The jGRASP Tutorials

Tutorials* for the jGRASP™ 1.7 Integrated Development Environment

James H. Cross II and Larry A. Barowski

Copyright © 2004 Auburn University

All Rights Reserved

March 1, 2004

DRAFT

*These tutorials are from the jGRASP Handbook.
Copyright © 2004 Auburn University

All Rights Reserved

Table of Contents

Overview of jGRASP and the Tutorials 1

1 Installing jGRASP 3

2 Getting Started 5

2.1 Starting jGRASP 6

2.2 Quick Start - Opening a Program, Compiling, and Running 7

2.3 Generating a Control Structure Diagram 9

2.4 Folding a CSD 11

2.5 Line Numbers 12

2.6 Creating a New File 13

2.7 Saving a File 16

2.8 Compiling a Program – A Few More Details 17

2.9 Running a Program - Additional Options 20

2.10 Using the Debugger 22

2.11 Opening a File – Additional Options 24

2.12 Closing a File 27

2.13 Exiting jGRASP 28

2.14 Exercises 28

2.15 Review and Preview of What's Ahead 29

3 Getting Started with Objects 30

3.1 Starting jGRASP 31

3.2 Navigating to Our First Example 32

3.3 Opening a Project and UML Window 33

3.4 The UML Window 34

3.5 Exploring the Features of the UML Window 35

3.5.1 Viewing the source code for a class 35

3.5.2 Displaying class information 35

3.5.3 Displaying Dependency Information 35

3.6 Viewing the Source Code 36

3.7 Compiling and Running the Program 37
3.8 Generating Documentation for the Project 38
3.9 Using the Object Workbench 39
3.10 Invoking a Method 41
3.11 Invoking Methods with Parameters 42
3.12 Invoking Methods on Object Fields 42
3.13 Invoking Inherited Methods 43
3.14 Running the Debugger on Invoked Methods 44
3.15 Creating Instance from the Java Class Libraries 44
3.16 Exiting the Workbench 44
3.17 Closing a Project 45
3.18 Exiting jGRASP 45
3.19 Exercises 46
4 Projects 47
4.1 Creating a Project 47
4.2 Adding files to the Project 49
4.3 Removing files from the Project 50
4.4 Generating Documentation for the Project (Java only) 51
4.5 Jar file Creation and Extraction 53
4.6 Active Project vs. Open Projects 53
4.7 Closing a Project 53
4.8 Exercises 54
5 UML Class Diagrams 55
5.1 Opening the Project 55
5.2 Generating the UML 55
5.3 Determining the Contents of the Class Diagram 58
5.4 Laying Out the UML Diagram 61
5.5 Displaying the Members of a Class 62
5.6 Displaying Dependencies Between Two Classes 63
5.7 Finding a Class in the UML Diagram 64
5.8 Opening Source Code from UML 64
5.9 Saving the UML Layout 65

5.10 Printing the UML Diagram 65
6 The Object Workbench 66
6.1 Invoking Static Methods 66
6.2 Creating an Object for the Workbench 69
6.3 Invoking a Method 71
6.4 Invoking Methods with Parameters 72
6.5 Invoking Methods on Object Fields 72
6.6 Invoking Inherited Methods 73
6.7 Running the Debugger on Invoked Methods 74
6.8 Exiting the Workbench 74
7 The Integrated Debugger 75
7.1 Preparing to Run the Debugger 75
7.2 Setting a Breakpoint 75
7.3 Running a Program in Debug Mode 76
7.4 Stepping Through a Program 77
7.5 Debugging a Program 81
8 The Control Structure Diagram (CSD) 82
8.1 An Example to Illustrate the CSD 82
8.2 CSD Program Components/Units 84
8.3 CSD Control Constructs 85
8.4 CSD Templates 89
8.5 Hints on Working with the CSD 90
8.6 Reading Source Code with the CSD 91
8.7 References 96

Overview of jGRASP and the Tutorials

jGRASP is a full-featured medium-weight integrated development environment, created specifically to provide visualizations for improving the comprehensibility of the 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 can be configured to work with almost any compiler. jGRASP, which is based on its predecessors, pcGRASP and UNIX GRASP (written in C/C++) is the latest IDE from the **GRASP** (**G**raphical **R**epresentations of **A**lgorithms, **S**tructures, and **P**rocesses) research group at Auburn University.

jGRASP currently provides for the automatic generation of two important software visualizations: the *Control Structure Diagram* (Java, C, C++, Ada, and VHDL) for source code visualization and the *UML Class Diagram* (Java) for architectural visualization. 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, which is 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 was 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 and source code editing. Source code may be edited directly as with any traditional editor. After editing, regenerating a CSD is fast and efficient and non-disruptive (approximately 5000 lines/sec). The source code can be folded based on CSD structure (method, loop, if statement, 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 **Object Workbench**, in conjunction with the UML class diagram, allows the user to create instances of classes and invoke their methods. This has proven to be an extremely useful paradigm 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*** are perhaps best utilized while using jGRASP; however, they are sufficiently detailed to be used in a stand-alone fashion (i.e., just reading them). They 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\tutorial_examples\). Users may want to copy the examples folder to their own directory 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 SDK have already been installed successfully. It is recommended for those students planning to install jGRASP and the Java SDK 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 assumes the reader will refer to the previous section as needed. It introduces projects, UML class diagrams, and the Object Workbench in jGRASP.

4. ***Projects*** – This tutorial introduces 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 is able to create a project (Tutorial 4) and understands the concept of a project.

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 teach object-oriented concepts and programming by allowing the user to create objects and invoke methods directly rather than 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.

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

1 Installing jGRASP

Currently, jGRASP is available from http://www.jgrasp.org in four versions: 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 (1) or (2) below is strongly recommended.

jGRASP exe (2.3 MB) – **Windows self-extracting exe file.** The full Java 2 SDK (J2SDK) must be installed in order to run jGRASP and compile and run Java programs.

jGRASP JRE exe (13.5 MB) – **Windows self-extracting exe file with JRE**. Since this includes a copy of the JRE, 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 J2SDK.* The *jGRASP JRE* version of jGRASP is convenient if you will be compiling programs in languages other than Java.

jGRASP pkg.tar.gz (2.1 MB) – **Mac OS X tarred and gzipped package file** (requires admin access to install). J2SDK is preinstalled on Mac OS X machines.

jGRASP (2.1 MB) – **Zip file**. After unzipping the file, refer to README file for installation instructions. *The full J2SDK must be installed in order to run jGRASP and to compile and run Java programs*.

For Windows 95/98/2000/XP - After downloading (1) or (2) 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 pressing ENTER. 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.

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. 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.

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/3.14p/winnt/(e.g., gnat-3.14p-nt.exe)

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

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

http://www.borland.com/bcppbuilder/freecompiler/cppc55steps.html

FORTRAN (GNU/Cygnus)

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

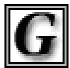
Java (J2SDK, Jikes)

http://java.sun.com/j2se/1.4/download.html

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., *J2SDK* for Java), you may need to change the Compiler Settings as by clicking on **Settings – Compiler Settings – Global (or 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 **Part 2 – Reference** of the ***jGRASP Handbook***, or see this topic in jGRASP Help.



jGRASP

You can start jGRASP by double clicking on the icon.

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 in 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 Generating a Control Structure Diagram

2.4 Folding a CSD

2.5 Line Numbers

2.6 Creating a New File

2.7 Saving a File

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

jGRASP

If you are working in a Microsoft Windows environment, you can start jGRASP by double clicking its icon on your Windows desktop. If you are working on a PC in a computer lab, you may not 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** (Project tab is combined with the Browse tab in version 1.7). The large *right pane* is for UML and CSD Windows. The *lower pane* with tabs for jGRASP messages, Compile messages, and Run Input/Output.

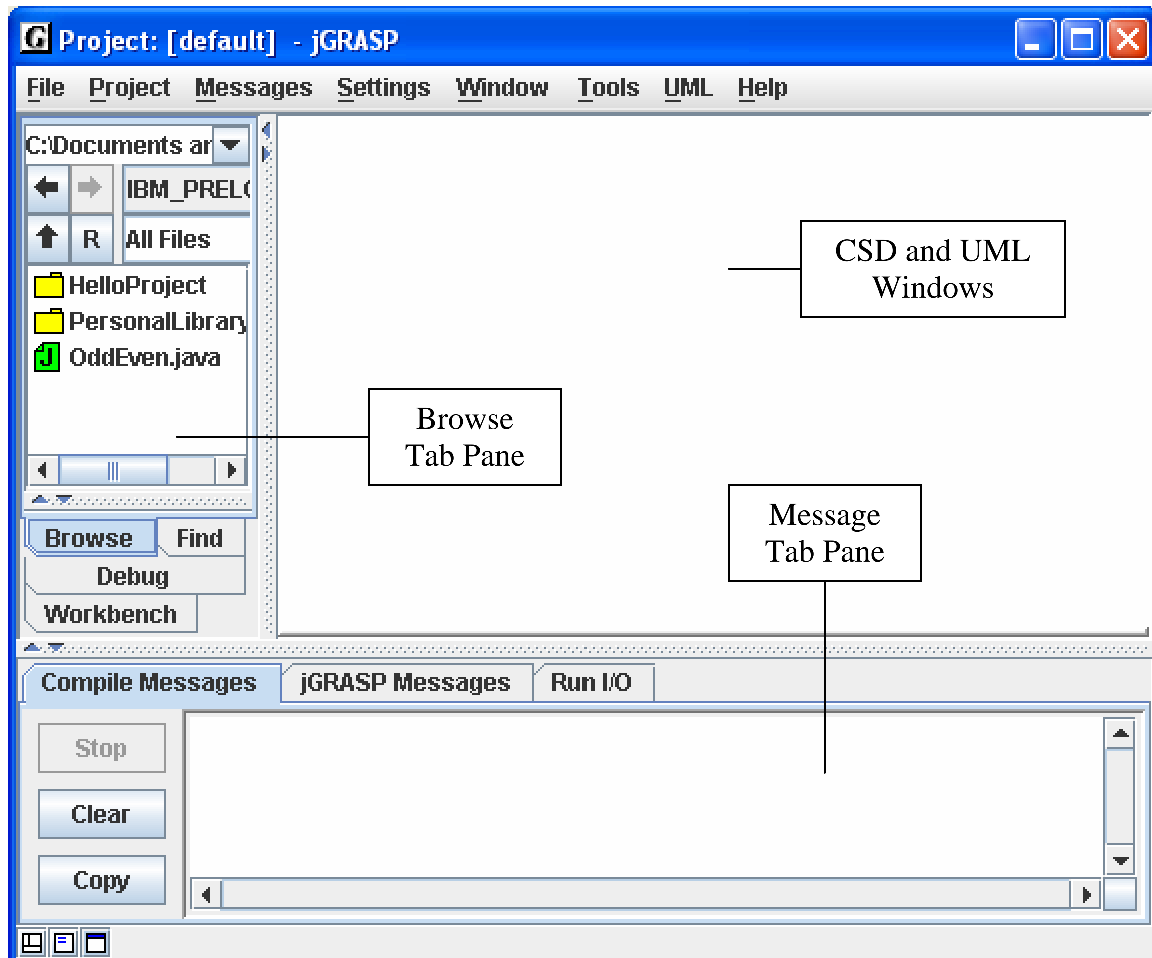


Figure 1. The jGRASP Virtual Desktop

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\tutorial_examples). If jGRASP was installed by a system administrator, you may not have write privileges for these files so you may need to copy the tutorial_examples folder to one of your personal folders (e.g., in your *My Documents* folder).

The files shown initially in the **Browse tab** will most likely be in your home directory. However, regardless of the opening default directory, you can navigate to the appropriate directory by double-clicking on a folder in the list of files or by clicking on the up-arrow as indicated in the figure below. The "**R**" refreshes the Browse pane. In the example, the Browse tab is displaying the contents of tutorial_examples.

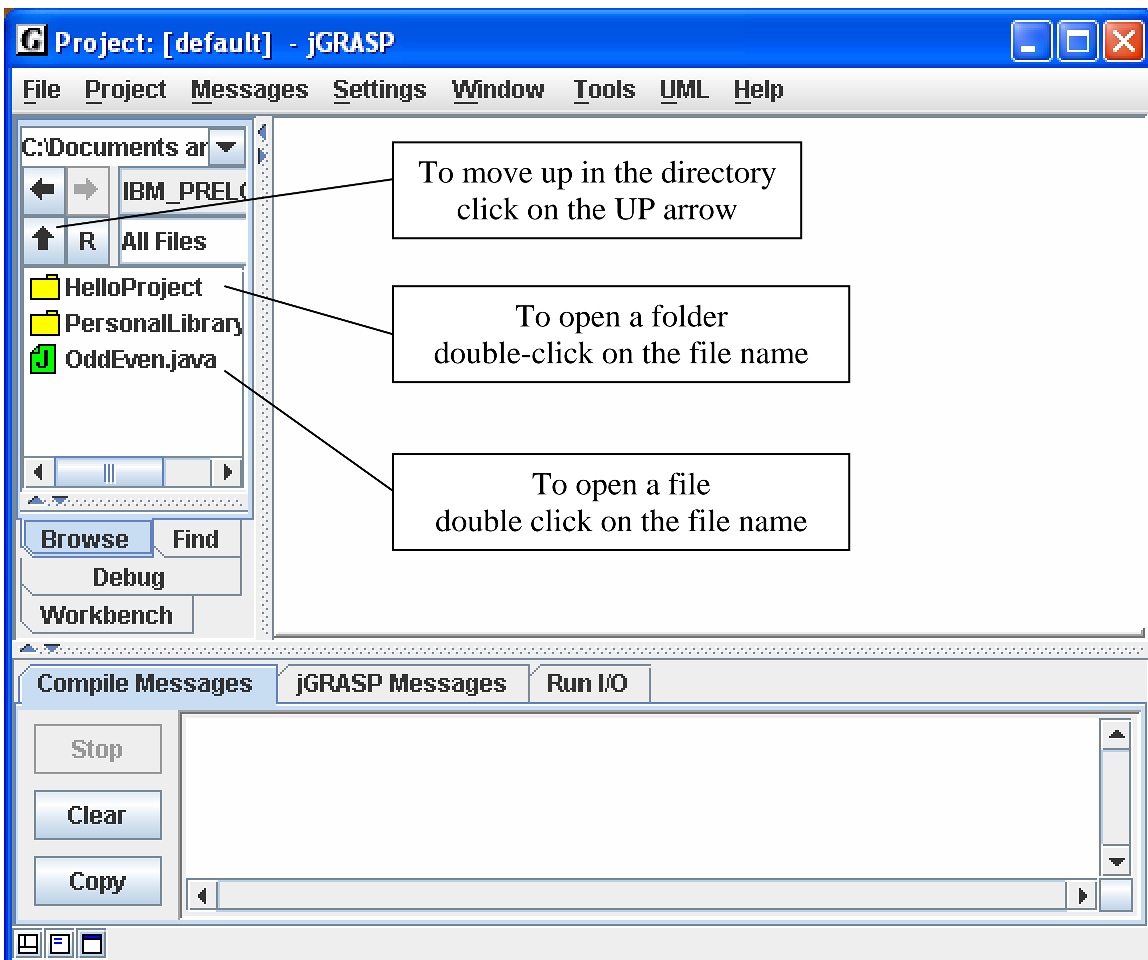


Figure 2. The jGRASP Virtual Desktop

Double-clicking on the HelloProject 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 the CSD Window has its own menu and toolbar icons across the top. 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 Compile menu, then select **Compile**. Alternatively, you can click on the Compile icon indicated by **Step 2** below. After a successful compilation (no error messages in the Run I/O tab), you are ready to run the program by clicking on the Run icon as shown in **Step 3** below, or you can click the Run menu and select **Run**. The standard input and output for your program will be in the Run I/O tab of the message pane.

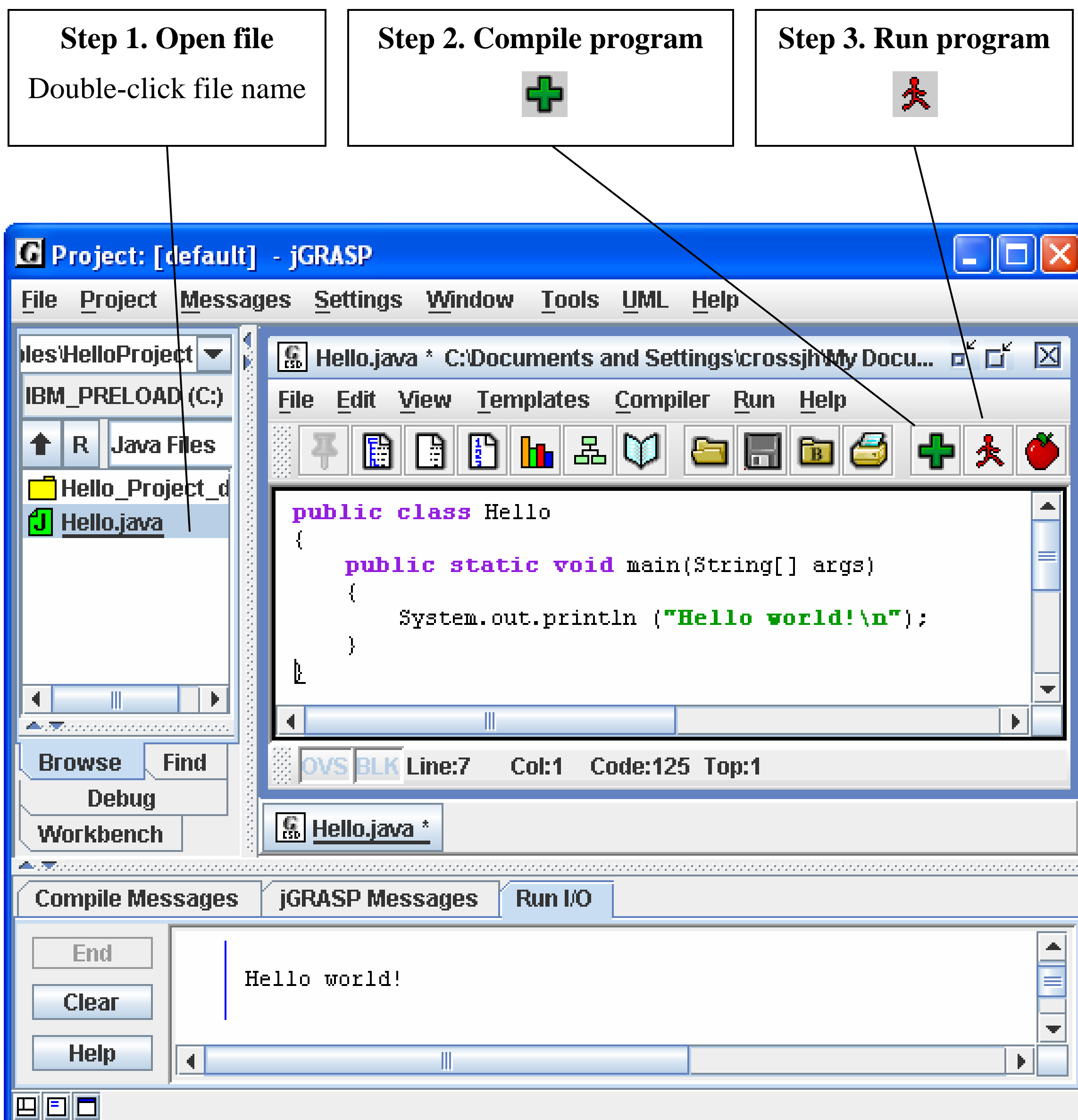


Figure 3. After loading file into CSD Window

2.3 Generating a Control Structure Diagram

You can generate a Control Structure Diagram in the CSD Window whenever you have a syntactically correct program. Generate the CSD for the program by doing <u>one</u> of the following:

Clicking the Generate CSD icon

or Clicking **View -- Generate CSD** on the menu

or Pressing F2

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.

