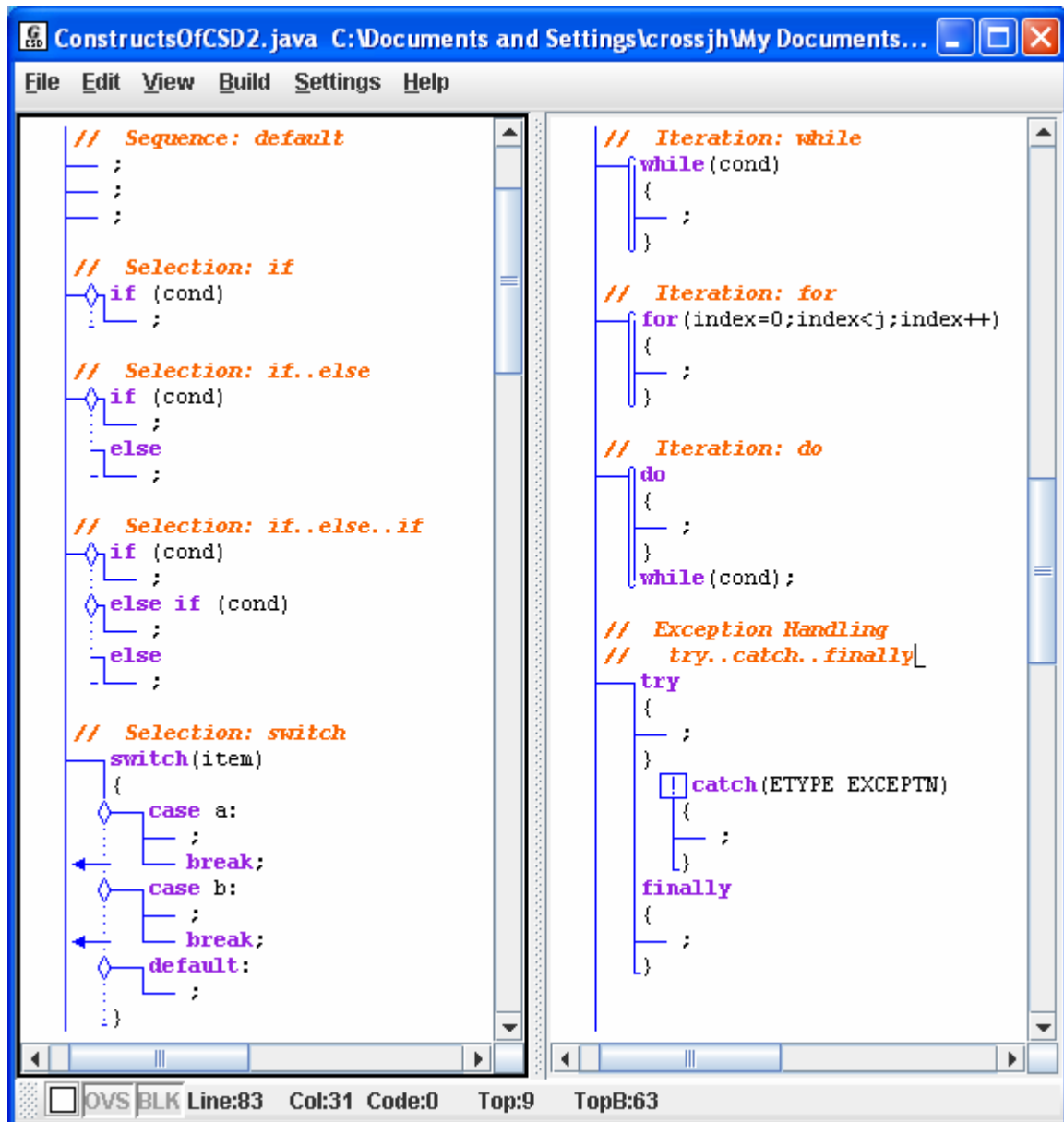


Figure 8-3. CSD Control Constructs generated in CSD Window

8.5 Hints on Working with the CSD

The CSD is generated based on the source code text in the CSD window. When you click **View – Generate CSD** (or press **F2**), jGRASP parses the source code based on a grammar or syntax that is slightly more forgiving than the Java compiler. If your program will compile okay, the CSD should generate okay as well. However, the CSD may generate okay even if your program will not compile. Your program may be syntactically correct, but not necessarily semantically correct. CSD generation is based on the syntax of your program.

Enter code in syntactically correct chunks - To reap the most benefit from using the CSD when entering a program, you should take care to enter code in syntactically correct chunks, and then regenerate the CSD often. If an error is reported, it should be fixed before you move on. If the error message from the *generate* step is not sufficient to understand the problem, compile your program and you will get a more complete error message.

“Growing a program” is described in the table below. Although the program being “grown” does nothing useful, it is both syntactically and semantically correct. More importantly, it illustrates the incremental steps that should be used to write your programs.

Step	Code to Enter	After CSD is generated
1. We begin by entering the code for a Java <i>class</i> . Note, the file should be saved with the name of the class, which in this case is MyClass.	<pre>public class MyClass { }</pre>	<pre>public class MyClass { }</pre>
2. Now, inside the class, we enter the text for a <i>method</i> called myMethod, and then re-generate the CSD by pressing F2 .	<pre>public class MyClass { myMethod() { } }</pre>	<pre>public class Hello { myMethod() { } }</pre>

<p>3. Next, inside myMethod, we enter a while loop with an empty statement, and then re-generate the CSD by pressing F2.</p>	<pre>public class MyClass { myMethod() { while (true) { ; } } }</pre>	<pre>public class MyClass { myMethod() { while (true) { ; } } }</pre>
--	---	---

8.6 Reading Source Code with the CSD

The CSD notation for each of the control constructs has been carefully designed to aid in reading and scanning source code. While the notation is meant to be intuitive, there are several reading strategies worth pointing out, especially useful with deeply nested code.

<p>Reading Sequence</p> <p>The visualization of sequential control flow is as follows. After statement s(1) is executed, the next statement is found by scanning down and to the left along the solid CSD stem. While this seems trivial, its importance becomes clearer with the if statement and deeper nesting.</p>	<pre>s(1); s(2); s(3);</pre>
<p>Reading Selection</p> <p>Now combining the <i>sequence</i> with <i>selection (if.. else)</i>, after s(1), we enter the if statement marked by the diamond. If the condition is <i>true</i>, we follow the solid line to s(2). After s(2), we read down and to the left (passing through the dotted line) until we reach the next statement on the vertical stem which is s(4). If the condition is <i>false</i>, we read down the dotted line (the false path) to the else and then on to s(3). After s(3), again we read down and to the left until we reach the next statement on the stem which is s(4).</p>	<pre>s(1); if (cond) s(2); else s(3); s(4);</pre>

<p style="text-align: center;">Reading Selection with Nesting</p> <p>As above, after s(1), we enter the <i>if</i> statement and if cond1 and cond2 are true, we follow the solid lines to s(2). After s(2), we read down and to the left (passing through both dotted lines) until we reach to the next statement on the stem which is s(4). If the cond1 is <i>false</i>, we read down the dotted line (the false path) to s(4). If cond2 is false, we read down the dotted line to the <i>else</i> and then on to s(3). After s(3), again we read down and to the left until we reach to the next statement on the stem which is s(4).</p>	<pre>s(1); if (cond1) if (cond2) s(2); else s(3); s(4);</pre>
<p style="text-align: center;">Reading Selection with Even Deeper Nesting</p> <p>If cond1, cond2, and cond3 are true, we follow the solid lines to s(2). Using the strategy above, we immediately see the next statement to be executed will be s(7).</p> <p>If cond1 is true but cond2 is false, we can easily follow the flow to either s(4) or s(5) depending on the cond4.</p> <p>If s(4) is executed, we can see immediately that s(7) follows.</p> <p>In fact, from any statement, regardless of the level of nesting, the CSD makes it easy to see which statement is executed next.</p>	<pre>s(1); if (cond1) if (cond2) if (cond3) s(2); else s(3); else if (cond4) s(4); else s(5); else s(6); s(7);</pre>

Reading without the CSD

It should be clear from the code at right that following the flow of control without the CSD is somewhat more difficult.

For example, after `s(3)` is executed, `s(7)` is next. With the CSD in the previous example, the reader can tell this at a glance. However, without the CSD, the reader may have to read and reread to ensure that he/she is seeing the indentation correctly.

While this is a simple example, as the nesting becomes deeper, the CSD becomes even more useful.