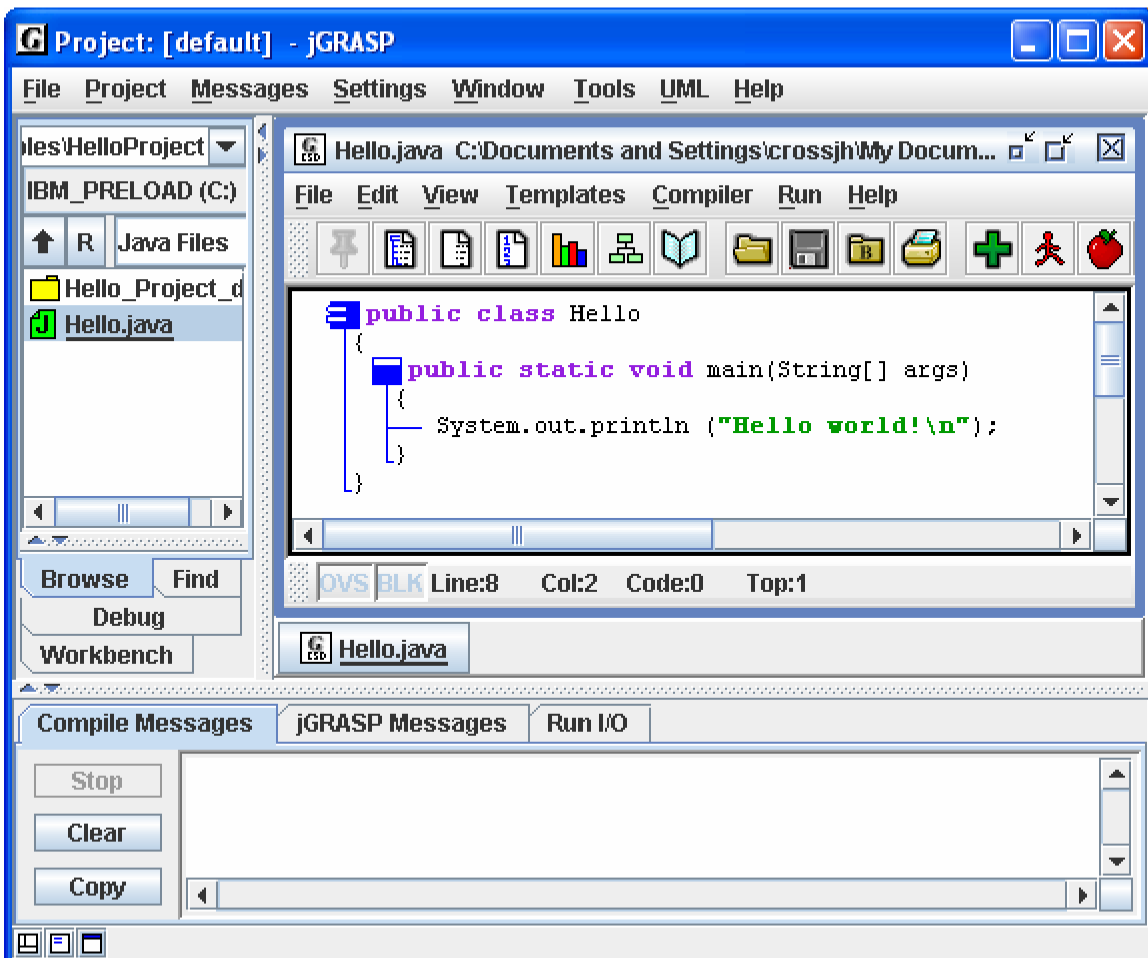


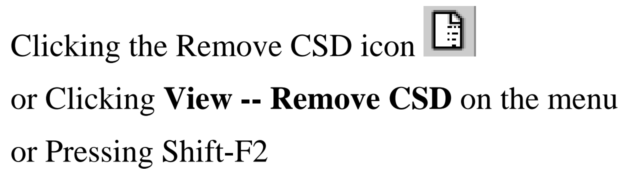
Figure 4. After CSD is generated

Otherwise, 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 5, the semi-colon was omitted at the end of the 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 one of the following:

Clicking the Remove CSD icon 

or Clicking **View -- Remove CSD** on the menu

or Pressing Shift-F2

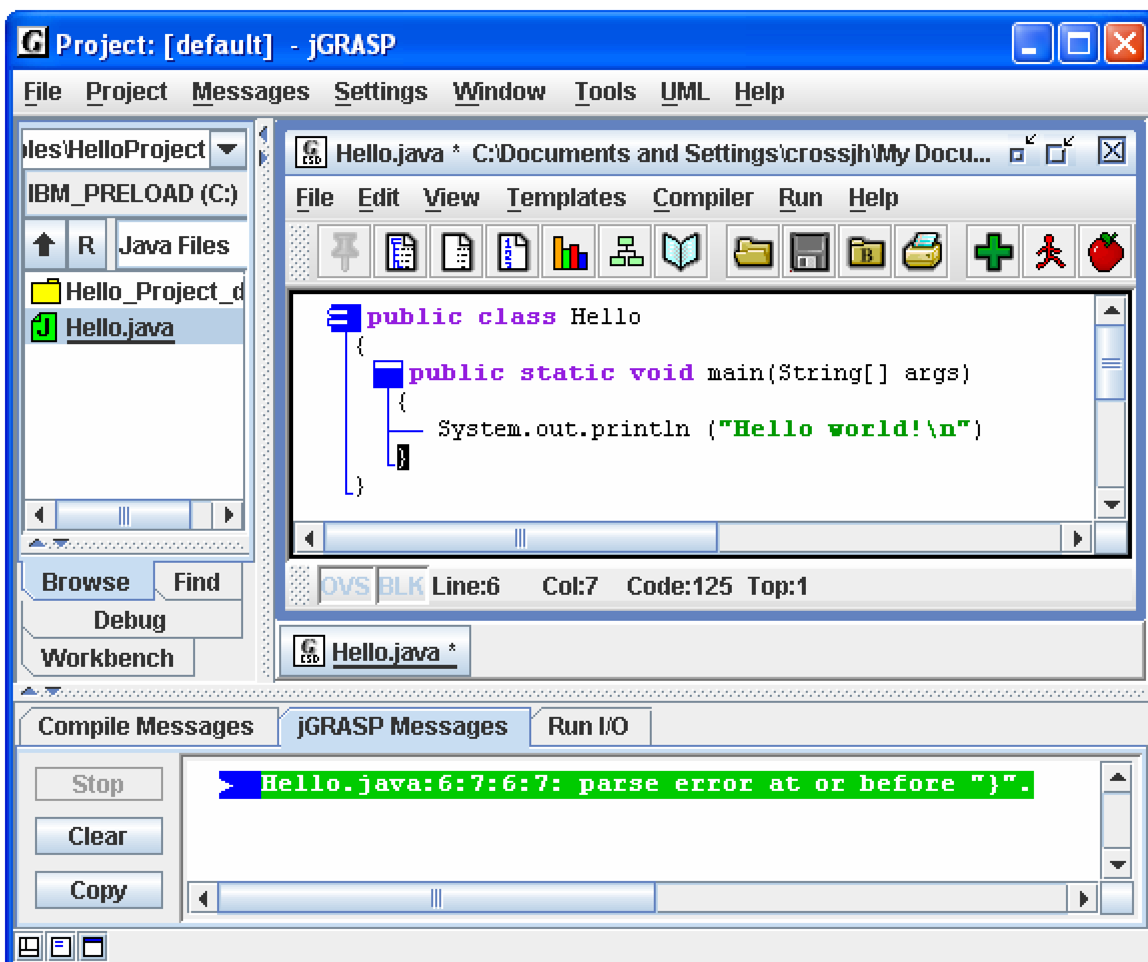


Figure 5. Syntax error detected

Remember, the purpose of using the CSD is to improve the readability of your program. While this is may not be obvious on a small simple program like the example, it should become apparent as the size and complexity of your programs increase.

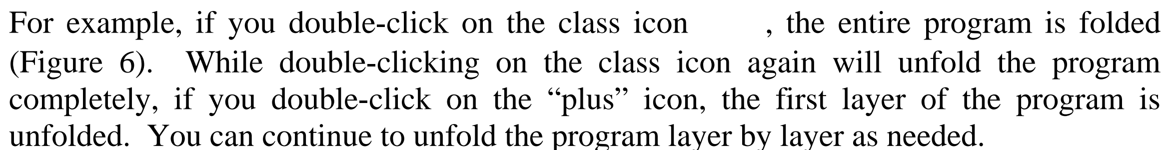
TIP: As you enter a program, try to enter it in “chucks” that are syntactically correct. For example, the following is sufficient to generate the CSD.

```
public class Hello
{
}
```

As soon as you think you have entered a syntactically correct chuck, you should generate the CSD. Not only does this update the diagram, it catches your syntax errors early.

2.4 Folding a CSD

“**Folding**” is another feature that many users find useful, especially 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 icon  , the entire program is folded (Figure 6). While double-clicking on the class icon again will unfold the program completely, if you double-click on the “plus” icon, the first layer of the program is unfolded. You can continue to unfold the program layer by layer as needed.

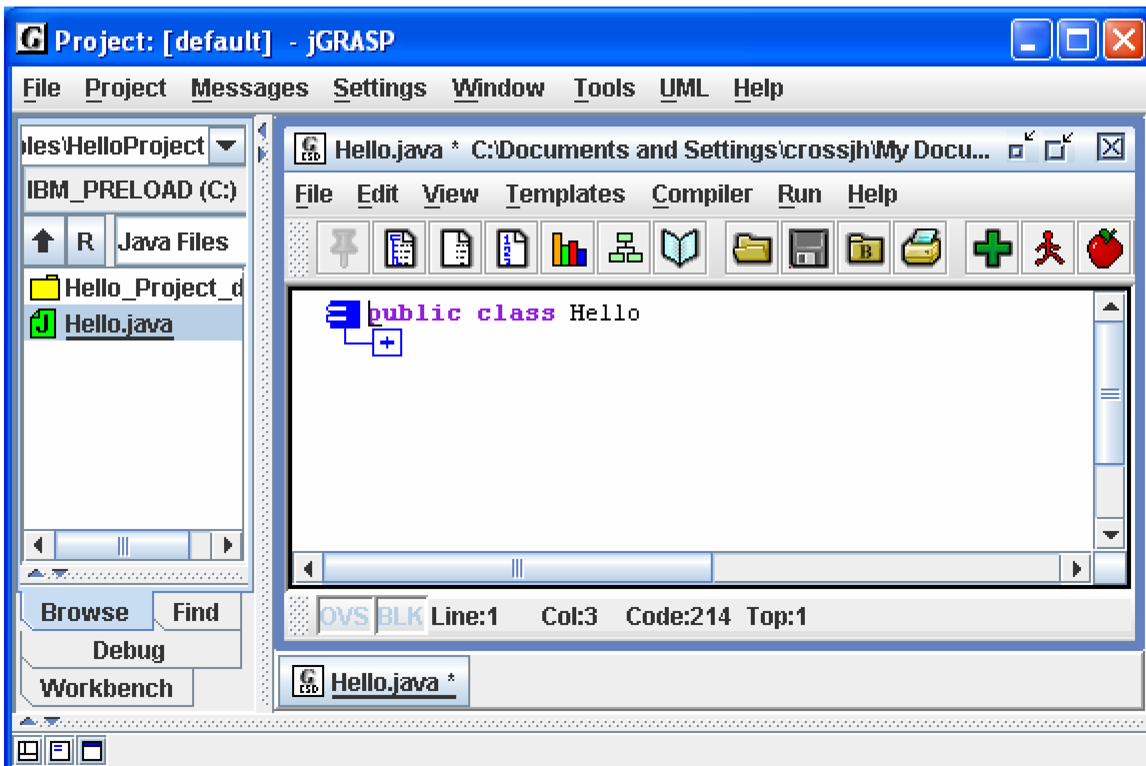
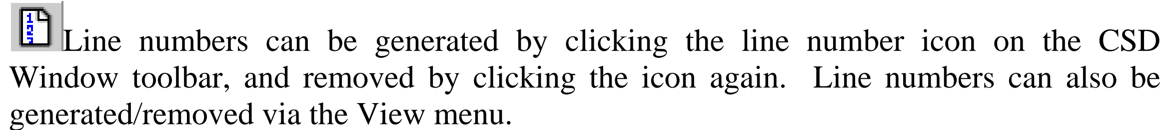


Figure 6. Folded CSD

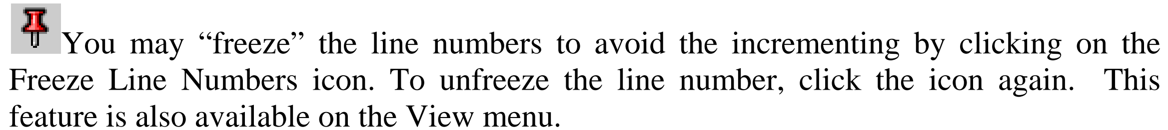
Although the example program has no loops or conditional statements, these may be folded by double-clicking the corresponding CSD control constructs. For other folding options, see the **View – Fold** menu.

2.5 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 he left of the source code as indicated in Figure 7.


Line numbers can be generated by clicking the line number icon on the CSD Window toolbar, and removed by clicking the icon again. Line numbers can also be generated/removed via the View menu.

With Line numbers turned on, new line numbers are inserted and/or added to the end each time you press "ENTER" on the keyboard. 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 icon. To unfreeze the line number, click the icon again. This feature is also available on the View menu.

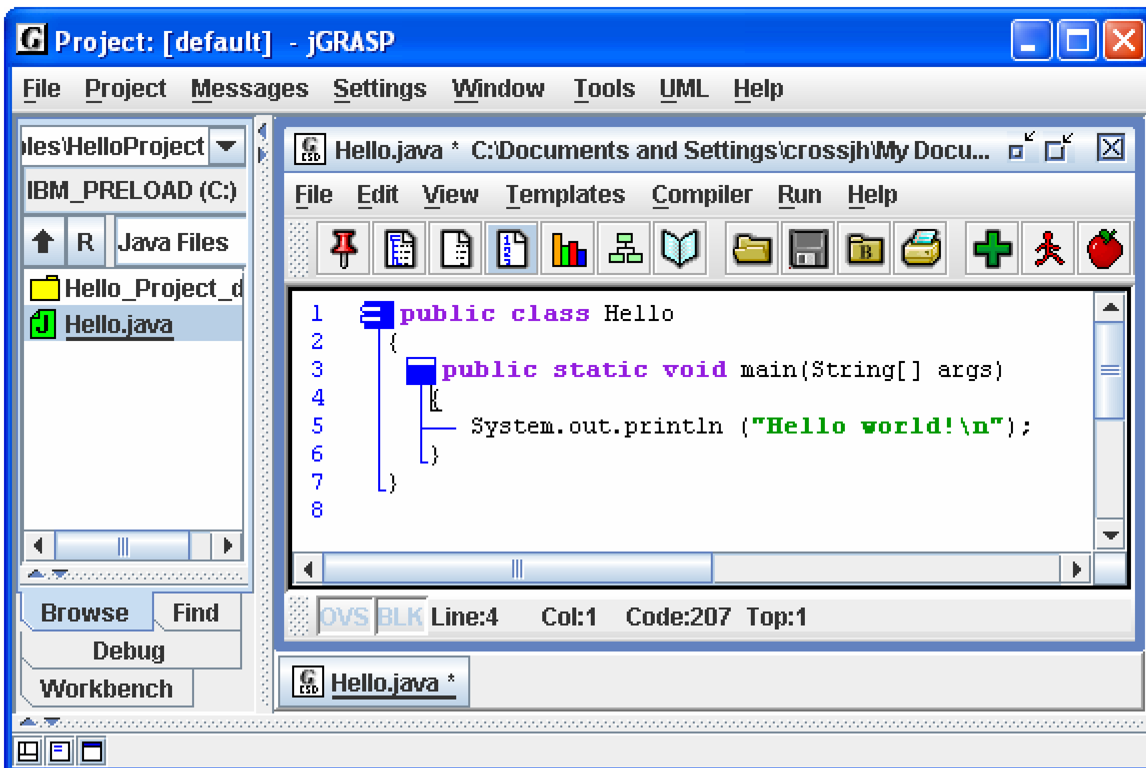


Figure 7. Line numbers in the CSD Window

2.6 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 all available languages. The languages for the last 25 files opened will be displayed in the 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 8 below. Notice the title for the frame, jGRASP CSD (Java), 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. Also, a *button* with the file name on it appears below the CSD window 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.

In the upper right corner of the CSD Window are three buttons that control its display:

The first button iconifies the CSD Window. The second either maximizes the CSD

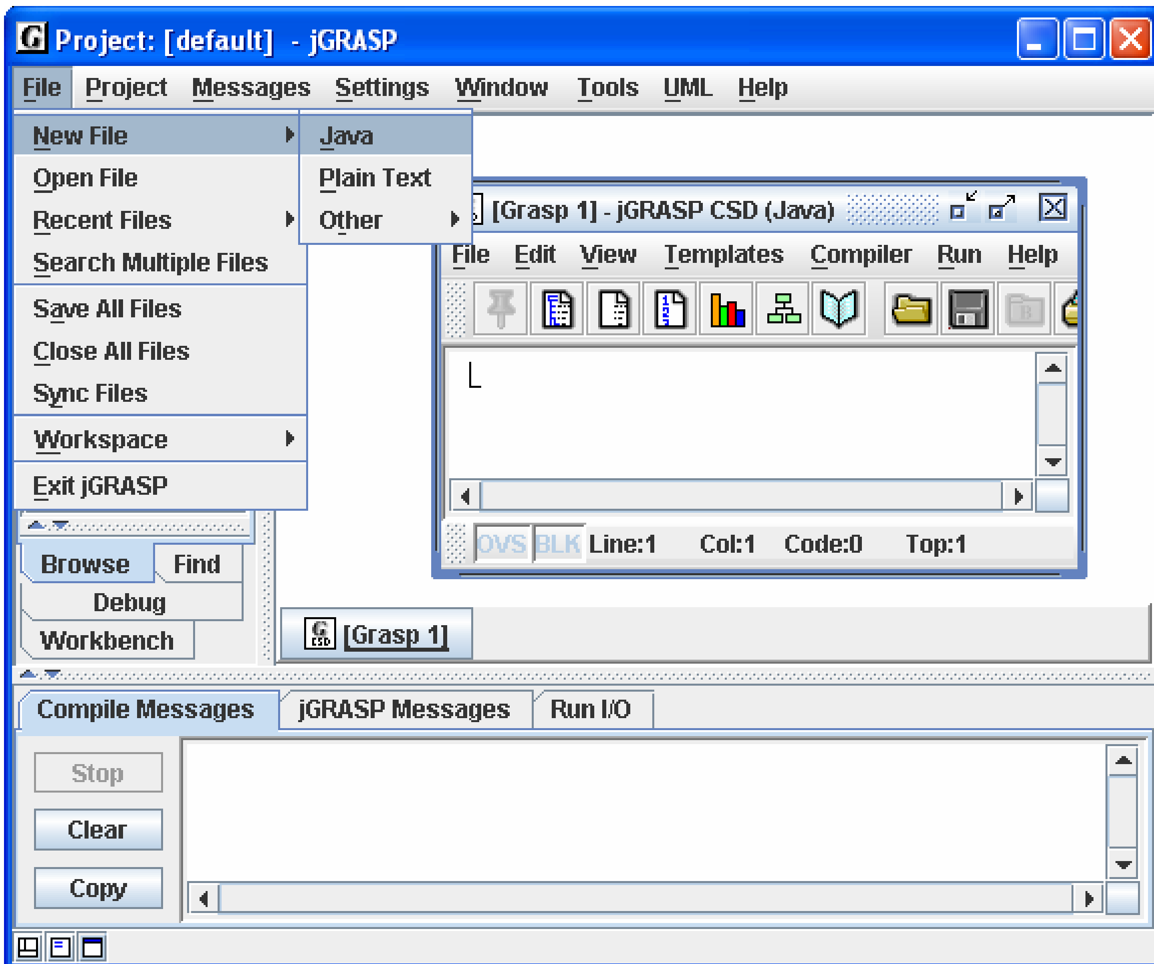


Figure 8. Opening a CSD Window for Java

Window relative to the jGRASP Desktop, 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 full screen by clicking the appropriate icon in the upper corner of it.

Figure 11 shows the CSD Window maximized within the virtual Desktop.

HINT: If you want all of your CSD Windows to be maximized automatically when you open them, click **Settings -- Desktop**, then turn on the option called **Open CSD Windows Maximized** (indicated by a check mark).