In addition to saving time in the reading process, the CSD aids in interpreting the source code correctly, as seen in the examples that follow.

```
s(1);  
if (cond1)  
    if (cond2)  
        if (cond3)  
            s(2);  
        else  
            s(3);  
    else  
        if (cond4)  
            s(4);  
        else  
            s(5);  
else  
    s(6);  
s(7);
```


Reading Correctly with the CSD	
<p>Consider the fragment at right with s(1) and s(2) in the body of the <i>if</i> statement.</p> <p>After the CSD is generated, the reader can see how the compiler will interpret the code, and add the missing braces.</p>	<pre>s(1); if (cond) s(2); s(3);</pre> <pre>s(1); if (cond) s(2); s(3);</pre>
<p>Here is another common mistake made glaring by the CSD.</p> <p>Most likely, the semi-colon after the condition was unintended. However, the CSD shows what there rather than what was intended.</p>	<pre>if (cond); s(2); s(3);</pre> <pre>if (cond); s(2); s(3);</pre>
<p>Similarly, the CSD provides the correct interpretation of the <i>while</i> statement.</p> <p>Missing braces . . .</p>	<pre>while (cond) s(2); s(3);</pre> <pre>while (cond) s(2); s(3);</pre>
<p>Similarly, the CSD provides the correct interpretation of the <i>while</i> statement.</p> <p>Unintended semi-colon . . .</p>	<pre>while (cond); s(2); s(3);</pre> <pre>while (cond); s(2); s(3);</pre>

As a final example of reading source code with the CSD, consider the following program, which is shown with and without the CSD. *FinallyTest* illustrates control flow when a **break**, **continue**, and **return** are used within **try** blocks that each have a **finally** clause. Although the flow of control may seem somewhat counterintuitive, the CSD should make it easier to interpret this source code correctly.

First read the source code without the CSD. Recall that by definition, the **finally** clause is always executed not matter how the **try** block is exited. Refer to the output if you need a hint. The output for *FinallyTest* is as follows:

```
finally 1
i 0
finally 2
i 1
finally 2
finally 3
```

***Try-Finally* with break, continue, and return statements**

```
public class FinallyTest {

    public static void main(String[] args) {
        b:
        try {
            break b;
        }
        finally {
            System.out.println("finally 1");
        }

        try {
            for(int i = 0; i < 2; i++) {
                System.out.println("i " + i);
                try {
                    if(i == 0) {
                        continue;
                    }
                    if(i < 0)
                        continue;

                    return;
                }
                finally {
                    System.out.println("finally 2");
                }
            }
        }
        finally {
            System.out.println("finally 3");
        }
    }
};
```

```
public class FinallyTest {

    public static void main(String[] args) {
        b:
        try {
            break b;
        }
        finally {
            System.out.println("finally 1");
        }

        try {
            for(int i = 0; i < 2; i++) {
                System.out.println("i " + i);
                try {
                    if(i == 0) {
                        continue;
                    }
                    if(i < 0)
                        continue;

                    return;
                }
                finally {
                    System.out.println("finally 2");
                }
            }
        }
        finally {
            System.out.println("finally 3");
        }
    }
};
```

In our experience, this code is often misinterpreted when read without the CSD, but understood correctly when read with the CSD.

References

[Cross 1998] J. H. Cross, S. Maghsoodloo, and T. D. Hendrix, "Control Structure Diagrams: Overview and Initial Evaluation," *Journal of Empirical Software Engineering*, Vol. 3, No. 2, 1998, 131-158.

[Hendrix 2002] T. D. Hendrix, J. H. Cross, S. Maghsoodloo, and K. H. Chang, "Empirically Evaluating Scaleable Software Visualizations: An Experimental Framework," *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, May 2002, 463-477.

9 Viewers for Data Structures

Viewers for objects and primitives were introduced in tutorials *3 Getting Started with Objects*, *6 The Object Workbench*, and *7 The Integrated Debugger*. In this tutorial, we introduce a family of “Presentation” views for data structures. A presentation view is a conceptual view similar to what one might find in a textbook with the added benefit of being dynamically updated as the user steps through the program.

Since objects only exist during execution, the viewers are tightly integrated with the workbench and debugger, and they can be opened for any primitive, object, or field of an object in the Debug or Workbench tabs. To use viewers with the debugger, (1) set a breakpoint, (2) run Debug (🐞), (3) after the instance has been created, drag it from the Debug tab; (4) step through program and observe the object in viewer. To use viewers with the workbench, (1) create an instance from the UML window or from the CSD window (■), (2) drag the instance from the Workbench tab; (3) invoke methods on the instance and observe the object in the viewer.

jGRASP currently includes *Presentation* views for many of the classes in the Java Collections Framework, including `ArrayList`, `LinkedList`, `TreeMap`, and `HashMap`. These are the focus of sections 9.1 – 9.4. Section 9.5 introduces a general *Presentation* view provided by the *Data Structure Identifier (DSI)* when it automatically detects a linked list or binary tree structure. This is an animated view that shows nodes being added and deleted from the data structure. The user can control the speed of the animation, the width of the nodes, and the scale of the overall visualization. This view is also configurable with respect to the structure mappings and the fields to display. Section 9.6 provides a summary of current and planned views supported by jGRASP.

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

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

9.1 Opening Viewers

9.2 Setting the View Options

9.3 Selecting Among Views

9.4 Presentation Views for `LinkedList`, `HashMap`, and `TreeMap`

9.5 Presentation (Data Structure Identifier) View

9.5.1 `LinkedListExample.java`

9.5.2 `BinaryTreeExample.java`

9.5.3 Configuring Views generated by the Data Structure Identifier

9.6 Summary of Views

9.7 Exercises

9.1 Opening Viewers

Let's begin by opening one of the example programs that comes with the jGRASP installation. After you have started jGRASP, use the Browse tab to navigate to the jGRASP\examples\Tutorials folder. If you have been working with the examples in the "Hello" or "PersonalLibrary" folders, you'll need to go up one level in the Browse tab by clicking the up arrow. [Note that if you don't have write access to this folder (e.g., if you are working on a computer in the university lab), you should copy this folder to a personal directory. This will need to be done using a file browser rather than jGRASP.] In the Tutorials folder you should find a folder called ViewerExamples. Open this folder by double-clicking on the folder name, and you should see a file called *ArrayListExample1.java*. Open this file by double-clicking the file name (Figure 9-1).

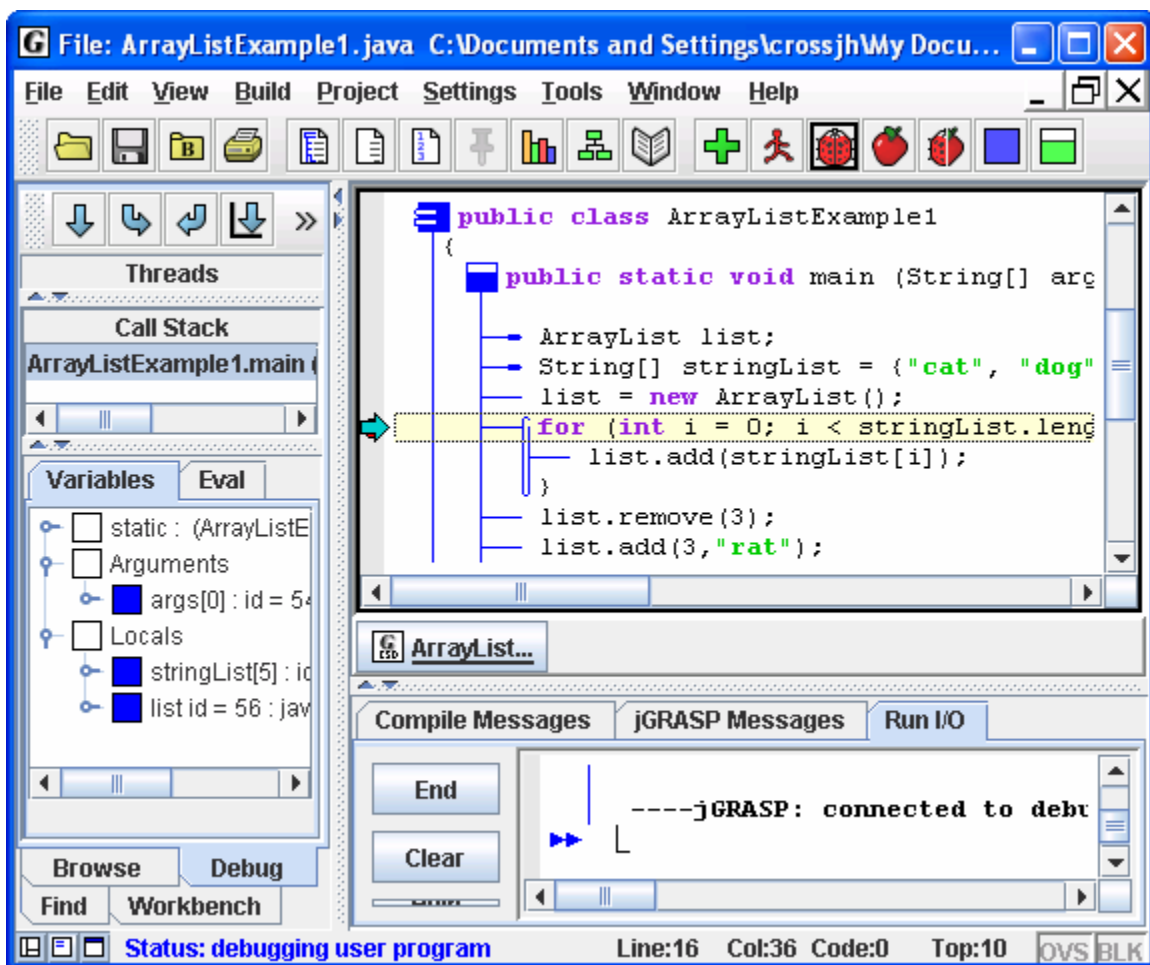





Figure 9-1. *ArrayListExample1.java*

A quick review of the program shows that it creates an `ArrayList` called `list` and adds strings to it from an array called `stringList`. Compile the program by clicking the green plus . Since the viewers are for visualizing objects and primitives as the program

executes, let's set a breakpoint on the first line of the *for* statement and start the debugger by clicking . Figure 9-1 shows the program stopped at the *for* statement. At this point in the program, the *list* object has been created, and it is shown in the Debug Variables tab. However, no elements have been added to *list*.