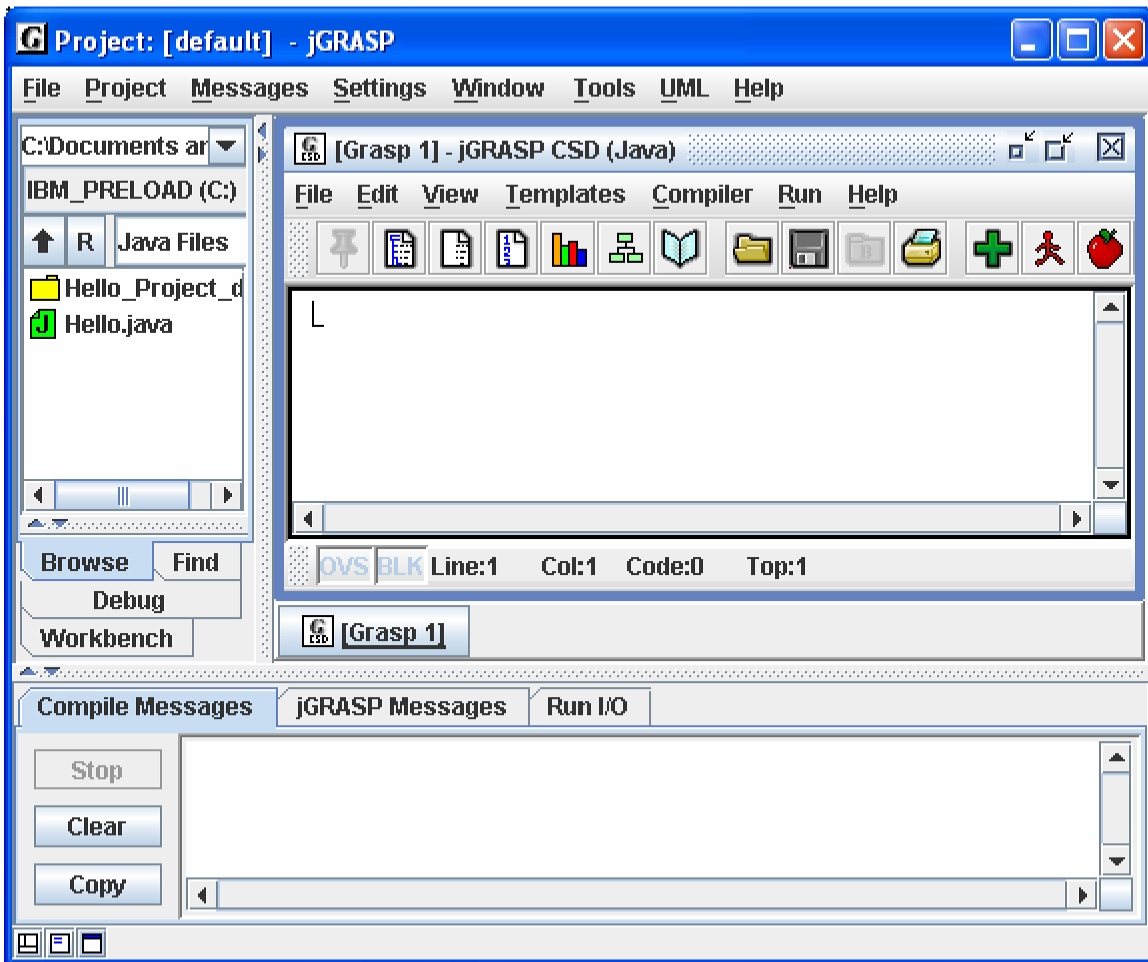


Figure 11. CSD Window expanded in Desktop

Type in the following Java program in the CSD Window, exactly as it appears. Remember, Java is case sensitive.

```
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 12.

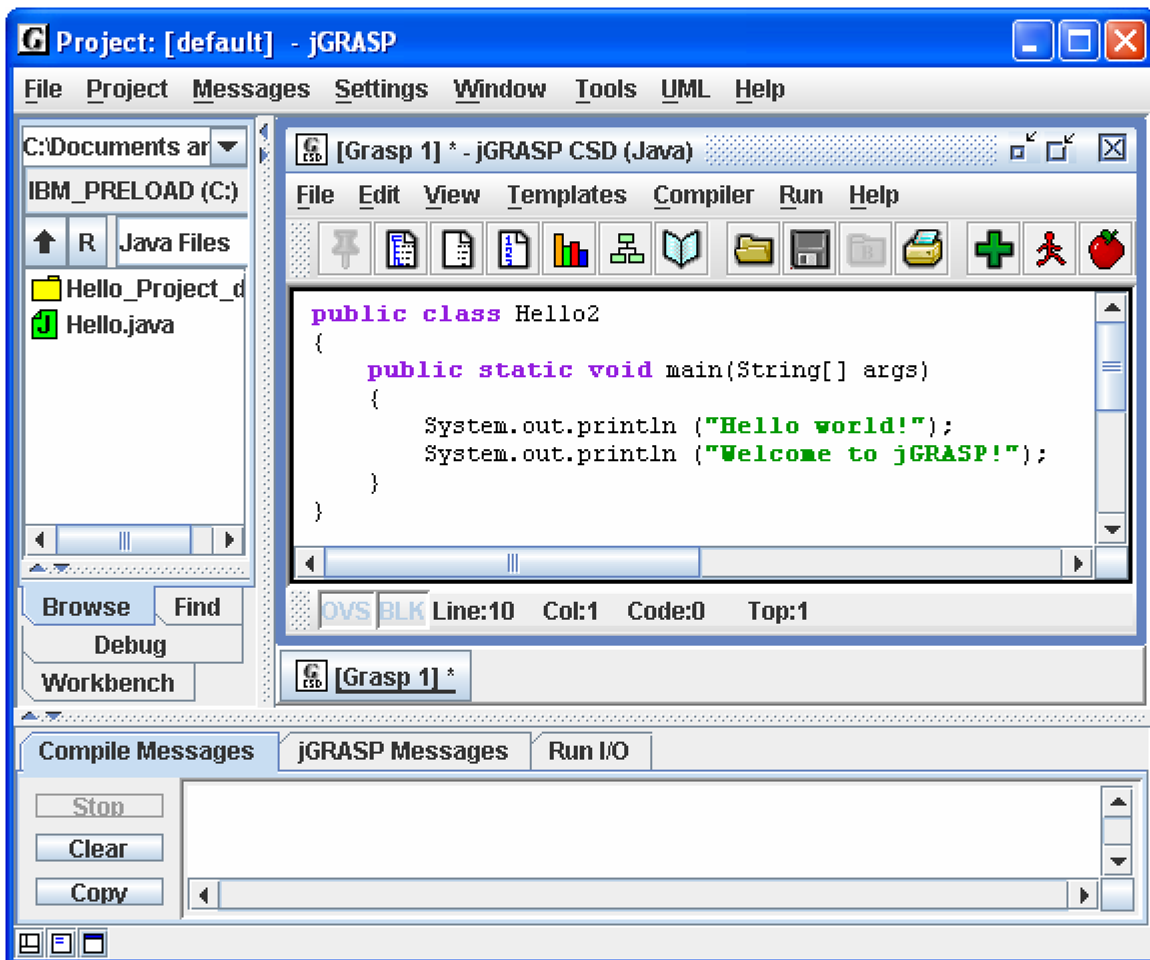


Figure 12. CSD Window with program entered

2.7 Saving a File

Save the program as "Hello2.java" by clicking the Save icon on the tool bar of the CSD Window, or you can click **File** -- **Save** on the CSD Window menu (not the Desk Top menu).

After you click on **Save**, the Save dialog box pops with the name of the file already 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 icon on the top row of the Save As 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 will be listed in the browse pane. If the program is not listed in browse pane, be sure the browse pane is set to the directory where the file was saved.

HINT: You can also use the Save icon on the toolbar.

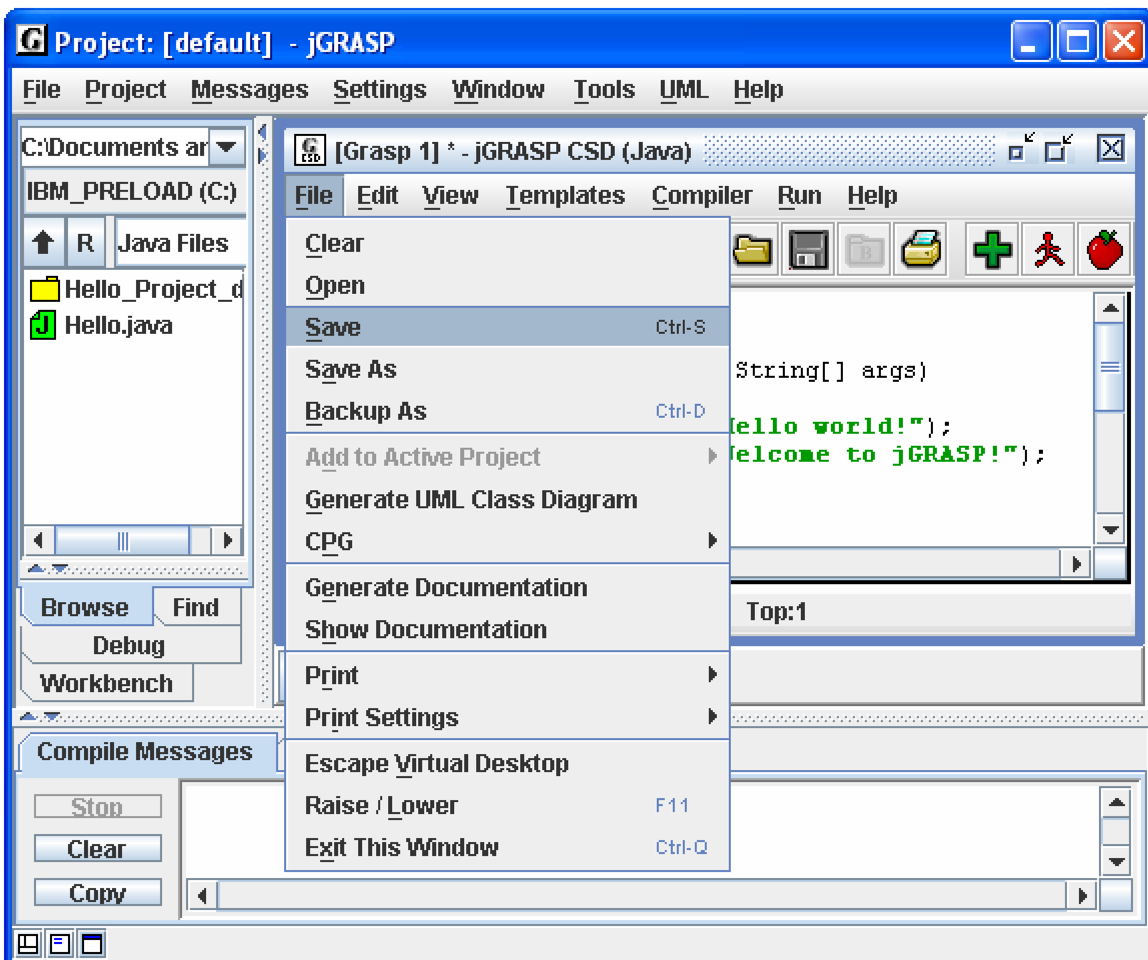
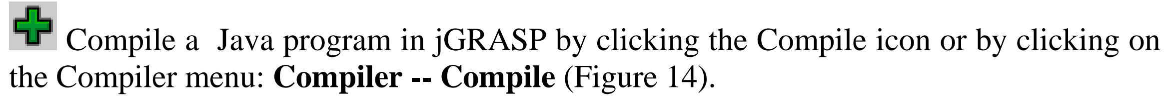


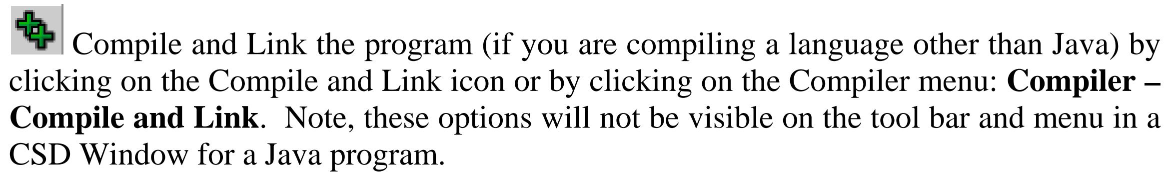
Figure 13. Saving a file from the CSD Window

2.8 Compiling a Program – A Few More Details

When you have a program in the CSD Window, either by loading a file or typing it in and saving it, you are ready to compile the program. 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 icon or by clicking on the Compiler menu: **Compiler -- Compile** (Figure 14).



Compile and Link the program (if you are compiling a language other than Java) by clicking on the Compile and Link icon or by clicking on the Compiler menu: **Compiler – Compile and Link**. Note, these options will not be visible on the tool bar and menu in a CSD Window for a Java program.

In the figure below, also note that **Debug Mode** is checked ON. This should be 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.

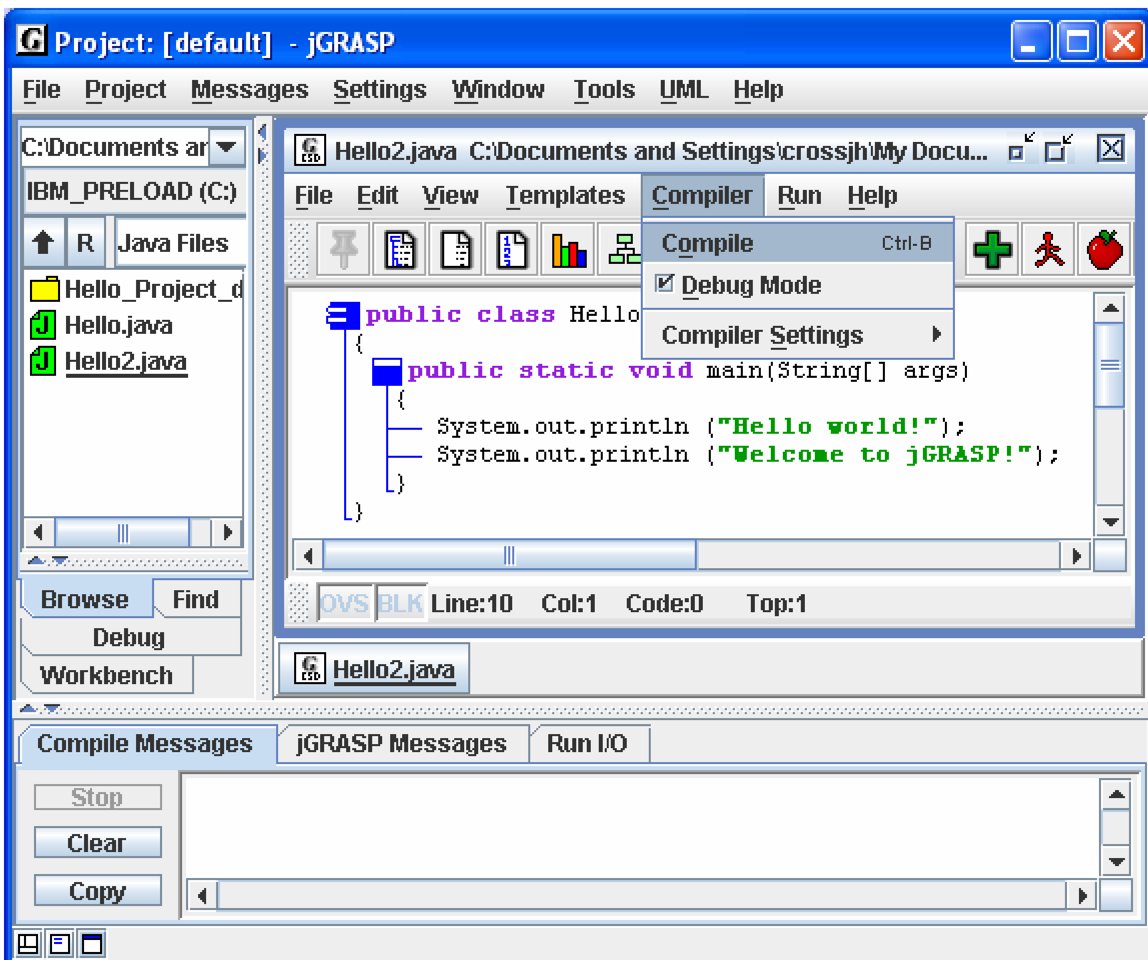


Figure 14. Compiling a program

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 15, and you are now ready to "Run" the program (see next section).

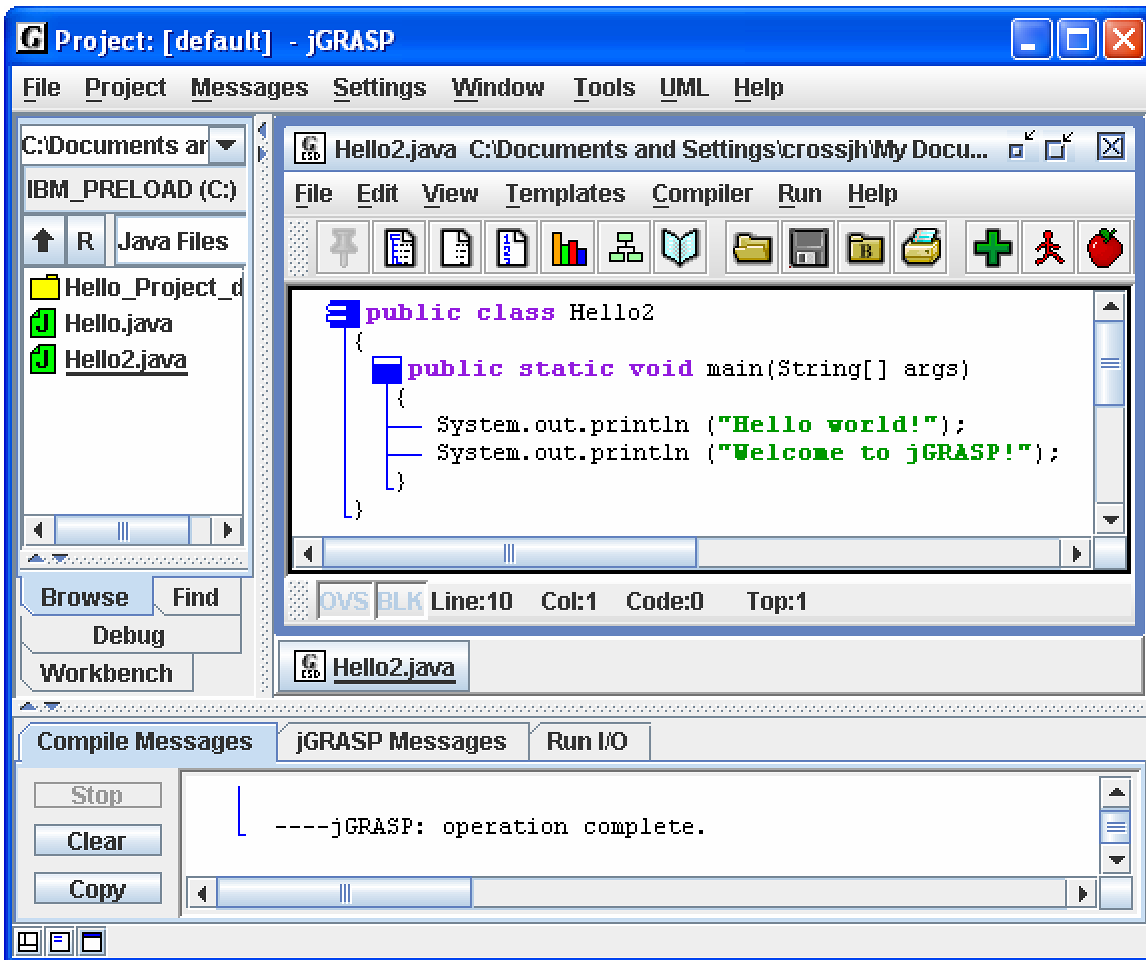


Figure 15. A successful compilation

Error Messages

If you receive an error message indicating "file not found," this 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 usually means the Java compiler (javac) was not installed properly. Go back to Section 1, Installing jGRASP, and be sure you have followed all the instructions. Once the Java SDK compiler is properly installed and set up, any errors reported should be about your program.

If your program does not compile, the errors reported by the compiler will be displayed in the Compile Messages window (Figure 16). The description of first error detected will be highlighted, and jGRASP automatically scrolls the CSD Window to the line where the error most likely occurred and highlights it.

Even if multiple errors are indicated, as soon you correct the first error reported, you should attempt to compile the program again. Sometimes a single error causes a cascade of reported errors.

Only after you have "fixed" all these reported errors will your program actually compile, and the program must compile before you can "Run" the program as described in the next section.

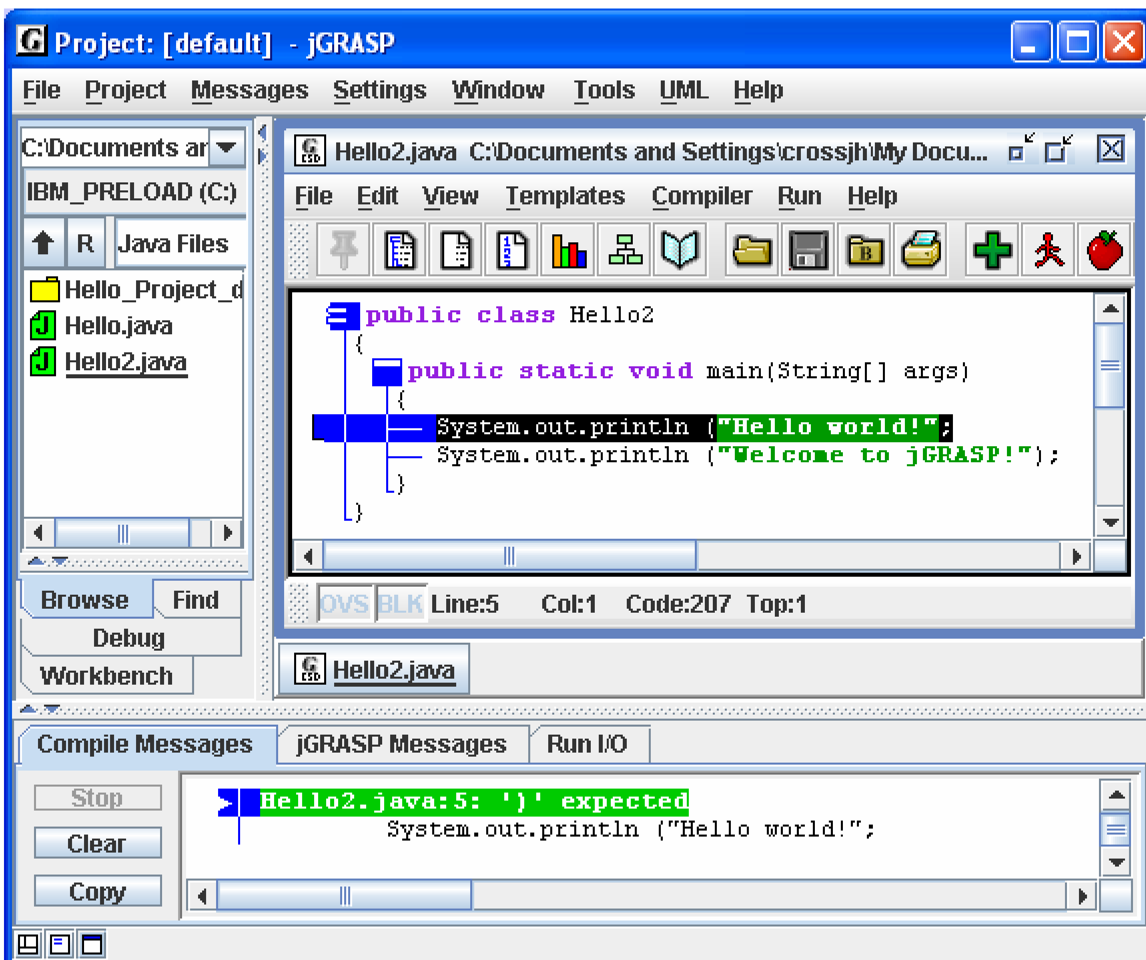


Figure 16. Compile time error reported

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 **Run – Run** on the CSD Window tool bar (Figure 17). The options on the Run menu allow you to run your program as an application (**Run**), as an Applet (**Run as Applet**), as an application debug mode (**Debug**), as an Applet in debug mode (**Debug as Applet**). Other options allow you to pass Run arguments and Run in an MS-DOS window rather than the jGRASP Run I/O message pane.

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

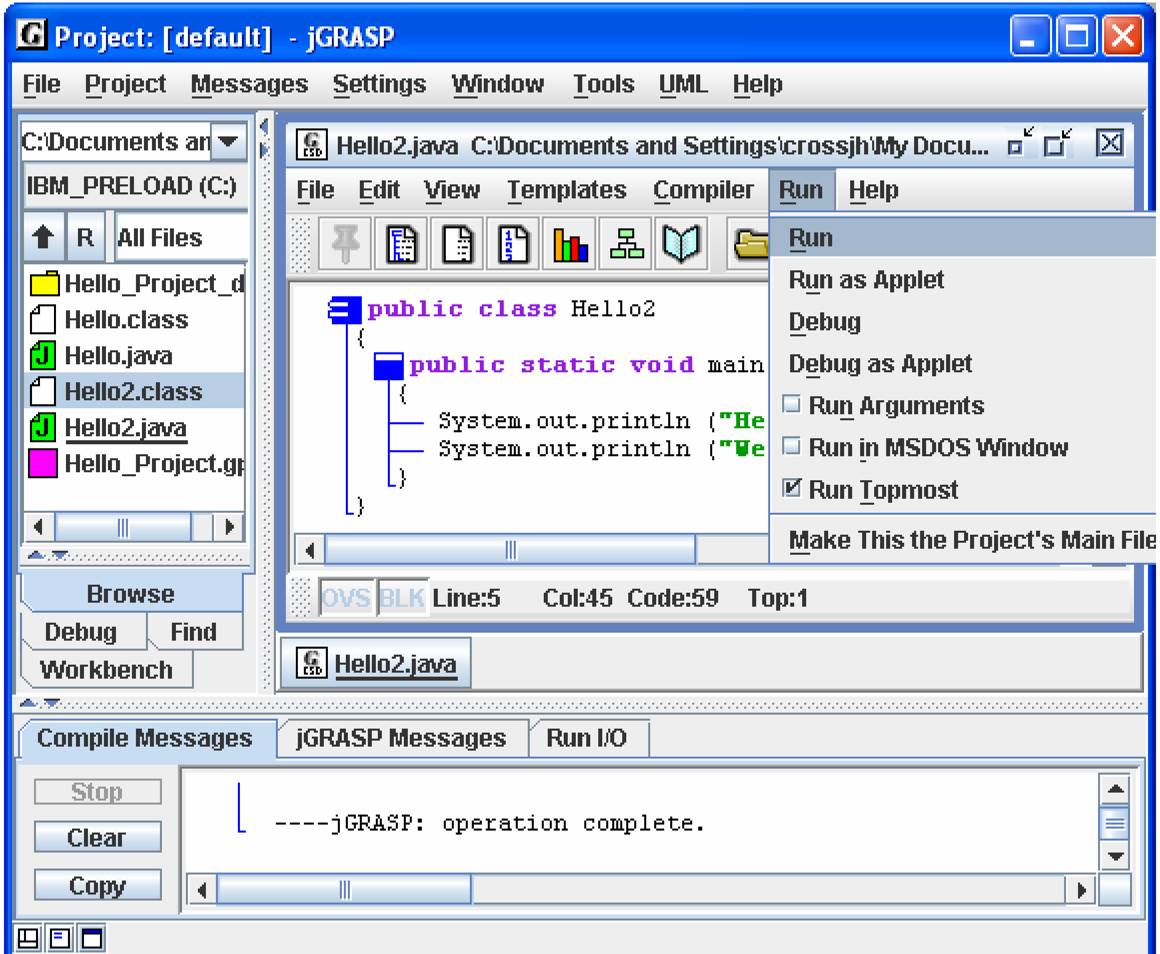


Figure 17. Running a program

Output

When you run your program, the Run I/O tab in the lower pane pops to the top of the Desktop. The results of running the program are displayed in this pane as illustrated in Figure 18.

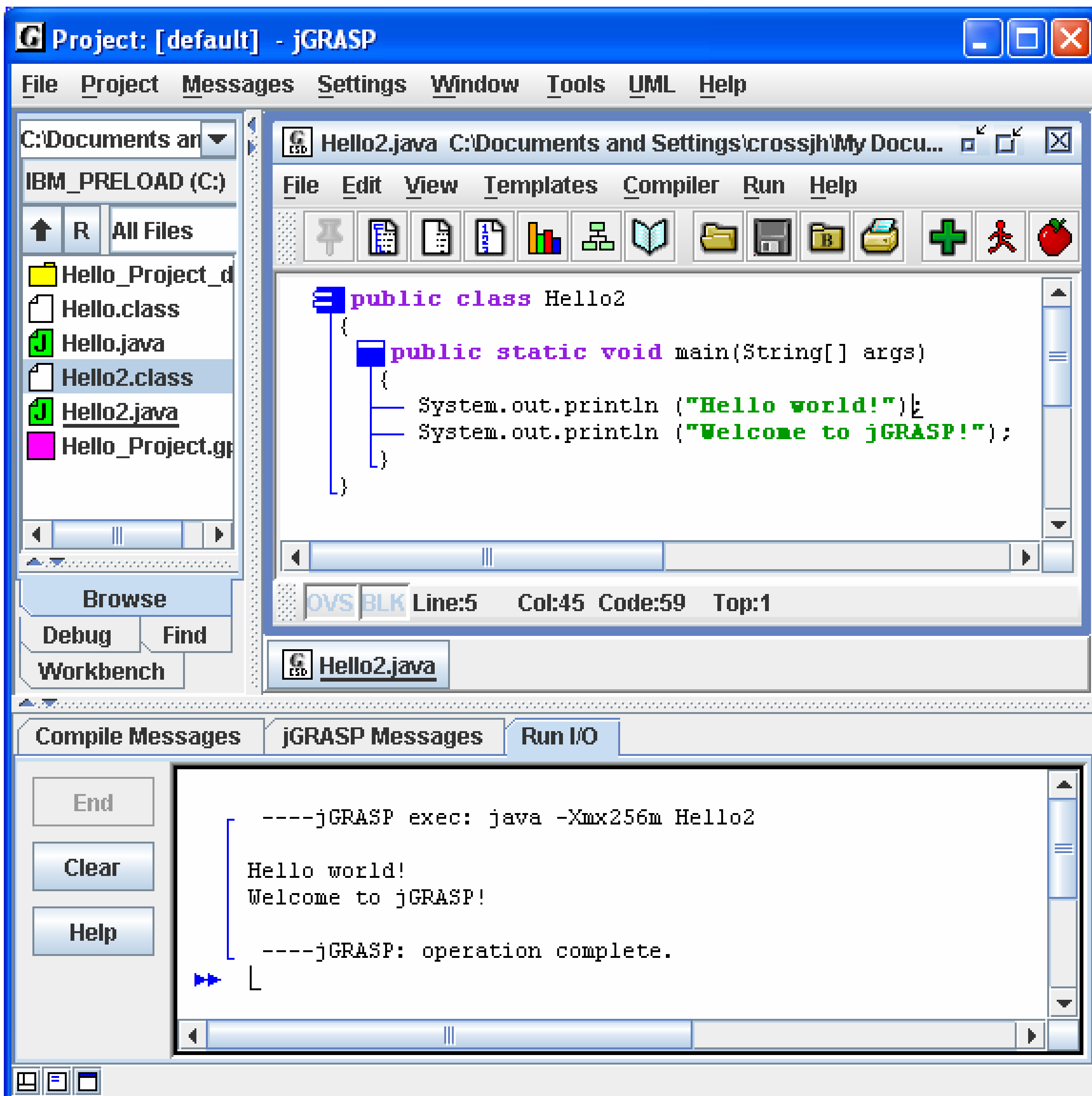