A separate **Viewer** window can be opened for any object (or field of an object) in the debug tab (or on the workbench). The easiest way to open a viewer is to left-click on an object and drag it from the debug tab (or workbench) to the location where you want the viewer to open. When you start to drag the object, a viewer symbol should appear to indicate a viewer is being opened. [Note: You can also open a viewer by right-clicking on the list object and selecting either **View by Name** or **View by Value**.] Let's left click on *list* and drag it from the Debug tab. When you release the left mouse button, the viewer should open. Figure 9-2 shows a viewer opened on *list* before any elements have been added.

To add elements to *list*, step through the program by clicking the “Step” button  on the Debug tab. Since the viewer is updated on each step, you should see the elements being added to the list. Red text indicates a change or, in this case, a new element. Figure 9-3 shows the view of *list* after going through the loop three times.

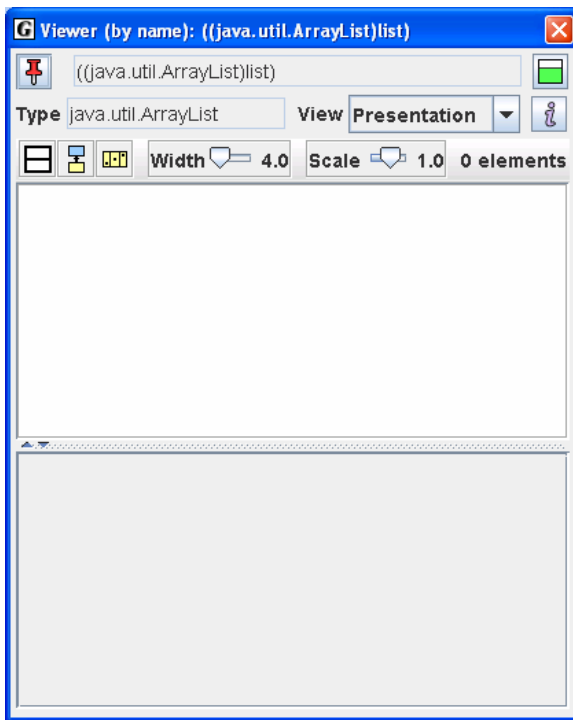


Figure 9-2. View of *list* with no elements

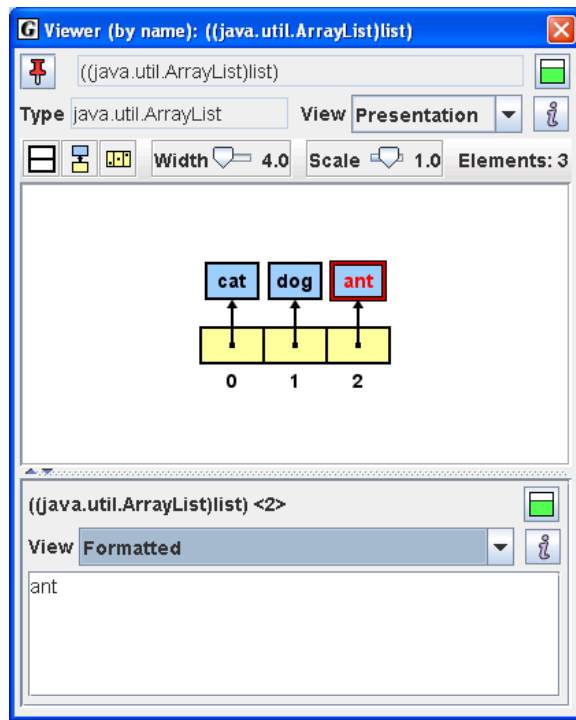


Figure 9-3. View of *list* with 3 elements

Each jGRASP viewer provides one or more subviews. When an element is selected, indicated by a red border, a view of the element itself is shown in the subview. Figure 9-3 shows “ant” selected in the ArrayList view. Since “ant” is a String, the subview is a String viewer for which the *formatted* view is the default.

9.2 Setting the View Options

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


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

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

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

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

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

Figure 9-4 indicates the location of the button or slider for each of these view options. Click on each of these and notice the change in the view. The ArrayList is shown vertically after the window is partitioned horizontally. The **View** drop down list and the Information button  is also indicated below.

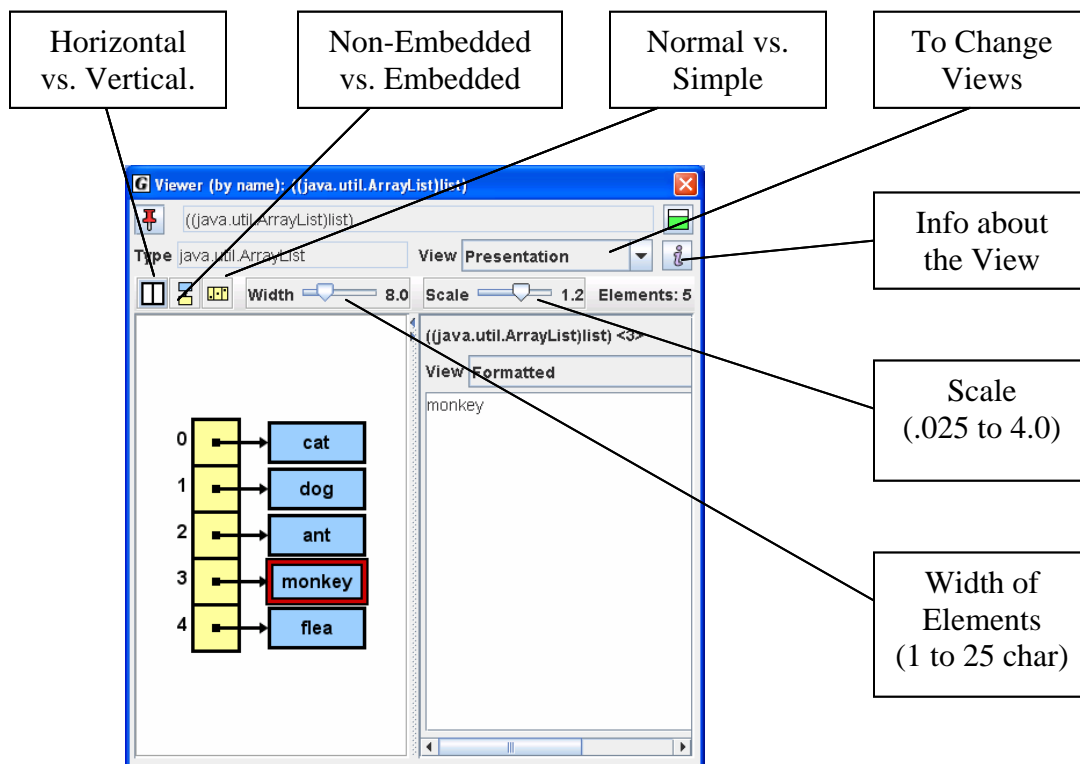



Figure 9-4. View of *list* in vertical mode, width set to 8, Scale set to 1.2, *monkey* selected and shown in subview

9.3 Selecting Among Views

Each viewer and subviewer provides one or more views among which you may select. Let's take a closer look at our ArrayList of Strings example. The *Presentation* view is the default for ArrayList and the other classes in the Java Collections Framework. Three other views are *Basic*, *Collection Elements*, and the Data Structure Identifier (see section 9.5). Figure 9-5 shows these options on the drop-down list (combo box) for ArrayList. Notice each of the subviews has its own *View* drop down list. Clicking the Information button () provides a description of the view selected in the list.

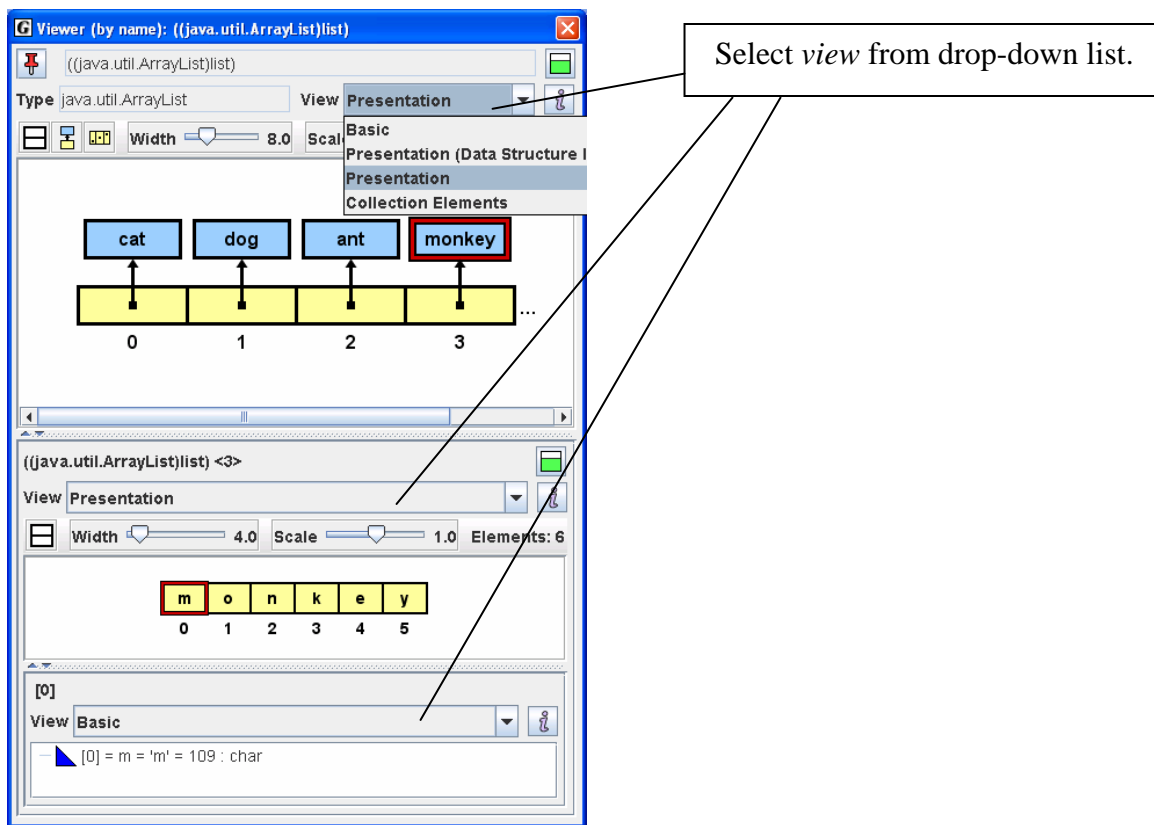


Figure 9-5. Selecting the *Collection*

Figure 9-6 shows the viewer after the *Collection Elements* view is selected. If *list* has many elements, this may be a more appropriate view than the *Presentation* view. The *Collection Elements* view was specifically designed to handle larger numbers of elements efficiently. As the number of elements increases, additional navigational controls appear on the viewer for moving about in the *ArrayList*. Notice that two subviews are also shown in Figure 9-6. When element 0 (indicated by “<0> = **cat**”) is selected, a subview for *String* opens below the main view. The view for *String* is set to *Presentation* in the figure; the default for *String* is *Formatted*. When the ‘c’ in “cat” is selected, a second subview is opened for the primitive type *char*, for which *Basic* is the default view. The *char* view in the figure is set to *Detail*, which shows additional information about ‘c’ including its value in hexadecimal, octal, and binary.

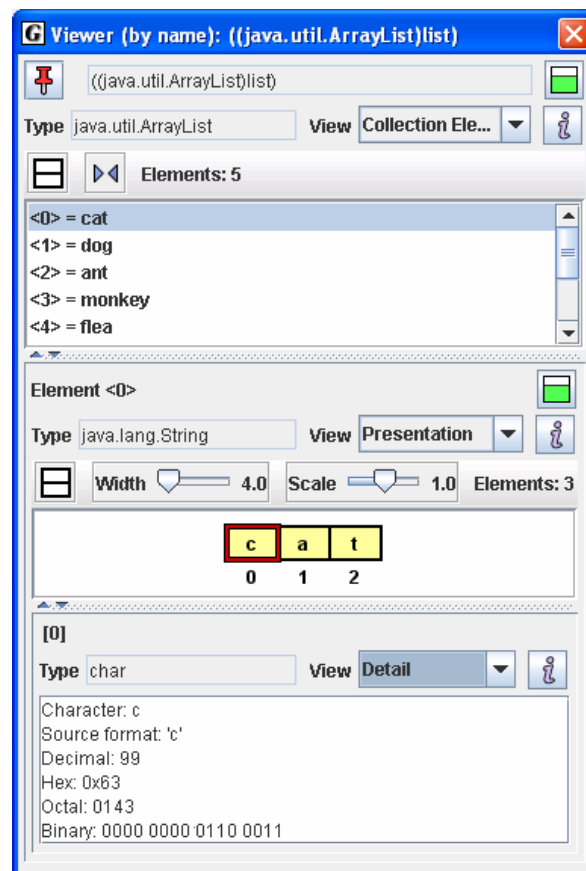




Figure 9-6. The *Collection Elements* view of *list* with two subviews: *Presentation* view for *String* “cat” and *Detail* view for *char* ‘c’

9.4 Presentation Views for LinkedList, HashMap, and TreeMap

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

In the Browse tab, locate CollectionsExample.java, and double-click on it to open it in the CSD window. Compile the program by clicking the green plus . Set a breakpoint on any executable statement in the program. Now start the debugger by clicking . Figure 9-7 shows the program stopped at a breakpoint on the line in the inner loop that adds an element to myArrayList. Notice that prior to the breakpoint, the variables myArrayList, myLinkedList, myHashMap, and myTreeMap were declared and their respective instances were created. With the program stopped at the breakpoint, we can open viewers for each of the variables listed in the Debug tab.

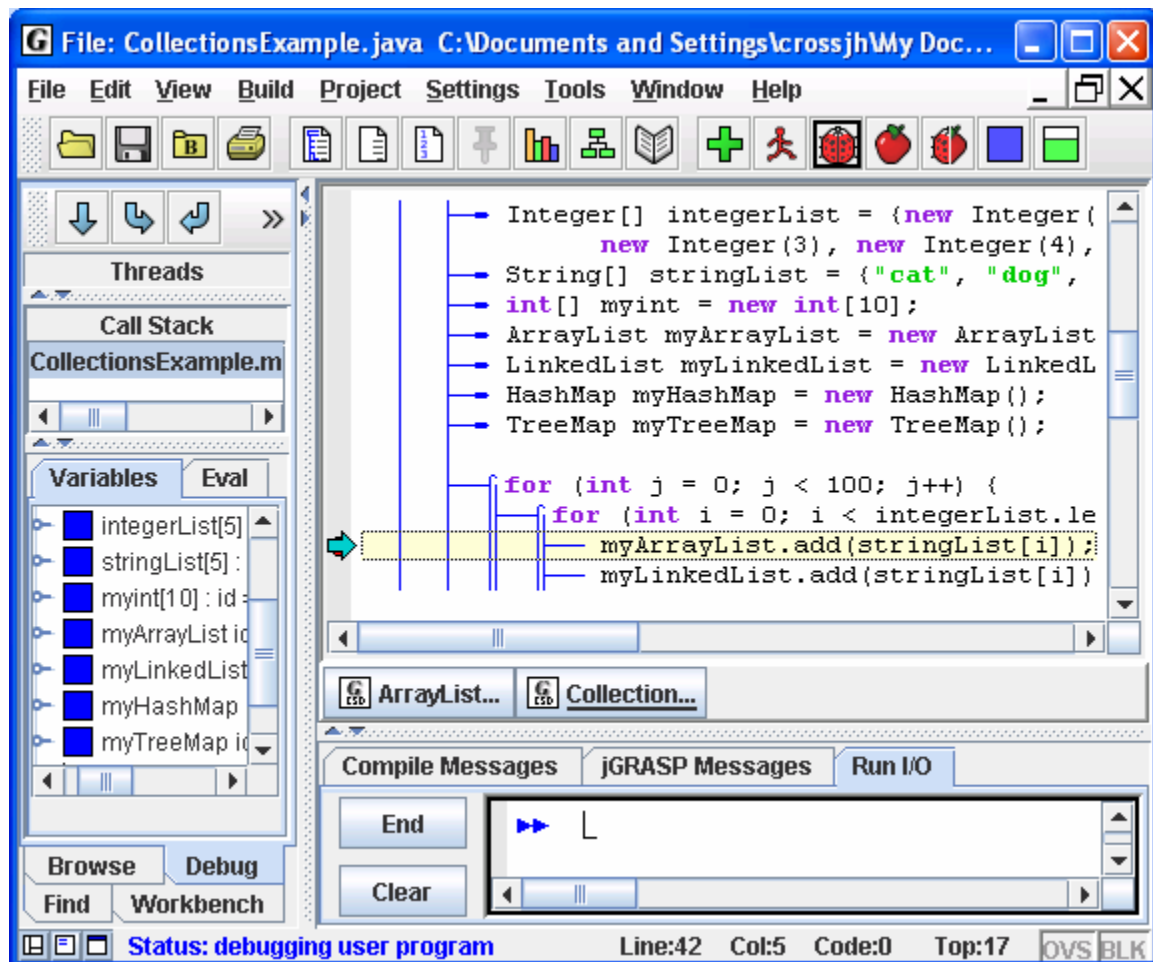


Figure 9-7. CollectionsExample.java stopped at a breakpoint

Below, Figure 9-8 shows a viewer set to *Presentation* view opened on myLinkedList after three elements have been added to it. Notice that myLinkedList is a doubly-linked list with a header node. The element “Ant” is selected in the main view and shown in the subview in *Presentation* view for the String Class. In this view, the ‘a’ in “ant” is selected, and the character subview is shown set to the *Detail* view.