Figure 18. Output from running the program

2.10 Using the Debugger

jGRASP provides an easy-to-use visual Debugger that allows you to set one more breakpoints, then step through a program statement by statement. To set a breakpoint, left-click on the statement where you want your program to stop, then right-click and select **Toggle Breakpoint** (Figure 19). You should see the red octagonal breakpoint symbol appear 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). 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 breakpoint symbol, left-click the mouse to set the breakpoint.

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 that allow a breakpoint in this program.

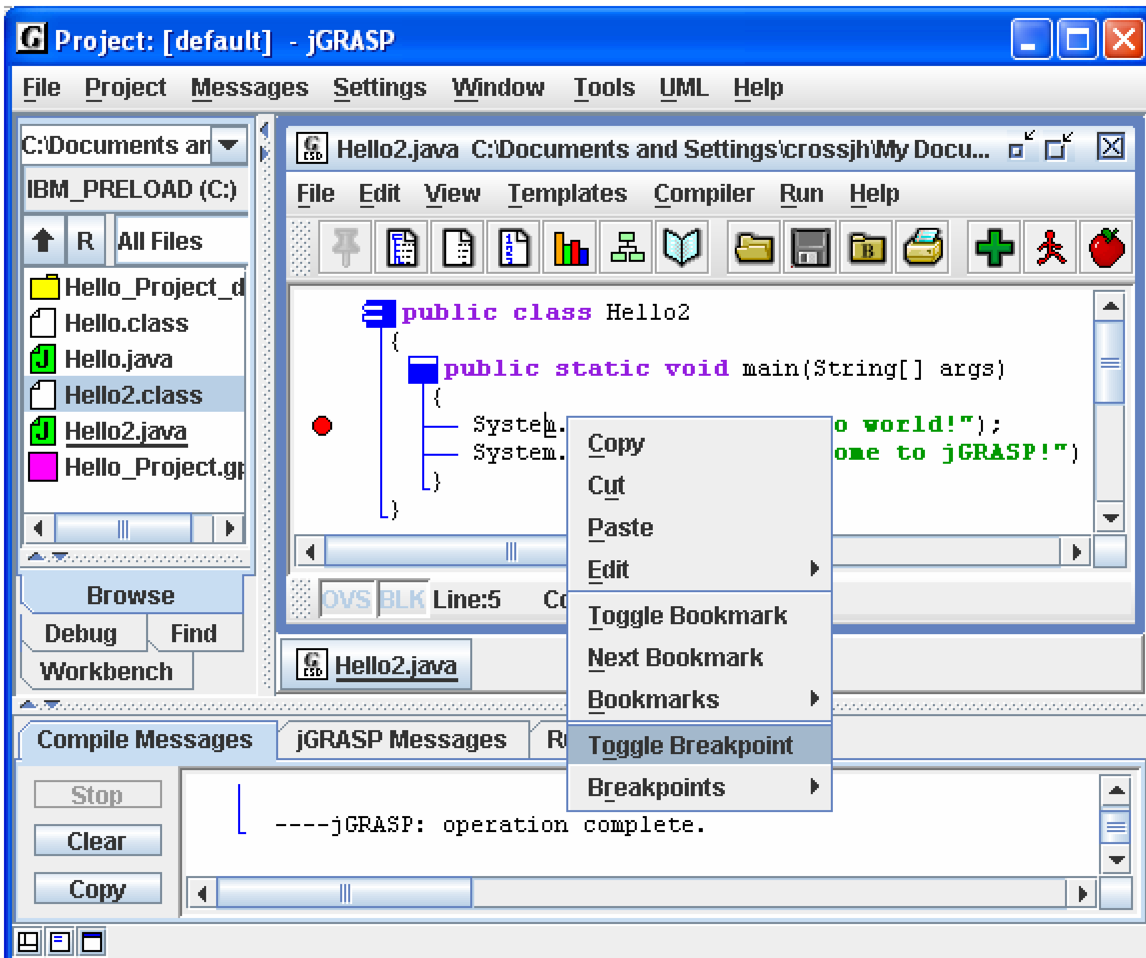


Figure 19. Setting a breakpoint

After setting the breakpoint, click **Run – Debug** (Figure 20). This should raise the Debug tab pane (in place of the Browse tab pane), and your program should stop at the breakpoint. The highlighted statement is the one about to be executed. To *step* the program, click on the "down-arrow" at the top of the Debug pane. Each time you click on the "down-arrow", 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.

In the example below, the program has stopped at the first output statement. When the *step button* (down-arrow) is clicked, this statement will be executed and "Hello world!" will be printed standard out and shown in the Run I/O tab pane. Clicking the step button again will output "Wecome 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.

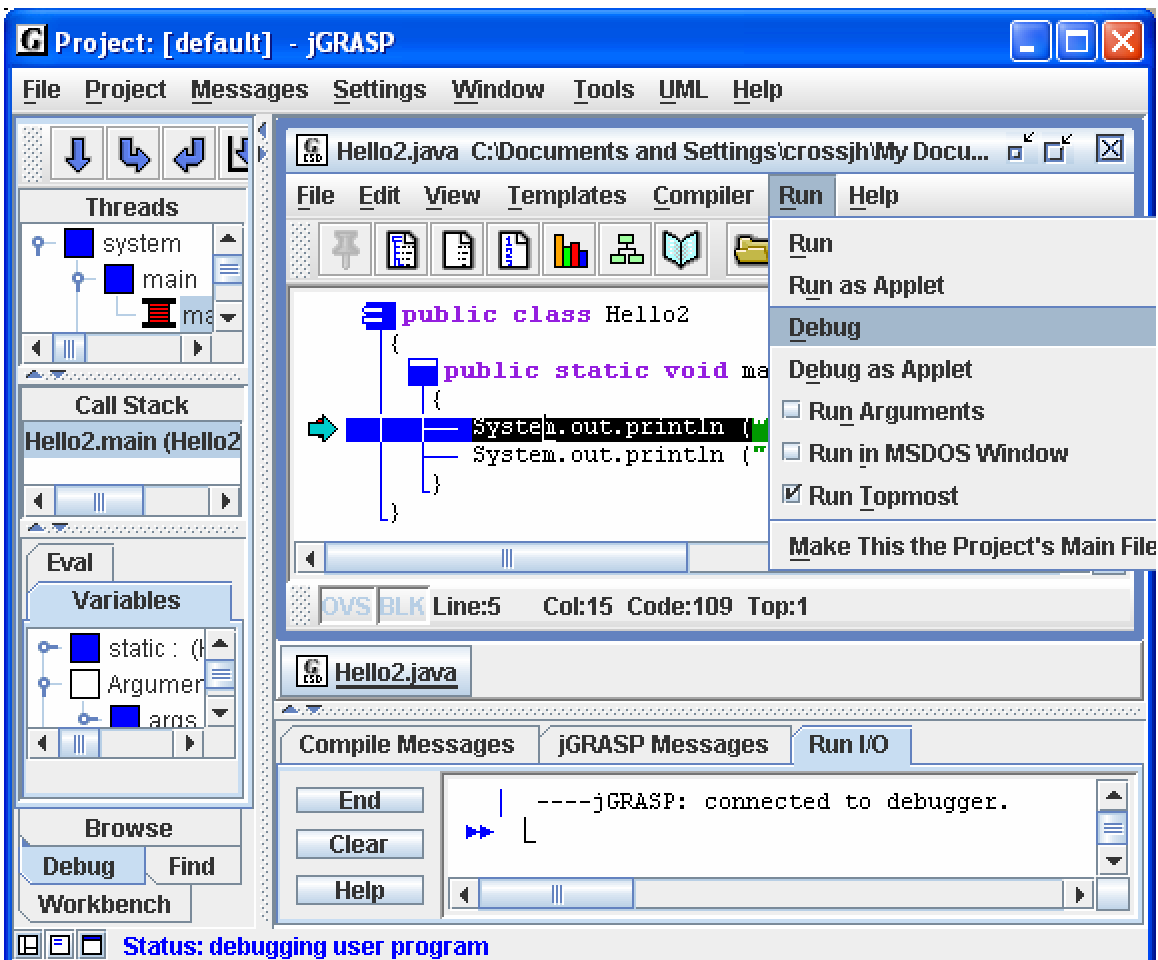


Figure 20. Starting the Debugger

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 pane, 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**.

(2) **Desktop Menu** - On the Desktop menu, click **File – Open** as illustrated in Figure 21. This will bring up the Open File dialog.

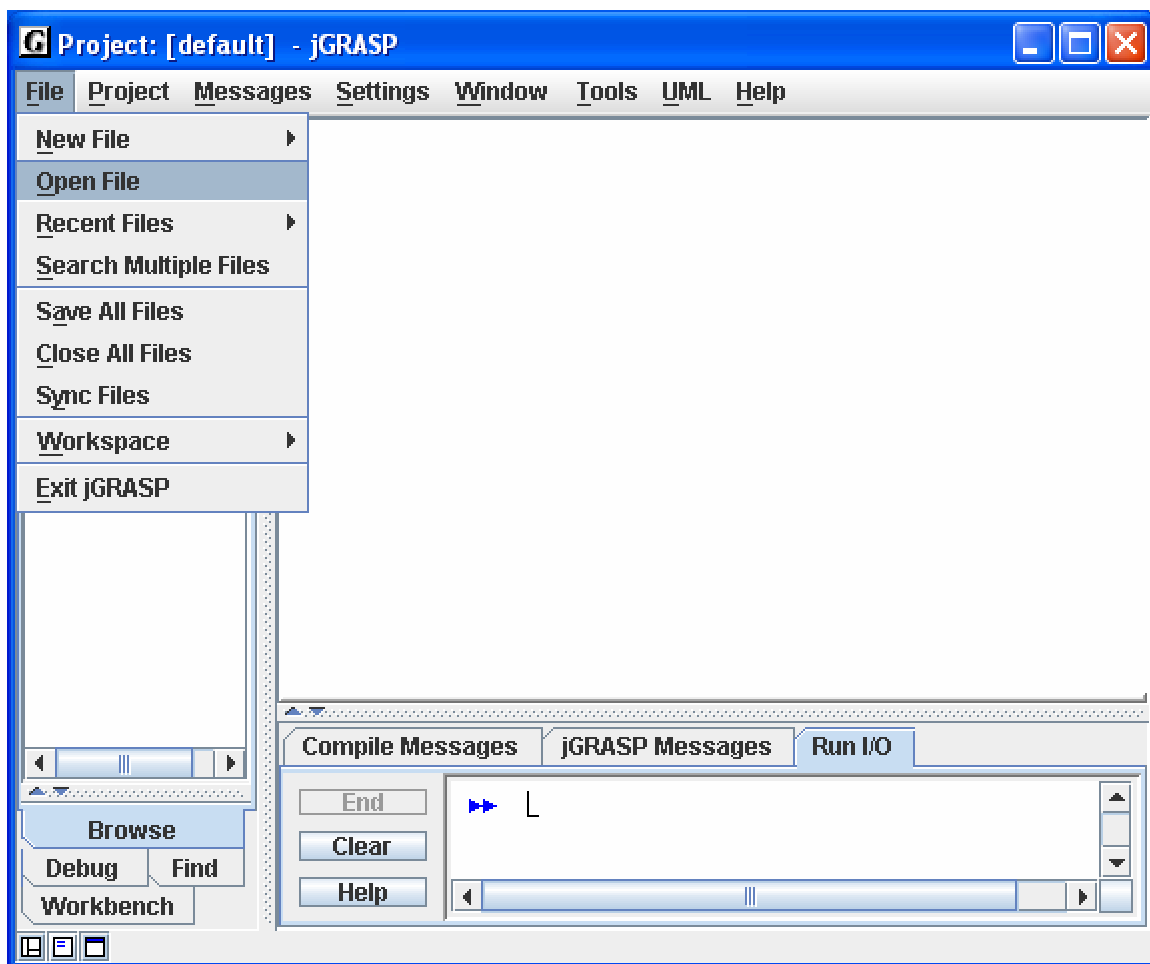


Figure 21. Opening a file from the Desktop

(3) **CSD Window Menu** - If you have a CSD Window open, click **File – Open** as shown in Figure 22. This will open the Open File dialog box, which will allow you browse up and down directories until you locate the file you want to open.

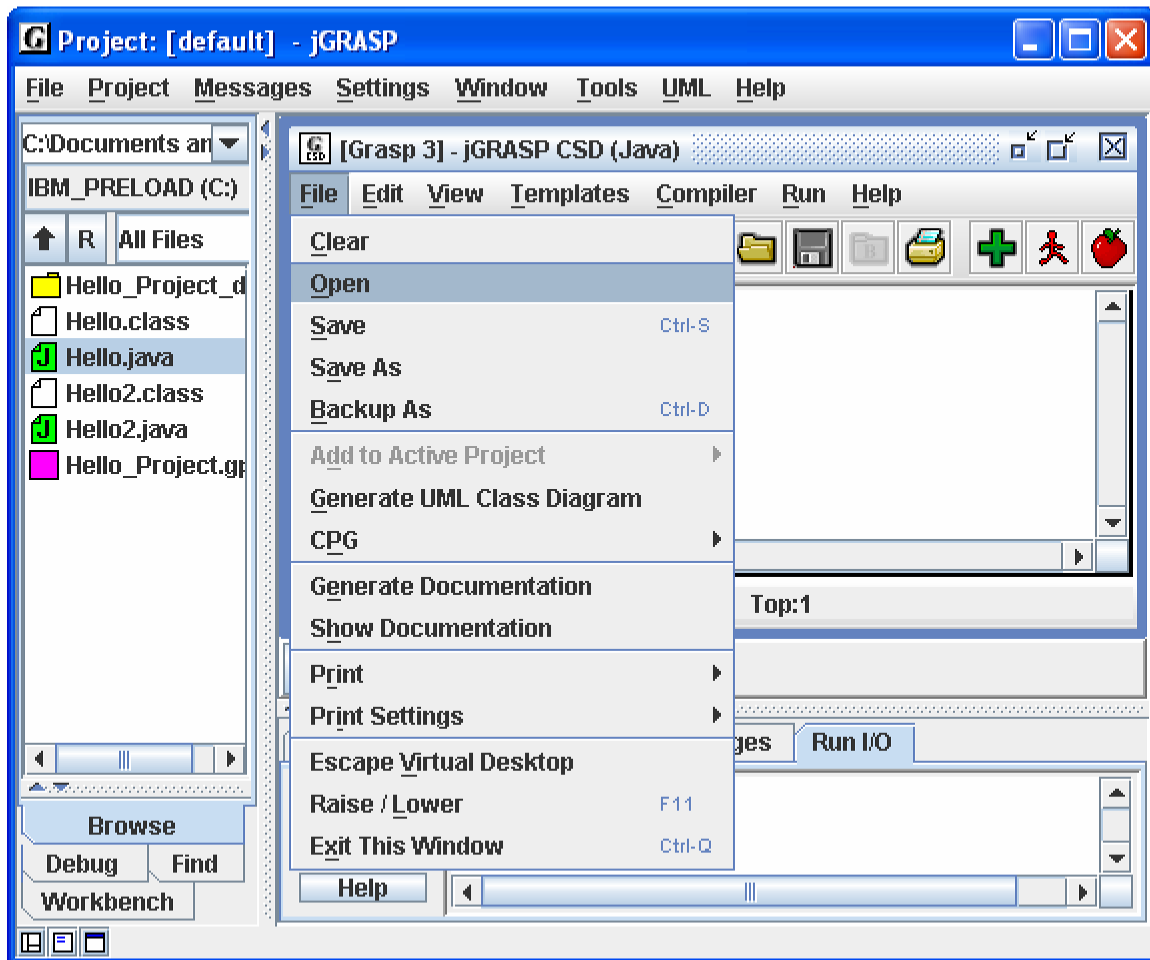


Figure 22. Opening a file from the CSD Window

(4) **Windows File Browser** - If you have a Windows file browser open (e.g., Windows Explorer, My Computer, or My Documents), and the file is marked as a jGRASP file, you can just double click the file name.

(5) **Windows File Browser (drag and drop)** - If you have a Windows file browser open (e.g., Windows Explorer or My Computer), you can drag-and-drop a file to the jGRASP Desktop canvas where the CSD Window will be displayed. However, files usually open more quickly by double-clicking (option 4 above) rather than using the drag-and-drop option.

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

You can have multiple CSD Windows open, each with a separate file. Each program can be compiled and run from its respective CSD Window. In Figure 18, two CSD Windows have been opened. One contains Hello.java and the other contains Hello2.java. If the window you want to work in is visible, simply click the mouse on it to bring it to the top. Otherwise, click the **Window** menu on the upper tool bar, and a drop down menu will list all of the open files. However, the easiest way to keep track of your open CSD windows is by clicking the window's button on the *windowbar* below the CSD Window. In Figure 23, the windowbar has buttons for Hello and Hello2. Notice that the Hello2 button is underlined to indicate that it is currently the top window. Hello2 is also underlined in the Browse tab.

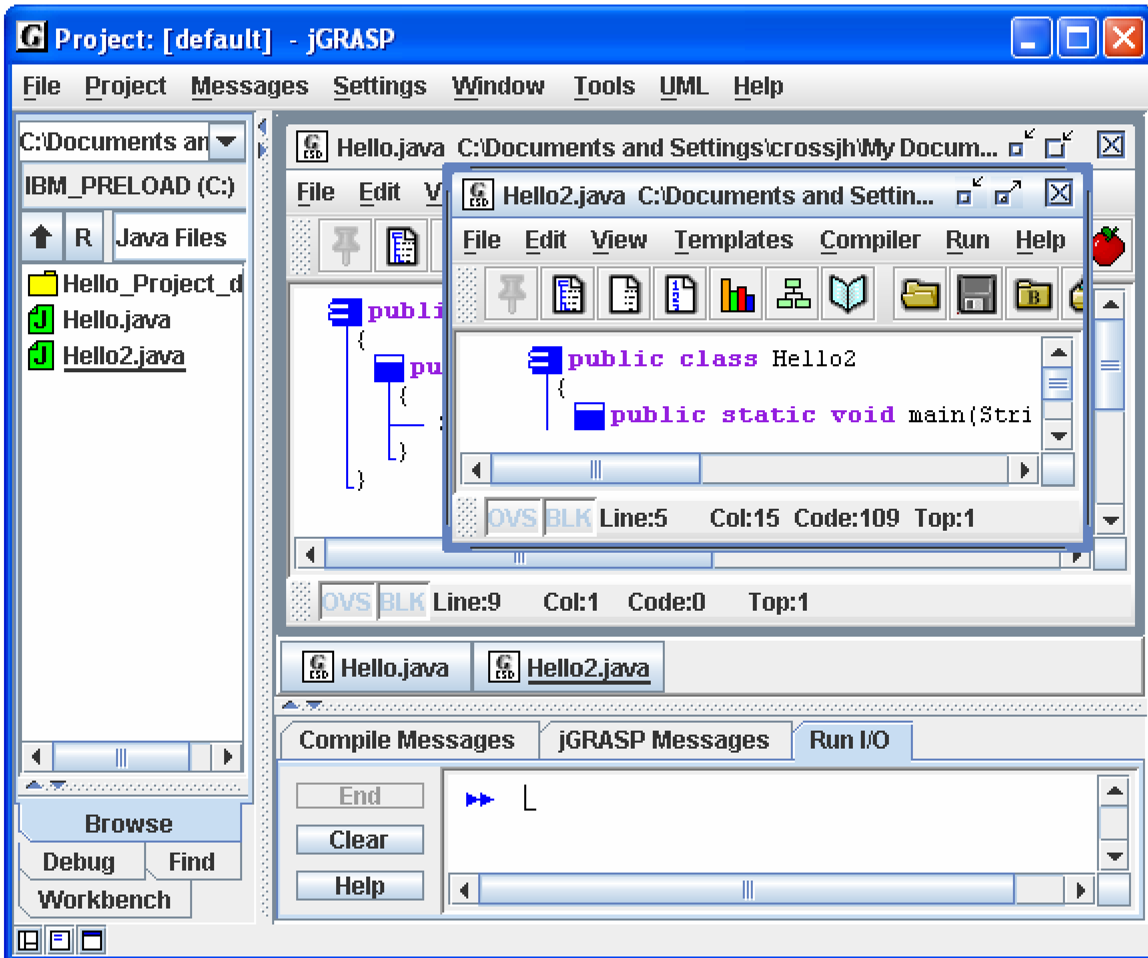
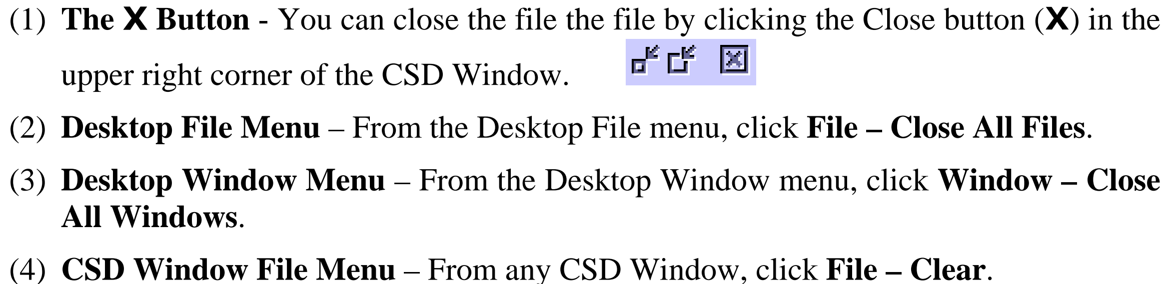


Figure 23. Multiple files open

2.12 Closing a File

The open files in CSD Windows can be closed in several ways. In each of the scenarios below, 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.

(1) **The X Button** - You can close the file the file by clicking the Close button (**X**) in the upper right corner of the CSD Window. 

(2) **Desktop File Menu** – From the Desktop File menu, click **File – Close All Files**.