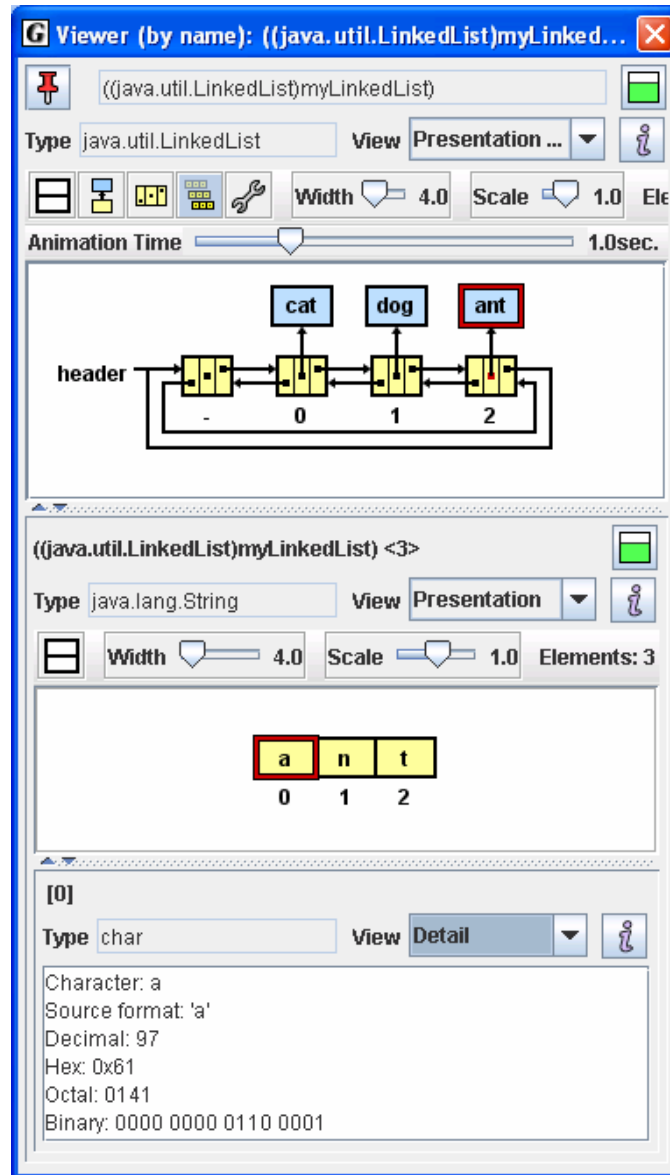


Figure 9-8. View of *myLinkedList* after three elements have been added

Figure 9-9 shows a viewer opened on the variable `myHashMap` after three elements have been added. Entry 7 is selected, indicated by the red border, and its *Basic* view is shown in the subview with fields: `key`, `value`, `hash`, and `next`. As elements are added to the `HashMap`, it is useful to use the `Scale` slider to zoom in and out on the structure so that the “topology” of its elements can be seen.

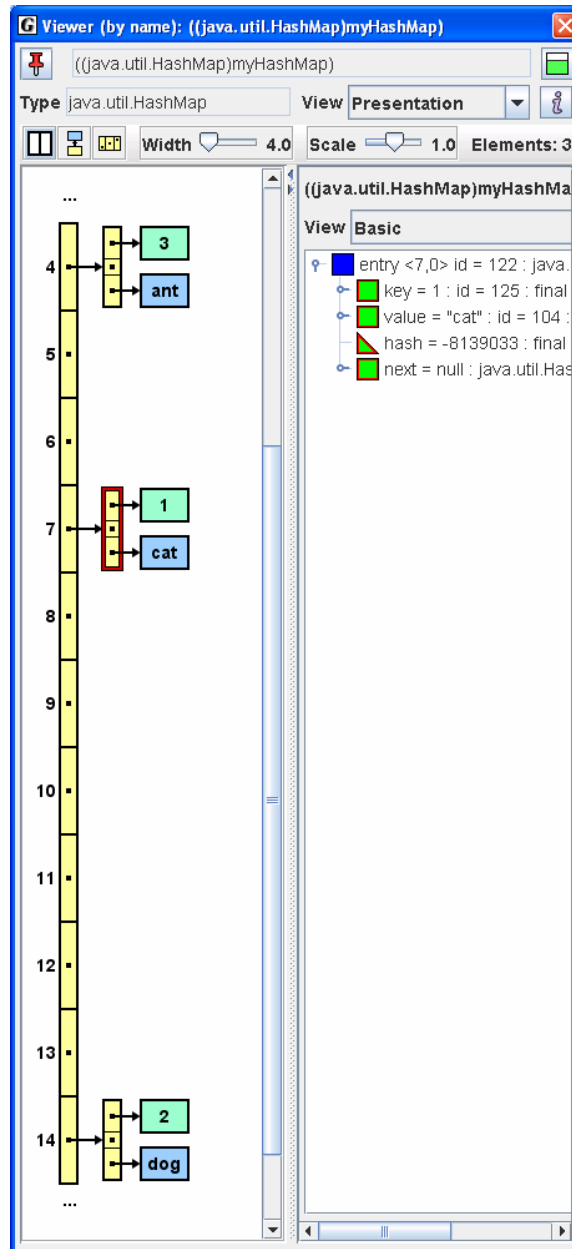


Figure 9-9. View of *myHashMap* after three elements have been added

Figure 9-10 shows a viewer opened on myTreeMap after five elements have been added. TreeMap uses a *Red-Black* tree as its underlying storage structure. The *Presentation* view for TreeMap has two subviews: one for the *Key* and one for the *Value*. In the figure below, the node with *Key* = “dog” and *Value* = 2 has been selected. In the String subview for *Key*, the view has been set to *Char Array* so “dog” is shown in the equivalent of the *presentation* view for an array of char.

Value is an instance of Integer and its view has been set to *Detail*. For *Value* = 2, we can see the value represented in decimal, hexadecimal, octal, and binary.

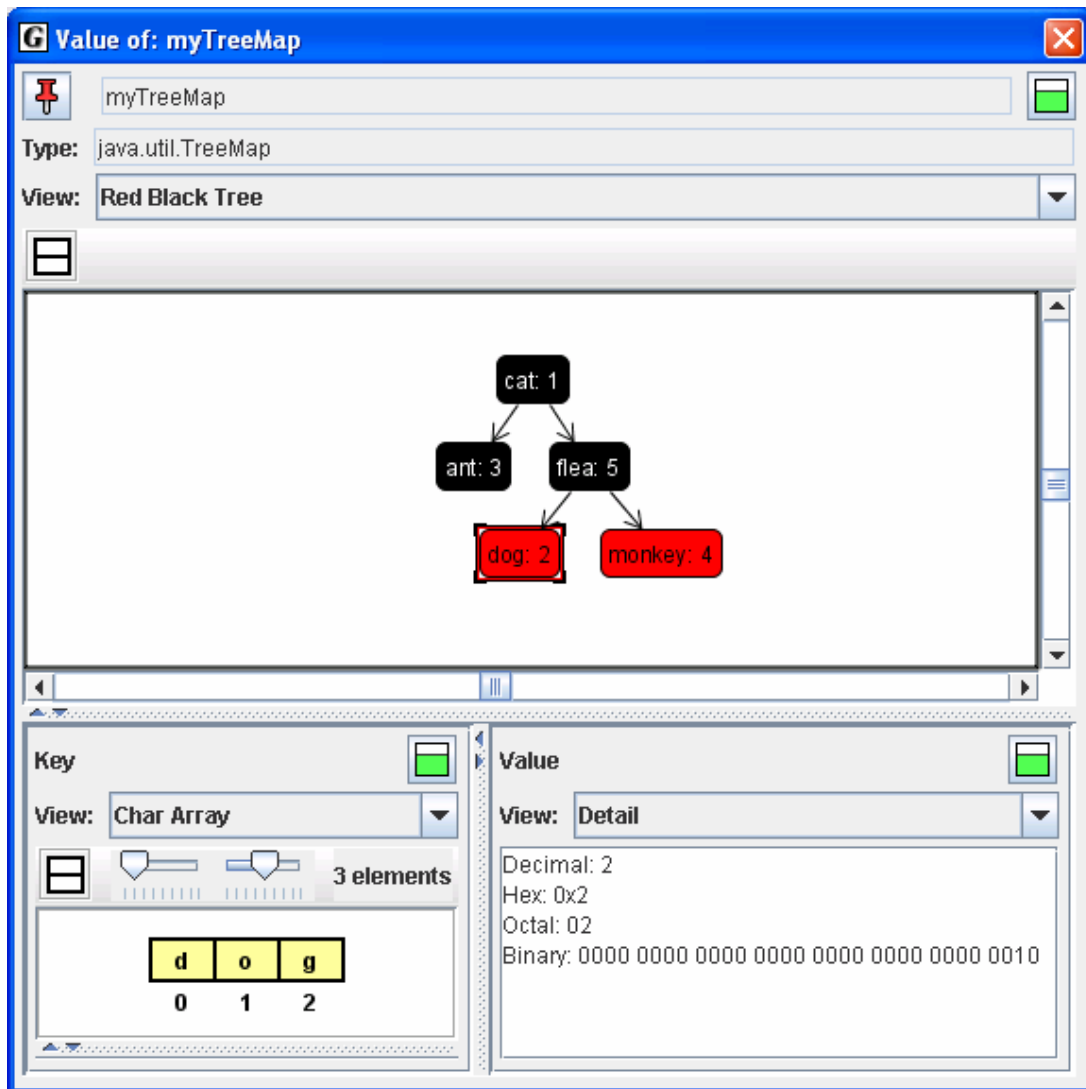


Figure 9-10. View of myTreeMap after five elements have been added

9.5 Presentation (Data Structure Identifier) View






Now we turn our attention to the verifying *Presentation* view generated by the *Data Structure Identifier*. This viewer has the following features: (1) automatically attempts to identify linked list and binary tree structures; (2) verifies relevant links; (3) displays nodes referenced by local variables; (4) provides animation for the insertion and deletion of nodes; and (5) can be configured by the user.

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

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

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

9.5.1 LinkedListExample.java

In the Browse tab, navigate to the *ViewerExamples* directory and open the file *LinkedListExample.java* by double-clicking on it. Generate the CSD, and then compile the program by clicking  on the toolbar. Set a breakpoint  in the left margin on a line inside the inner loop (e.g., on the line where *list* is declared and a new *LinkedList* object is created). Now click the Debug button  on the toolbar. Figure 9-11 shows the program after it has stopped at the breakpoint prior to creating an instance of *LinkedList* called *list*. Click Step  on the controls at the top of the Debug tab. When *list* is created, you should see it in the Variables tab of the Debug window. Now open a viewer on *list* by selecting and dragging *list* from the Debug window. Add two elements to the linked list by stepping () through the inner loop twice.

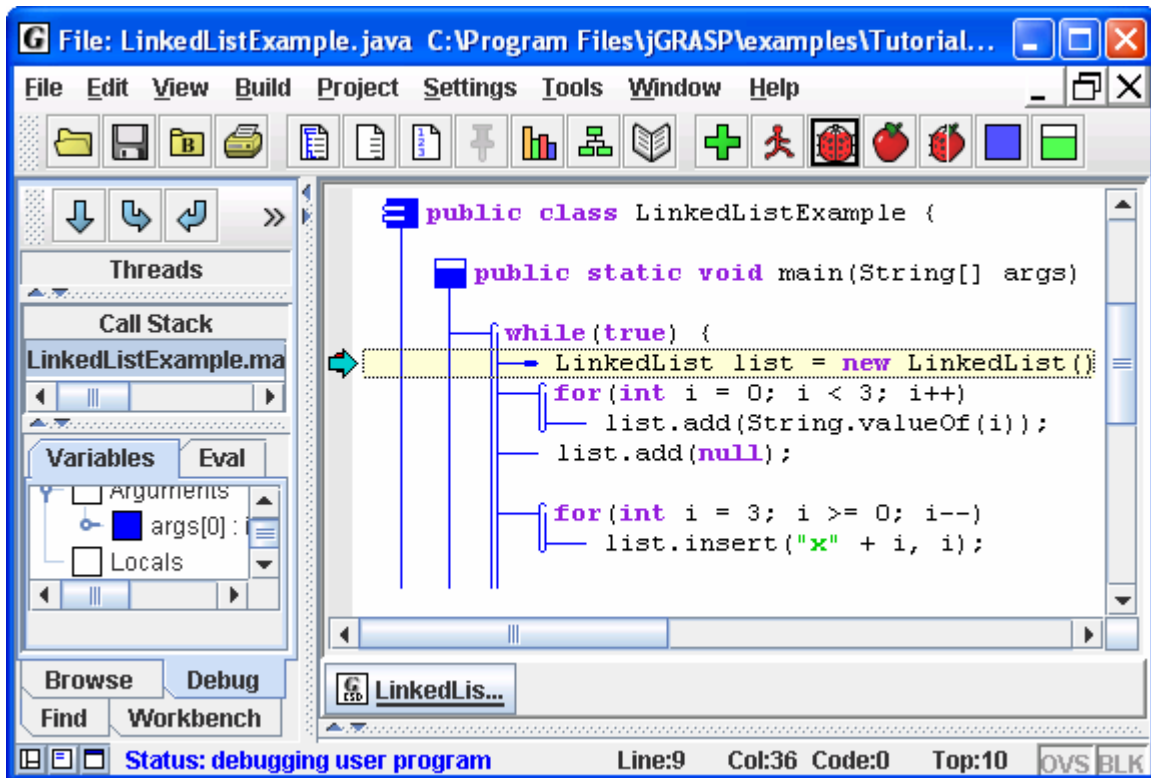




Figure 9-11. *LinkedListExample.java* stopped at a breakpoint

Figure 9-12 shows a view of *list* after two elements have been added. Note that the viewer is set to either *Linked List Example* view or *Presentation (Data Structure Identifier)* view. The *Linked List Example* view is a special viewer written specifically for the *LinkedList* class in the *jgraspvex* package. As described above, the *Presentation (Data Structure Identifier)* view, shown in Figure 9-13, can be used for any class the represents a binary tree or linked list data structure. To switch between the views, use the View drop down list. After experimenting with the two views, change the View to *Presentation (Data Structure Identifier)* by selecting this on the drop down list as shown in Figure 9-13.

Now you are ready to see the animation of a local node being added to the linked list. You need to step into to the *add* method by clicking the *Step in* button  at the top of the debug tab. Each time you click , the program will either step into the method indicated or step to the next statement if there is no method call in the statement. Figure 9-14 shows *list* after *node.next* for the new node has been set to *head*. Figure 9-15 shows *list* after *head* has been set to *node*. As you repeatedly *step in*, you should see added and inserted nodes “slide” up into *list* and removed nodes slide out of *list*. Note that the Call Stack in the Debug tab indicates the methods into which you have stepped.

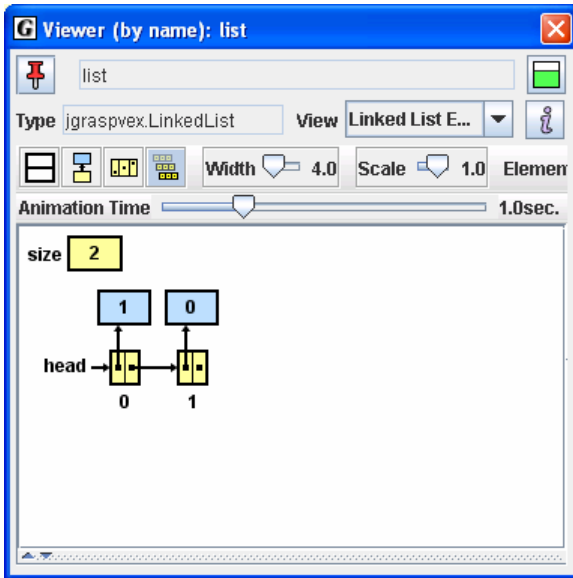


Figure 9-12. Linked List Example view of *list*

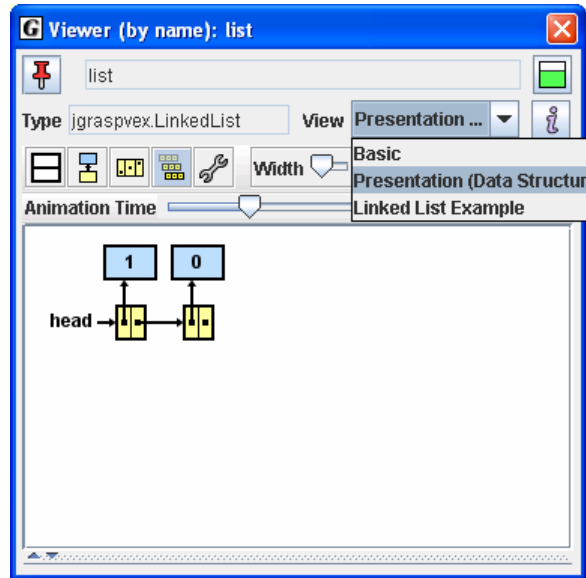


Figure 9-13. Presentation (Data Structure Identifier) view of *list*

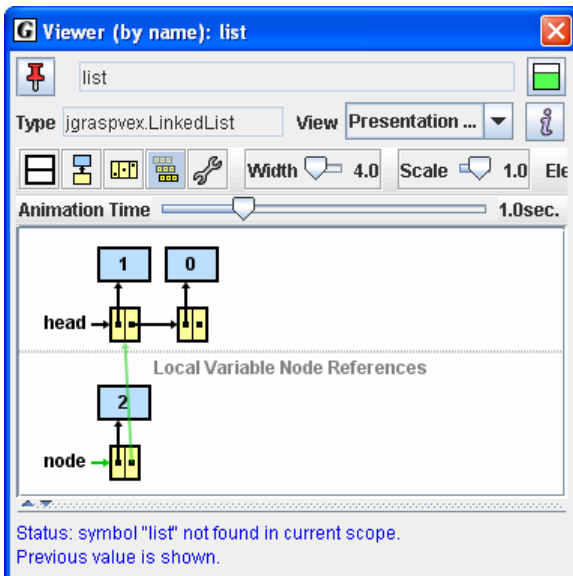


Figure 9-14. Node about to be added to *list*

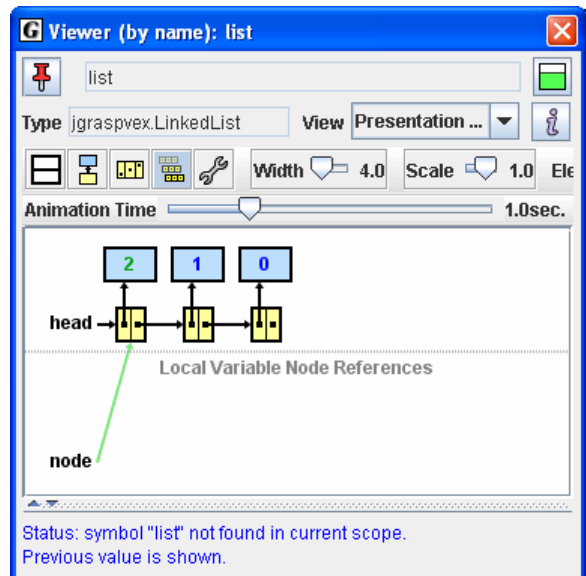





Figure 9-15. After node has been added to *list*

9.5.2 BinaryTreeExample.java

Now let's take a look at an example of another common data structure, the *binary tree*. In the Browse tab, navigate to the *ViewerExamples* directory and open the file *BinaryTreeExample.java* by double-clicking on it. After compiling it, set a breakpoint  in the left margin on a line inside the inner loop (e.g., on the line where *bt.add(..)* is called). Now click the Debug button  on the toolbar. Figure 9-16 shows the program after it has stopped at the breakpoint prior to adding any nodes to *bt*. Now open a viewer on *bt* by selecting and dragging it from the Debug window. The *Presentation (Data Structure Identifier)* is shown in subsequent figures with viewers for *bt*. Add two elements to the *bt* by stepping () through the inner loop twice.