(3) **Desktop Window Menu** – From the Desktop Window menu, click **Window – Close All Windows**.

(4) **CSD Window File Menu** – From any CSD Window, click **File – Clear**.

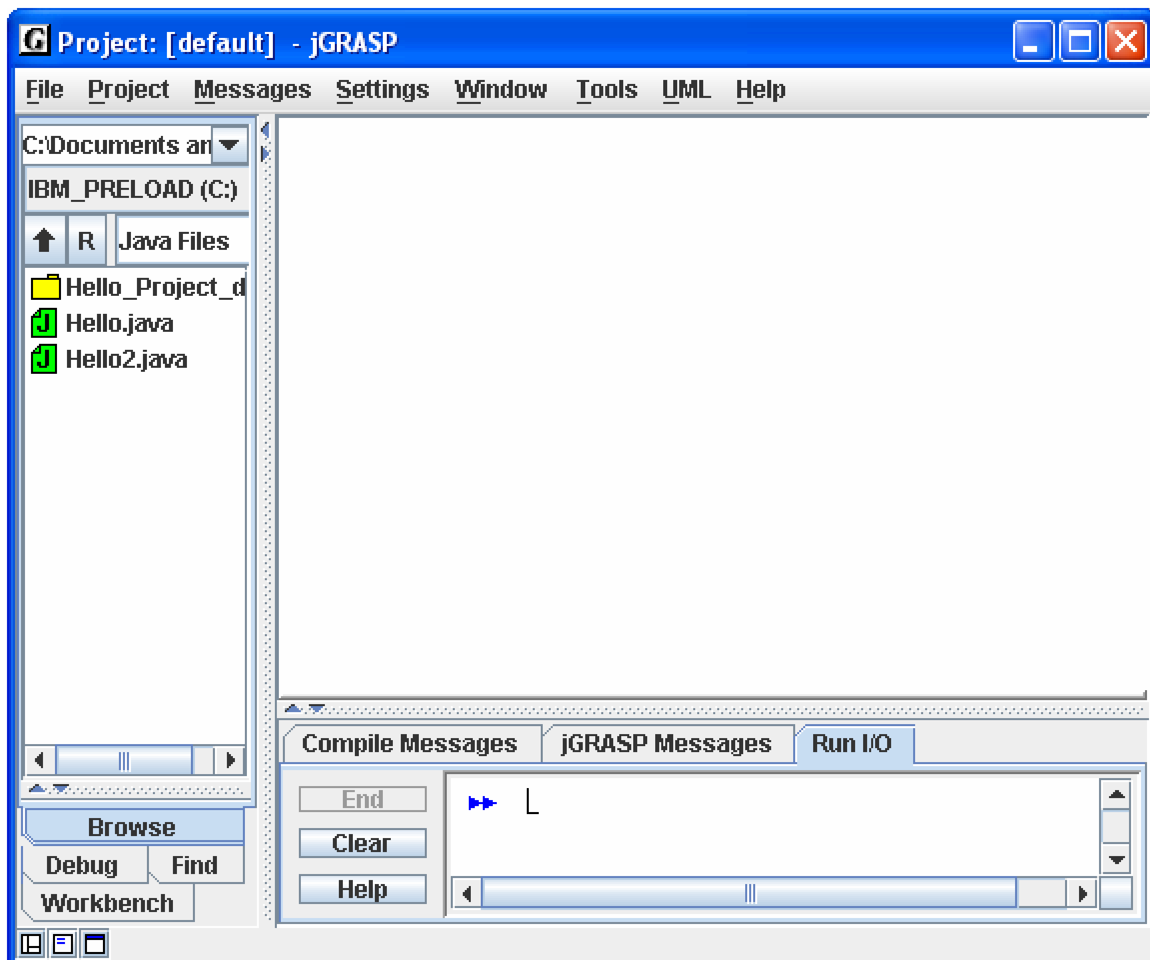


Figure 24. Desktop with all CSD Windows closed

2.13 Exiting jGRASP

When you have completed you 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 normally, 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. 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 open in a CSD Window.

(1) **The X Button** - You can exit jGRASP by clicking the Close button (**X**) in the upper right corner of the Desktop.

(2) **Desktop File Menu** – From the Desktop 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.

(3) Display the line numbers for your program.

(4) Fold up your program then unfold it in layers.

(5) On the Compiler menu, set the Debug Mode ON (with check box), if it is not already ON. [We recommend that this be left ON.] Recompile your program.

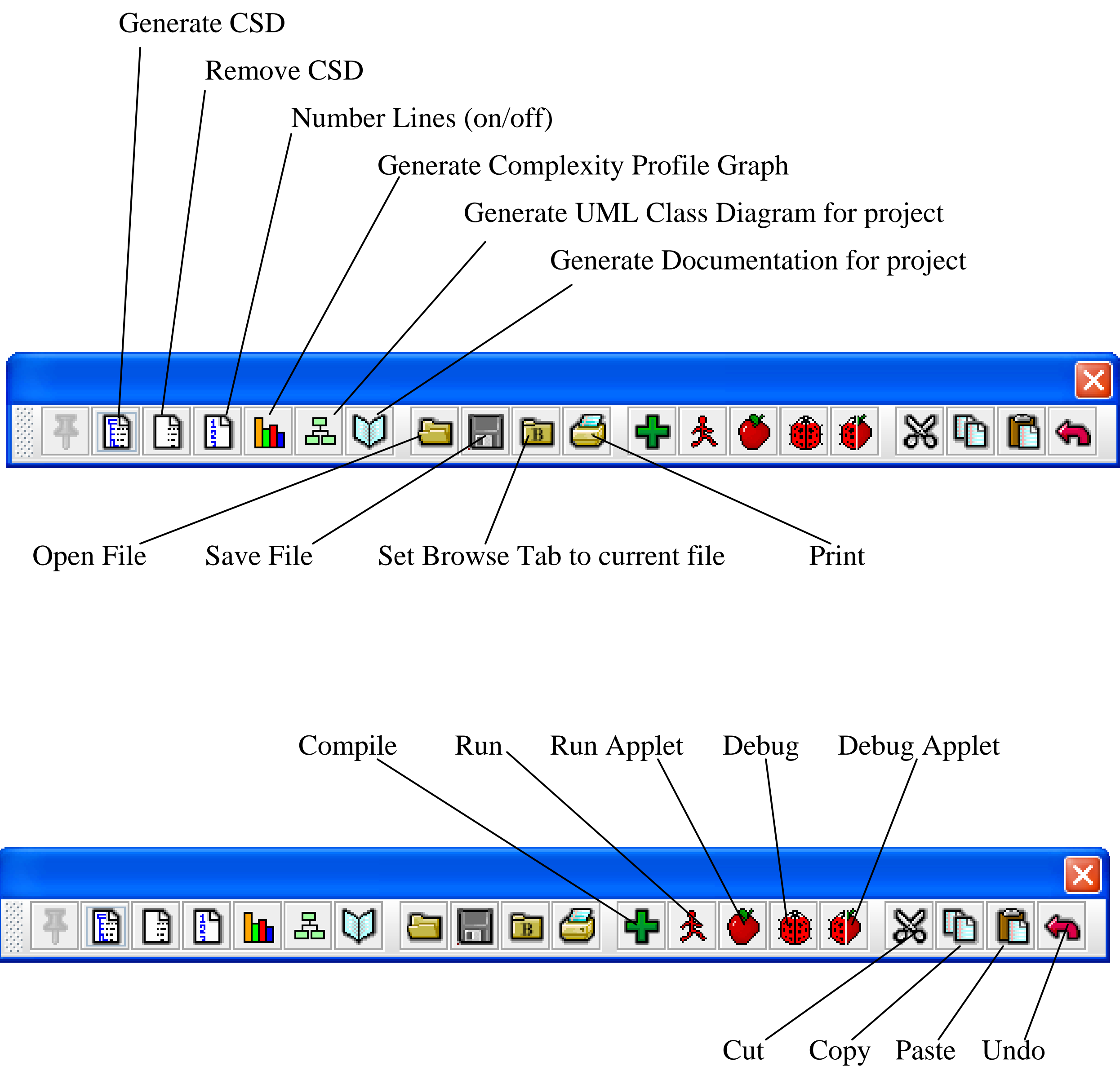
(6) Set a breakpoint on the first executable line of your program then run it with the debugger. Step through each statement, checking the Run I/O window for output. [**Note**: Make sure you have compiled your program with Debug Mode checked ON; see the Compiler menu.]

(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 CSD Window *toolbar*. Hovering the mouse over an icon on the toolbar will provide a "tool hint" to help remember is function. Also, **View – Toolbar Buttons** will allow you to display labels (text) for each icon.

While most of the icons are self explanatory, two deal with projects (Generate UML class diagrams and Generate Documentation). Projects, UML class diagrams, the Object Workbench, and the Debugger are covered in sections 4, 5, 6, and 7. Section 3 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 have had experience working with any IDE, this tutorial can be done without having done the previous section, *Getting Started*. However, at some point you should make sure you can do the exercises at the end of the previous section.

Objectives – When you have completed this tutorial, you should be able to use projects, UML class diagrams, and the Object Workbench 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

3.3 Opening a Project and UML Window

3.4 The UML Window

3.5 Exploring the Features of the UML Window

3.5.1 Viewing the source code for a class

3.5.2 Displaying class information

3.5.3 Displaying Dependency Information

3.6 Viewing the Source Code

3.7 Compiling and Running the Program

3.8 Generating Documentation for the Project

3.9 Using the Object Workbench

3.10 Invoking a Method

3.11 Invoking Methods with Parameters

3.12 Invoking Methods on Object Fields

3.13 Invoking Inherited Methods

3.14 Running the Debugger on Invoked Methods

3.15 Creating Instance from the Java Class Libraries

3.16 Exiting the Workbench

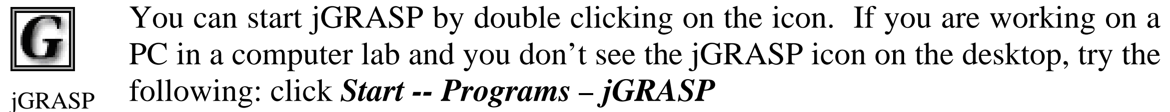
3.17 Closing a Project

3.18 Exiting jGRASP

3.19 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, using the methods provided by their respective classes, toward some useful purpose. In this tutorial, we'll examine a simple program called PersonalLibrary that consists of five class files. In jGRASP, these five class files are organized as a project.



You can start jGRASP by double clicking on the icon. 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** (Project tab is combined with the Browse tab in version 1.7). The large *right pane* is for UML and CSD Windows. The *lower pane* has tabs for jGRASP messages, Compile messages, and Run Input/Output.

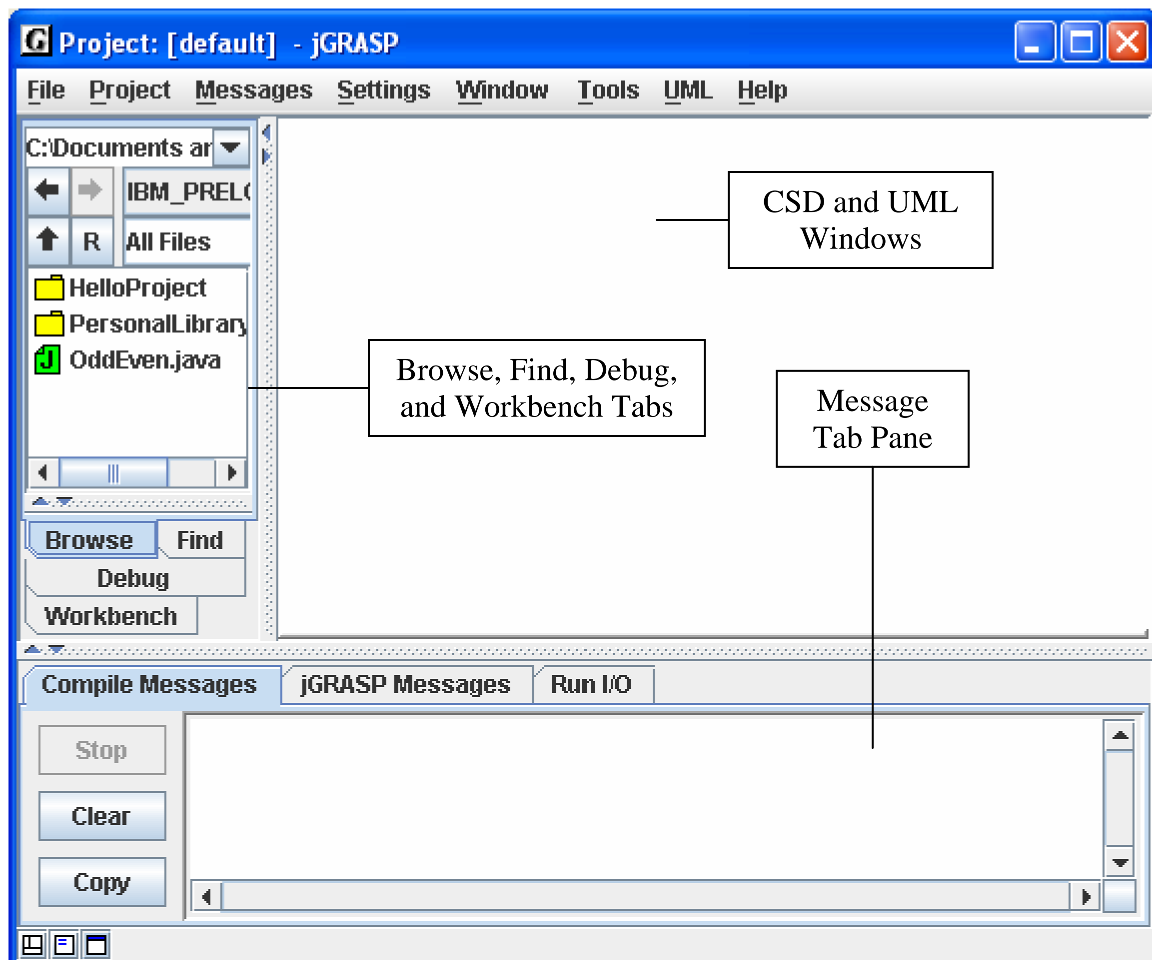


Figure 25. The jGRASP Virtual Desktop

3.2 Navigating to Our First Example

Example programs are available in the jGRASP folder in the directory where it was installed (e.g., c:\program files\jgrasp\examples\tutorial_examples). 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_examples folder to one of your personal 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 tutorial_examples.

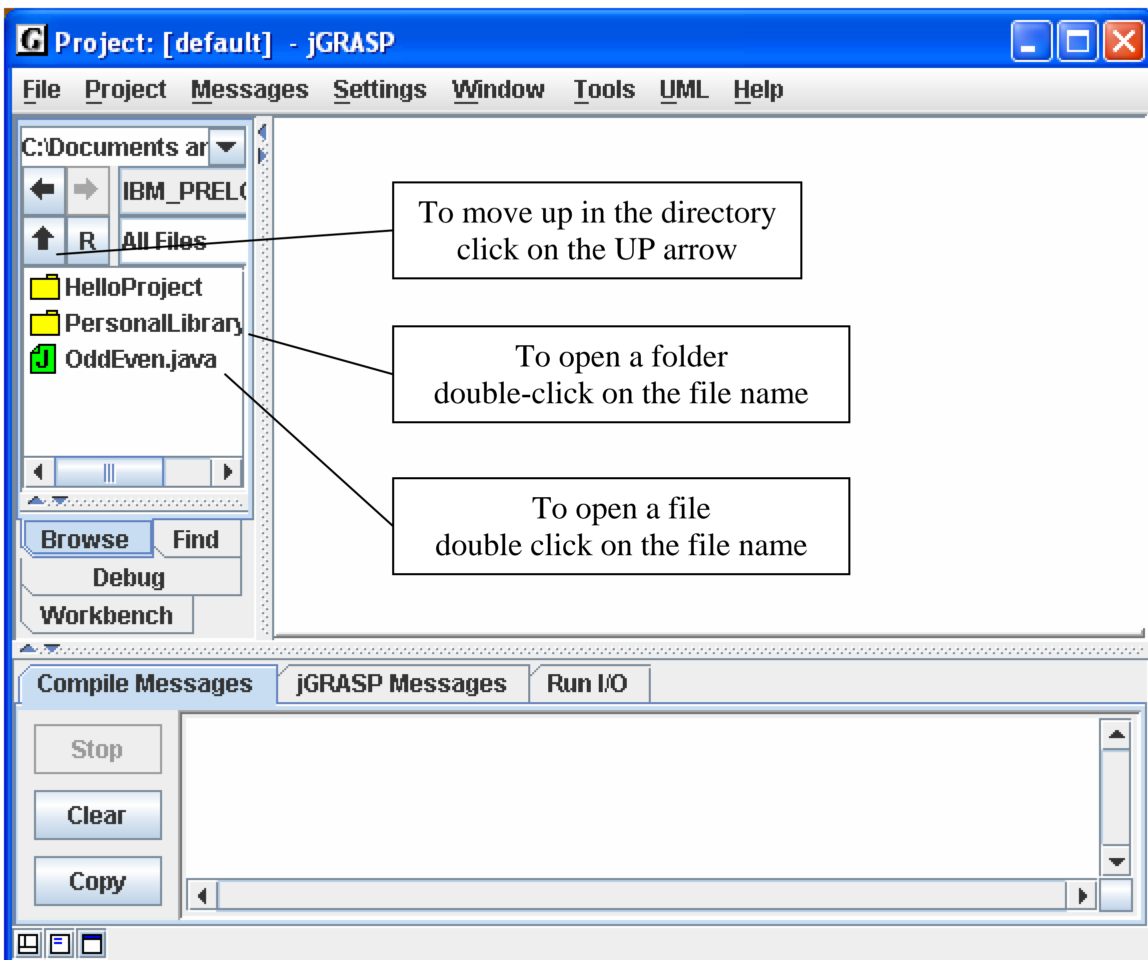


Figure 26. The jGRASP Virtual Desktop