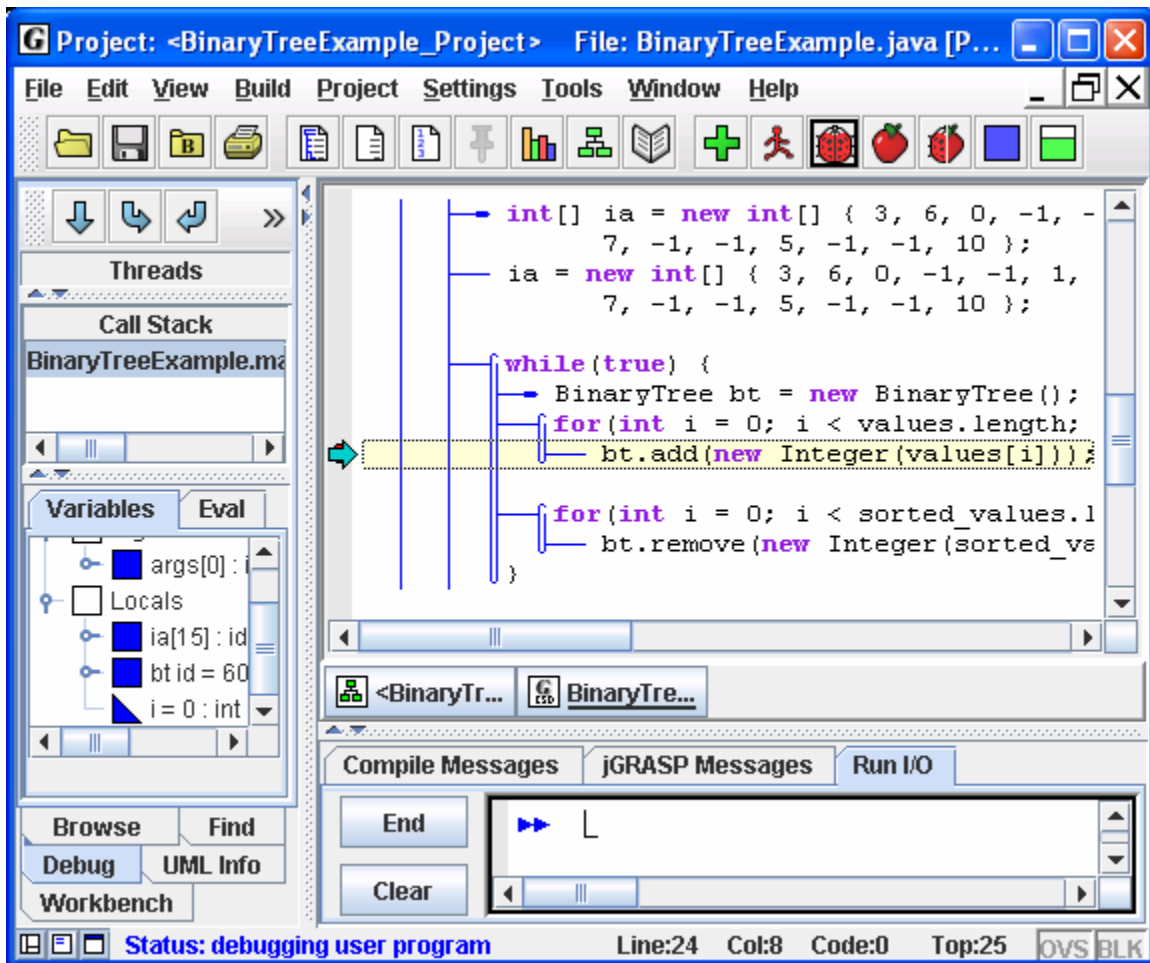




Figure 9-16. *BinaryTreeExample.java* stopped at a breakpoint

Now you are ready to see the animation of a local node being added to the linked list. You need to step into to the *add* method by clicking the *Step in* button  at the top of the debug tab. Each time you click , the program will either step into the method indicated or step to the next statement if there is no method call in the statement. Figure 9-14 shows *bt* after *root* has been passed into the *add* method as *branch*, and Figure 9-15 shows *bt* after *branch.left* has been set to *node*. As you repeatedly *step in*, you should see added and inserted nodes “slide” up into *bt* and removed nodes slide out of *bt*. Note that the Call Stack in the Debug tab indicates the methods into which you have stepped.

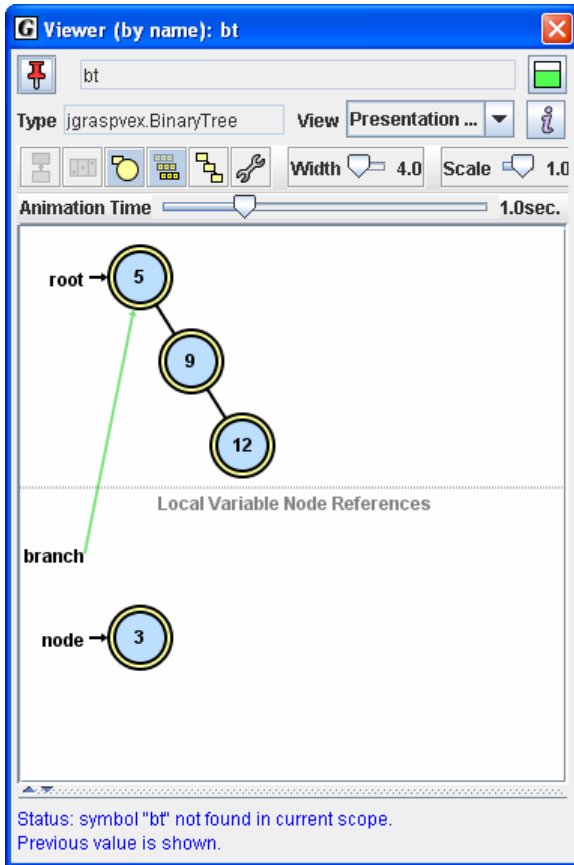


Figure 9-19. Binary tree example as node is about to be added to *bt*

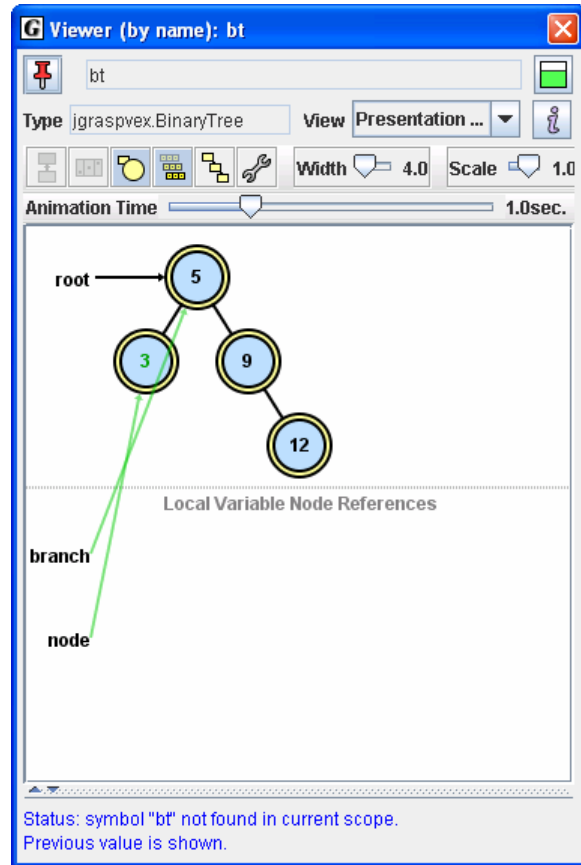



Figure 9-20. Binary tree example after node is added to *bt*

9.5.3 Configuring Views generated by the Data Structure Identifier

The Data Structure Identifier is using a set of heuristics in its attempt to determine if the object for which a view is being opened is a linked list or binary tree structure. Since the view it provides is only a guess, some additional configuration may be needed in order to attain an appropriate *Presentation* view. Consider the binary tree example from above, shown again in Figure 9-21. Figure 9-22 to 9-24 show the result of (1) clicking the *Configure* button  (located to the left of the *Width* slider), (2) selecting the *Fields Display* tab, (3) selecting the *size* field, and (4) clicking OK or Apply.

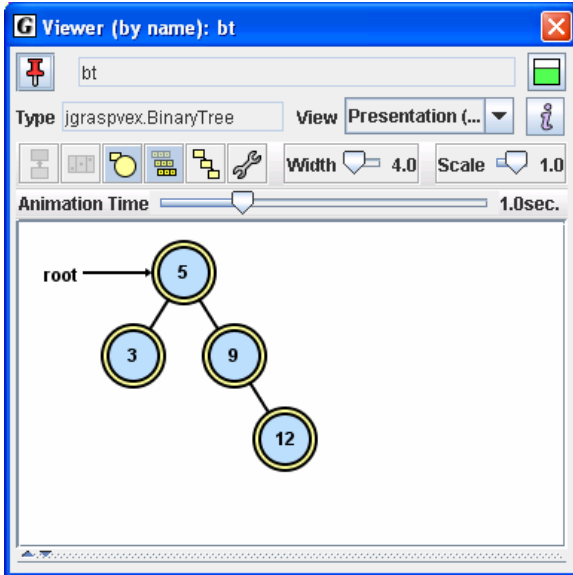


Figure 9-21. Binary tree example

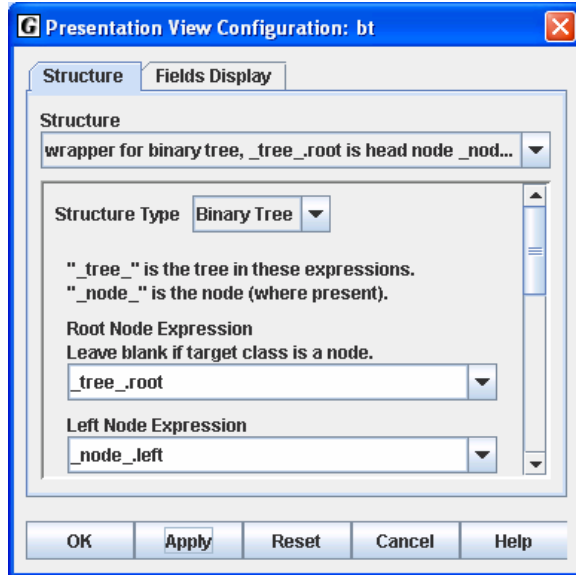


Figure 9-22. Configuration dialog (after clicking *Configure* button  on viewer)

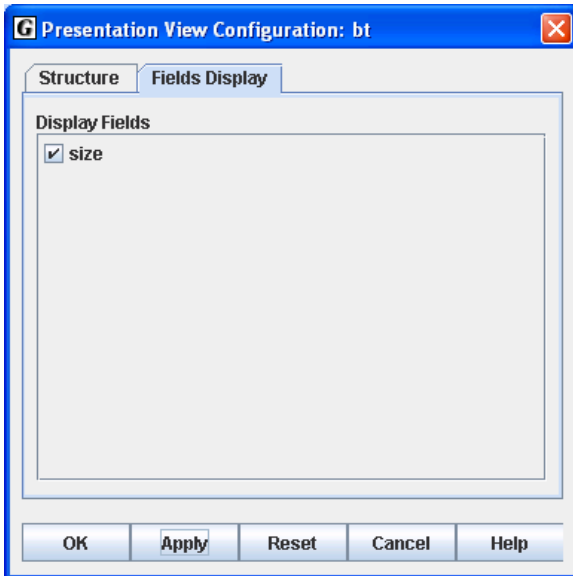


Figure 9-23. Configuration dialog with *Fields Display* tab and *size* selected

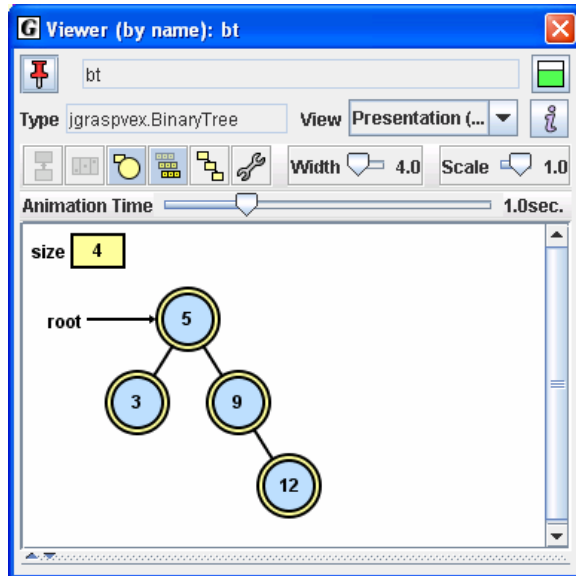


Figure 9-24. Binary tree example after OK (Apply) on Configuration dialog

Deciding which fields to display as above is the most common configuration operation to perform on the view provided by the Data Structure Identifier. For some data structures one (or more) of the fields is treated as a formal part of the conceptual diagram itself. For example, the binary tree example has two fields, *size* and *root*, and the viewer treats *root* as part of the diagram, but not *size*. Only the fields that are not part of the diagram are listed on the *Fields Display* tab.

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

Continuing with the binary tree example above, Figure 9-23 shows that the *Structure Type* for *bt* has been changed from Binary Tree to Linked List. Figure 9-24 shows the data structure after the configuration change has been applied. Notice that transparent red arrows represent links that are not quite correct for a linked list.

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

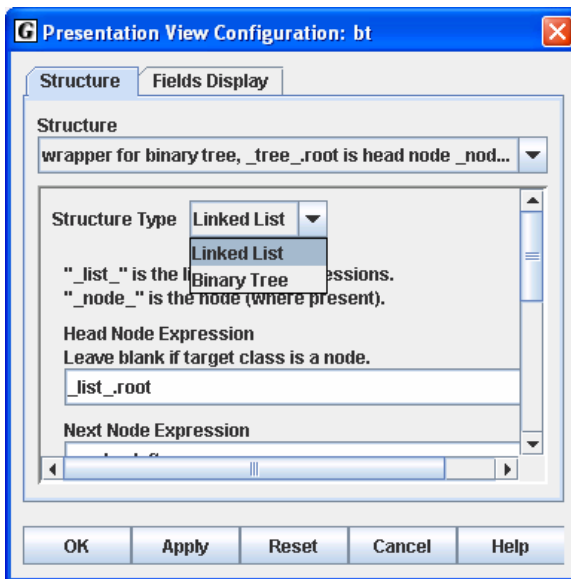


Figure 9-23. Changing structure type of *bt* from Binary Tree to Linked List

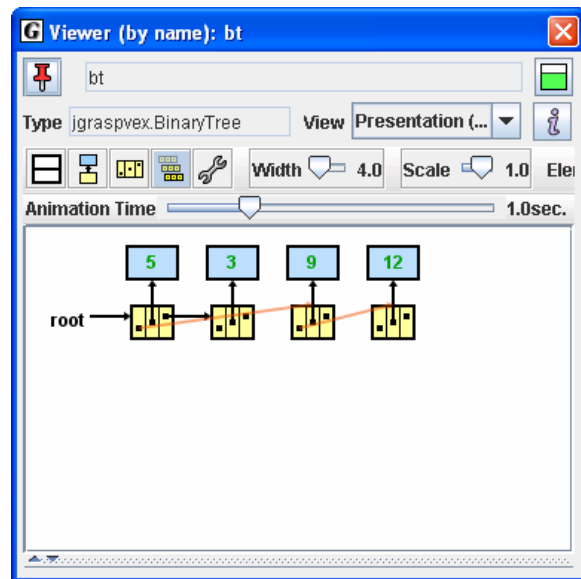


Figure 9-24. Binary tree instance *bt* shown as a linked list with transparent red links indicating it is not a linked list

9.6 Summary of Views

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

General Description of Views

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

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

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

array, *String*, *ArrayList*, *LinkedList*, *HashMap*, *TreeMap*

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

Version 1.8.4 (Aug 06) – current release

linked lists, binary trees



Version 1.8.5 Beta (Nov 06)

binary heap, array containers (e.g. queues), red black trees, AVL trees

Version 1.8.5 Beta (TBD)

hash tables, graphs, multi-way trees



9.7 Exercises

- (1) Open *CollectionsExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops the breakpoint, open a viewer on instances of one or more of the following, then step through the program:
 - a. array
 - b. ArrayList
 - c. LinkedList
 - d. TreeMap
 - e. HashMap
- (2) Continuing with the program from above, let's use the Auto-Step feature of the jGRASP Debugger. With the program stopped at a breakpoint and one or more viewers open, select *Auto Step*  on the debug control panel. Now click the *Step* button . You can control the speed of the steps with the slider bar beneath the step controls.
- (3) Create floating point variable in your program by adding the statement:

```
double myDouble = 4096.0;
```

After compiling the program, set an appropriate breakpoint, and run the program in Debug mode. Open a viewer on *myDouble* with the view set to *Detail*. The *Detail* view for float and double values shows the internal exponent and mantissa representation used for floating point numbers.

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

- (4) Open *LinkedListExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops the breakpoint, open a viewer on list, open a viewer for it. Select *Auto Step*  on the debug control panel. Now click the *Step in* button . You can control the speed of the steps with the slider bar beneath the step controls on the debug control panel. You can control the speed of the animation with the slider bar on the viewer.
- (5) Open *BinaryTreeExample.java* and repeat the task described in (4).