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 and make it active, 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 for files and the lower 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 icon shown beneath the project name in the open projects section of the Browse tab. Alternatively, on the desktop menu you can click **Project – Active Project – Generate/Update UML**.

After you have opened the UML Window, you can compile and run your program in the traditional way. 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.

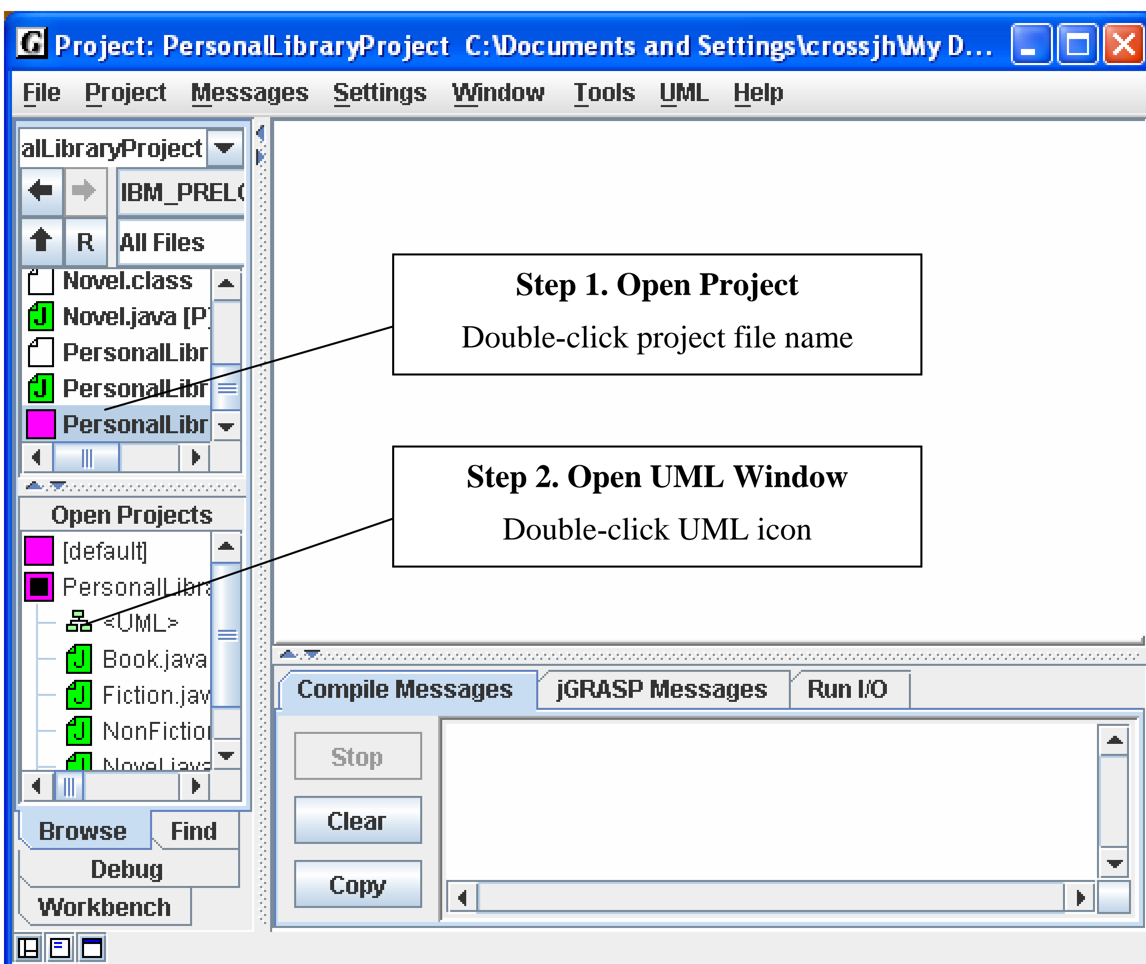


Figure 27. After loading file into CSD Window

3.4 The UML Window

In the figure below, the UML window has been opened for the PersonalLibraryProject and the class diagram has been generated. Notice the UML Window has its own set of menus across the top as well as its own toolbar with icons for launching operations. Below the toolbar is a panning rectangle, a set of scaling buttons, and an Update button 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. Anytime a UML or CSD window is opened, a button for it is placed on the windowbar. Clicking the button pops its window to the top. Windowbar buttons can be reordered by dragging them around on the windowbar.

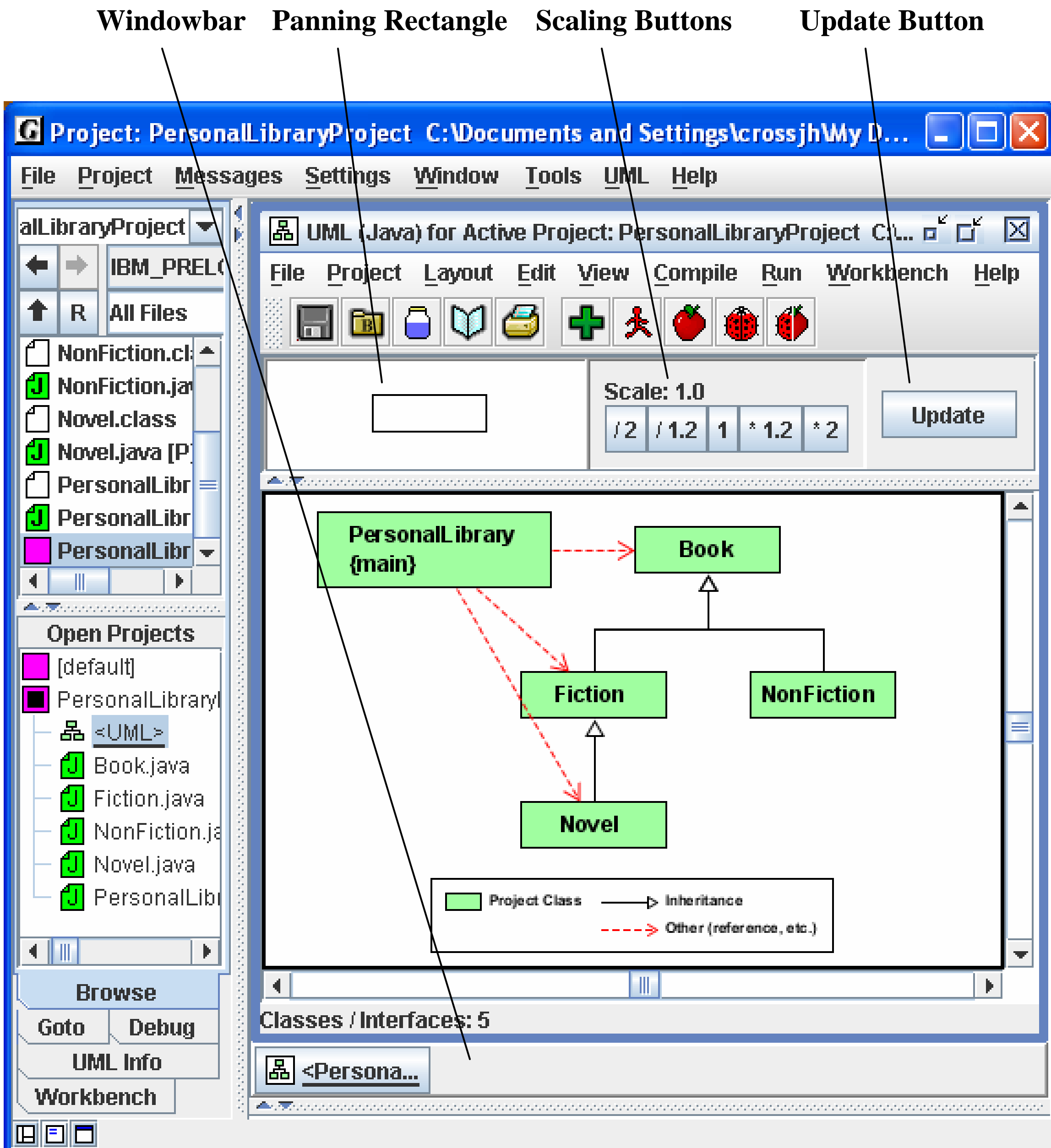


Figure 28. After opening the UML Window

3.5 Exploring the Features of the UML Window

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 Window.

3.5.1 Viewing the source code for a class

1. In the UML diagram, double-click on the PersonalLibrary class. This should open in the source file in a CSD window. Notice a button for 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 2.3-2.5 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.

3.5.2 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.

3.5.3 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.

3.6 Viewing the Source Code

To view the source code for a class in the UML diagram, simply double-click on the class symbol, or double-click the file name in the Browse tab under files or open projects. 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 the CSD Window has its own set of menus and toolbar icons across the top. These include Generate CSD, Remove CSD, and Number Lines as well as icons for the traditional Compile and Run.

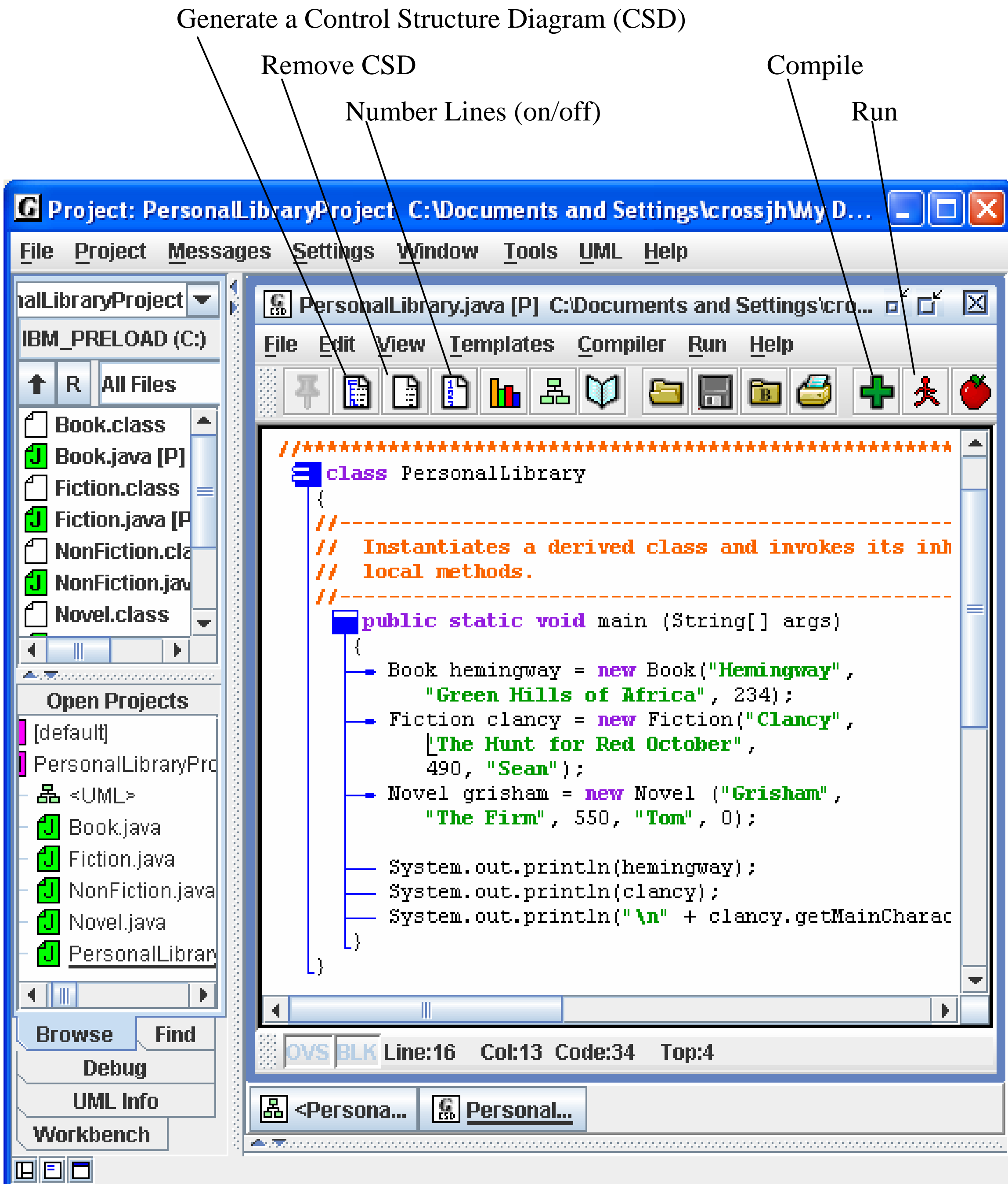


Figure 29. After the CSD is generated

3.7 Compiling and Running the Program

You can compile the files in the UML window by clicking the green plus as indicated in **Step 3** below. If at least one the classes in the diagram has a *main* method, you can also run the program as shown by **Step 3**. When you compile or run the program, the Compile Messages and/or Run I/O Tabs pop open to show the results.

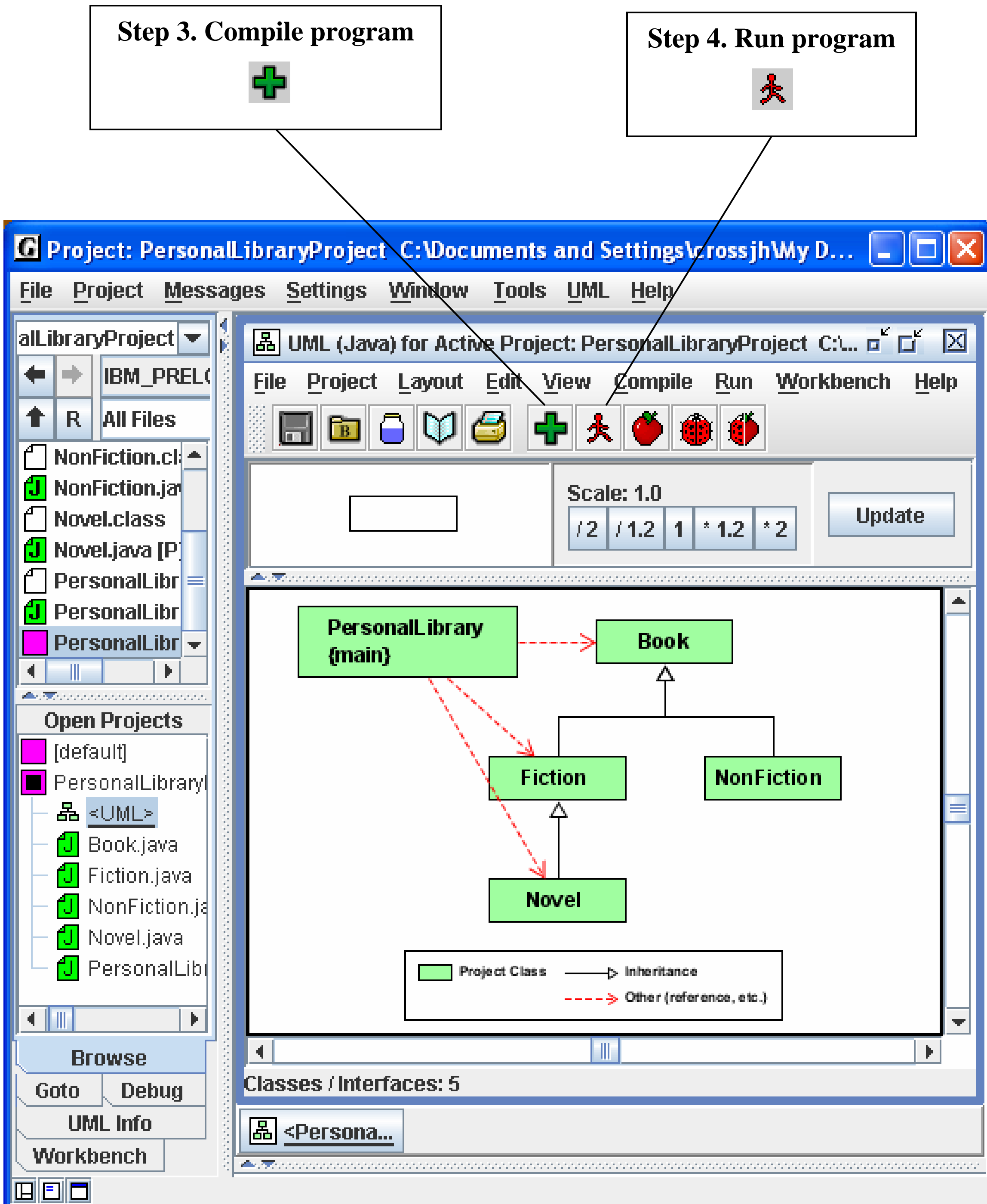


Figure 30. After loading file into CSD Window

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 on Desktop menu **Project -- Active Project <PersonalLibraryProject> -- Generate Documenation**). Or click the Generate Documentation icon on the UML window 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 PersonalLibaryProject_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. When you click **Generate** on the dialog, jGRASP calls the javadoc utility, included with the J2SDK, to create a complete hyper-linked document. The documentation generated for PersonalLibaryProject is shown below.

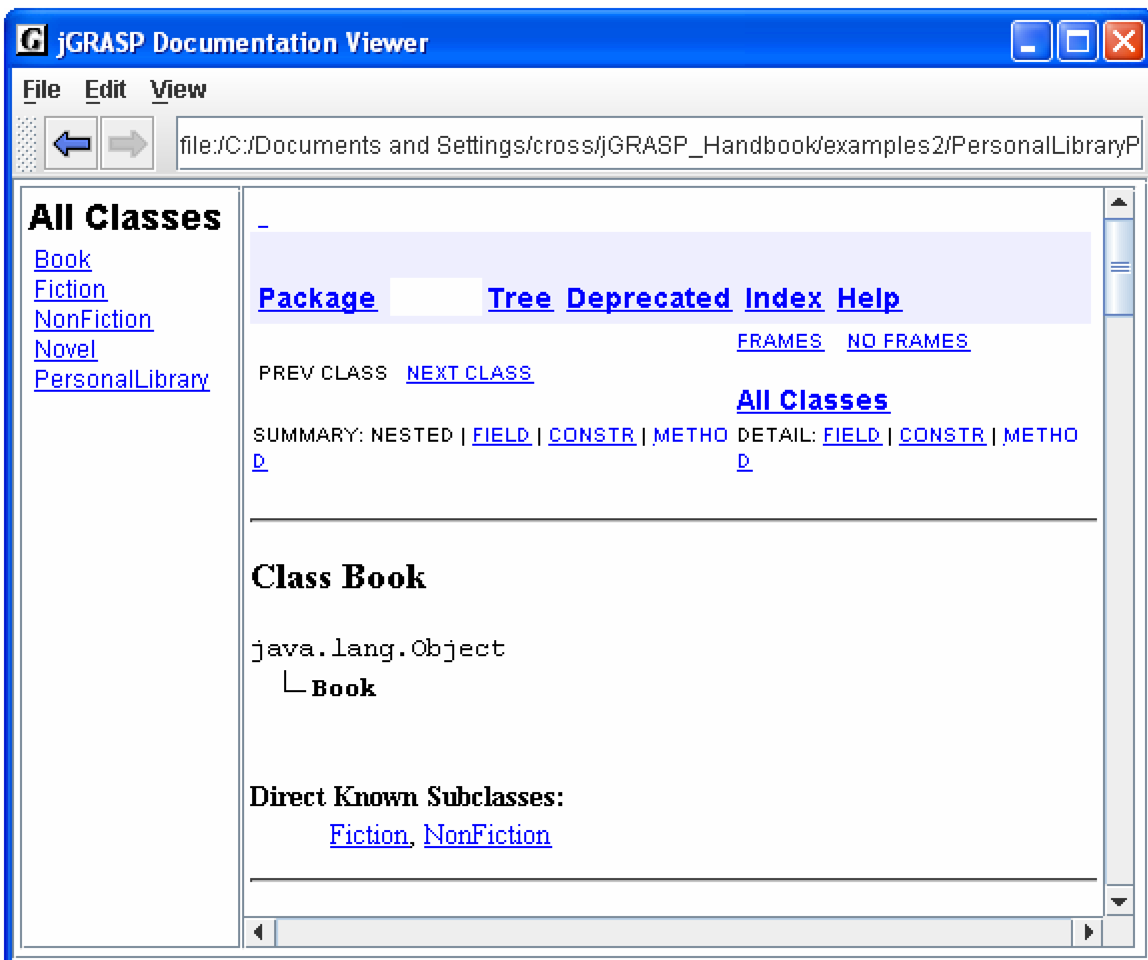


Figure 31. After generating documentation for PersonalLibaryProject

3.9 Using the Object Workbench

Now we are ready to begin exploring the Object Workbench. The figure below shows the PersonalLibraryProject loaded in the UML window. Above, we learned how to run the program in the traditional way as an application. Since *main* is a static method, we can also invoke it directly from the class diagram by right-clicking on PersonalLibary and selecting **Invoke Method**. When the Invoke Method dialog pops up, select and invoke *main* (with or 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 the figure below. A list of constructors will be displayed in a dialog box.

If a parameterless constructor is selected as shown in Figure 33, 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 34. The values for the parameters) should be filled in prior to clicking **create**.

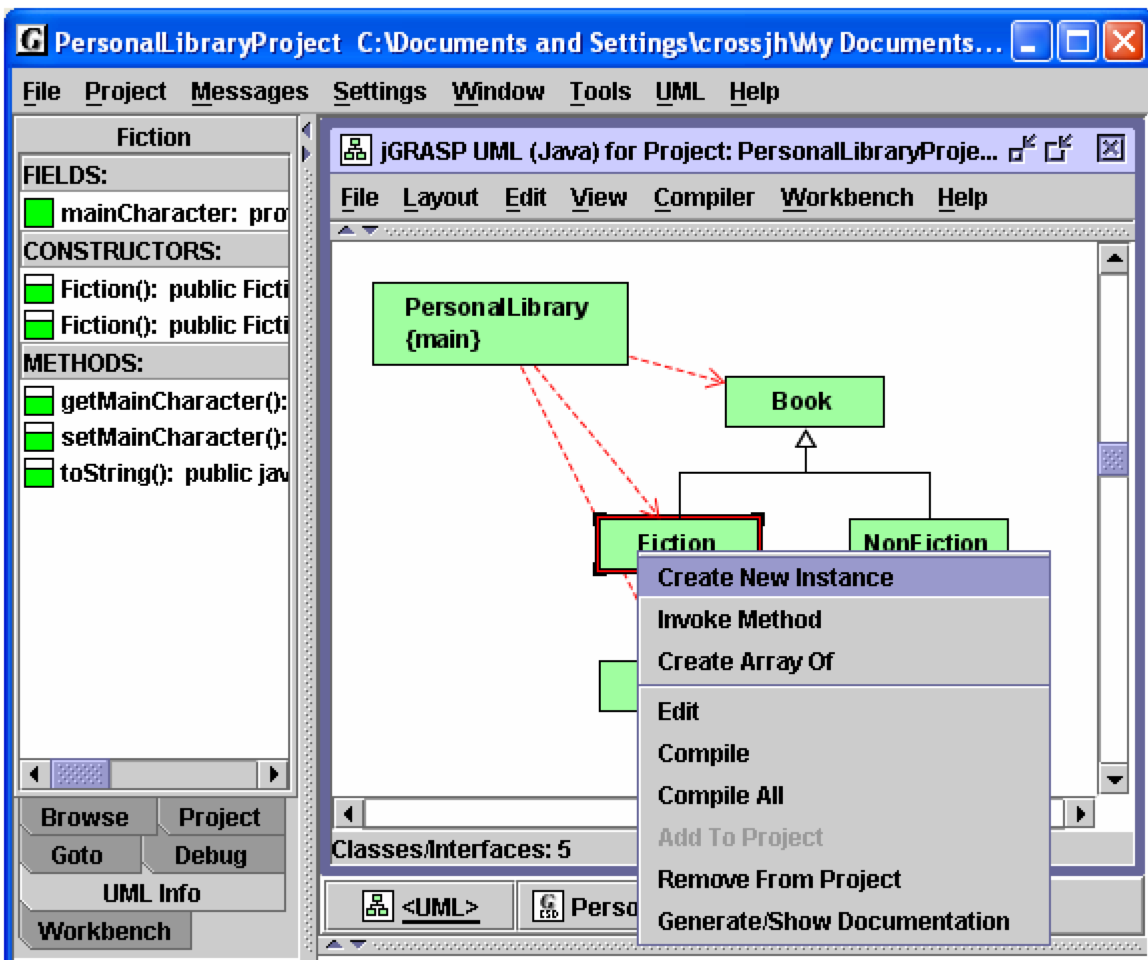


Figure 32. Creating an Object for the Workbench

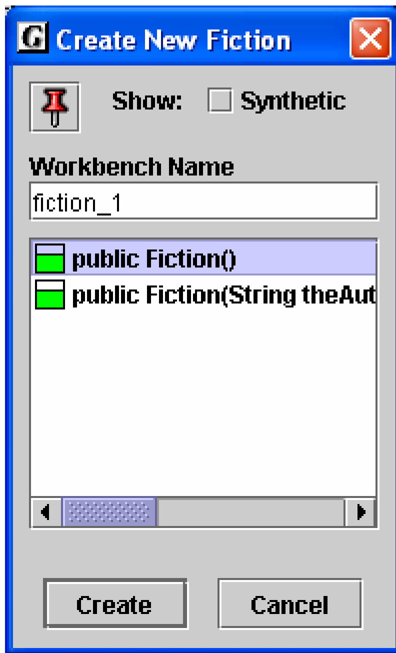


Figure 33. Selecting a constructor

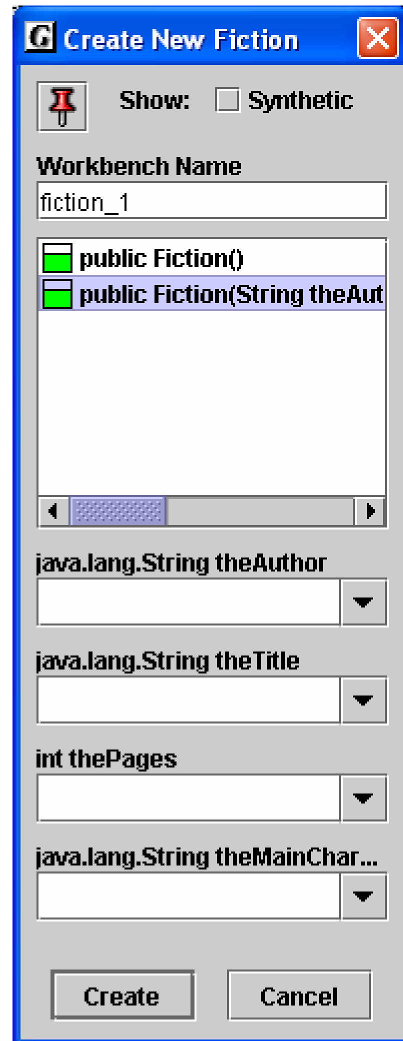


Figure 34. Constructor with parameters

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 remain open. This is convenient for creating multiple instances of the same class.

In Figure 35, the Workbench tab is shown after two instances of Fiction and one of Novel have been created. The first object, fiction_1, has been expanded so that the fields (mainCharacter, author, title, and pages) can be viewed. Since the first three fields are instances of the String class, they to can also be expanded. You should also note that mainCharacter is color coded green since it is the only field declared locally in Fiction. The other fields are color coded orange to indicate they were inherited from a parent, which in this case was Book.

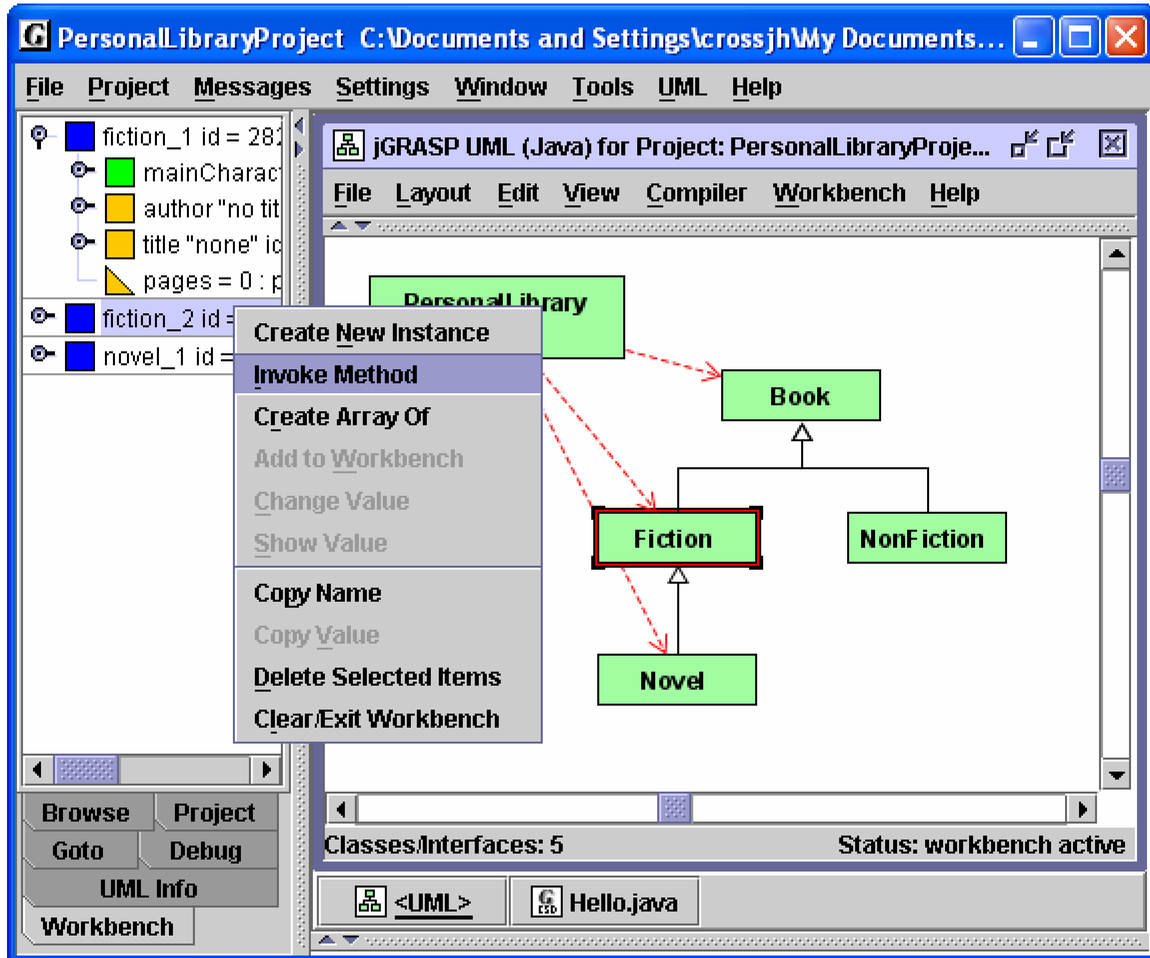


Figure 35. Workbench with two instances of Fiction

3.10 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 35, fiction_2 has been selected, followed by a right mouse click, and then Invoke Method has been selected. A list of local methods will be displayed in a dialog box as shown in Figure 36. You may also display inherited methods by selecting the appropriate parent. After one of the methods is selected and the parameters filled in as necessary, then click **Invoke**. This will execute the method and display the return value (or void) in a dialog, as well as display any

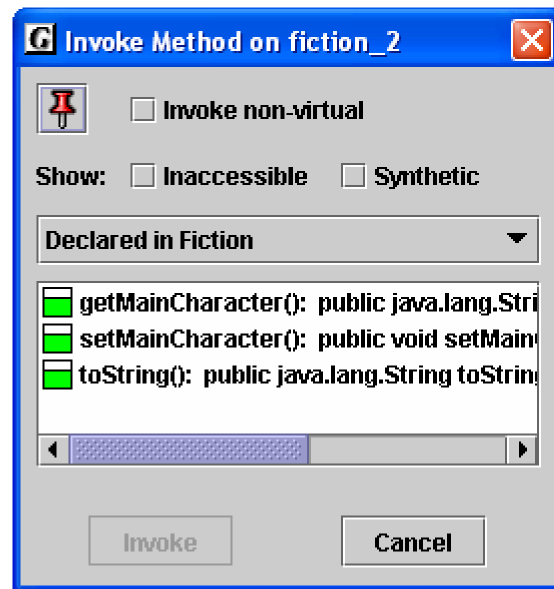


Figure 36. Selecting a method

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.

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 toStirng(). 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.

3.11 Invoking Methods with 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 simple 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.

3.12 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 35, fiction_1 had been expanded to show its fields (mainCharacter, author, title, and pages). Since the field mainCharacter is itself object of the class String, you can invoke any of the String methods. For example, right-click on mainCharacter, select **Invoke Method**. When the dialog pops up (Figure 37), scroll down and select the first *toUpperCase()* method and click **Invoke**. This should pop up the Result dialog with

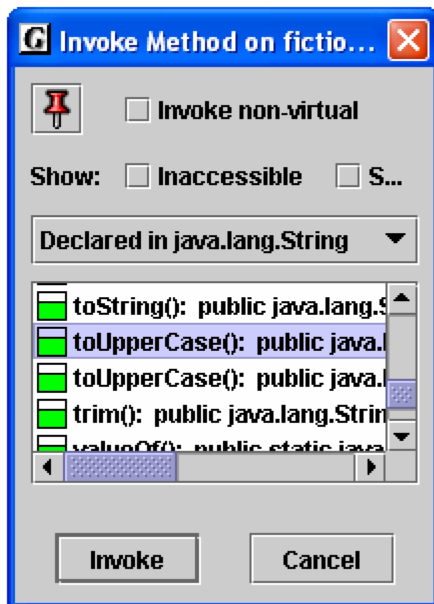


Figure 37. Invoking a String method

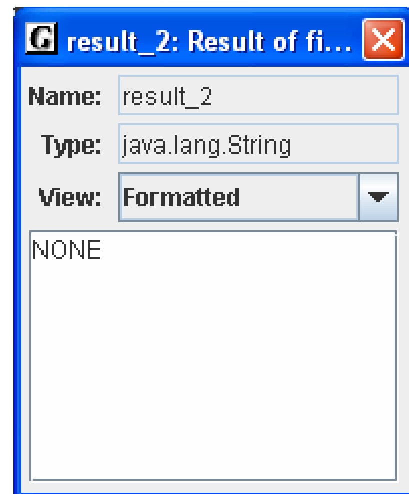


Figure38. Result of fiction_1.mainCharacter. toUpperCase()

"NONE" as the return value (Figure 38). 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.

3.13 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 35) and select **Invoke Method**, the dialog in Figure 39 pops up. However, only the toString() method is listed because it is the only one declared in Novel. To view inherited methods, find the pull-down menu located above the list. Notice it is currently set to "Declared in Novel". Right-clicking on the menu reveals all of the superclasses of Novel (Figure 40). Selecting "Declared in superclass Fiction" lists all methods inherited from Fiction (Figure 41). Notice the orange color coding indicating "inherited" similar to the fields on the workbench. However, in this case, toString() is gray to indicate it has been overridden by the toString() method declared in Novel.

Figure 39. Invoking a method for novel_1

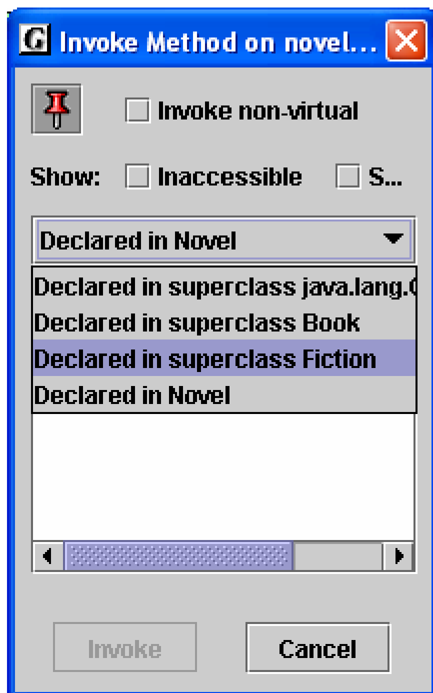


Figure 40. Viewing superclasses for novel_1

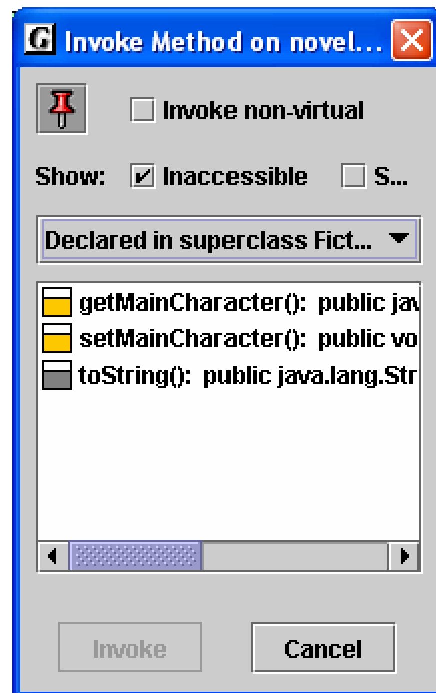


Figure 41. Viewing superclasses for novel_1

3.14 Running the Debugger on Invoked Methods

When objects are on the workbench, the workbench is running Java in debug mode. 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, examine fields, resume, etc. in the usual way. See the Tutorial entitled "The Integrated Debugger" for more details.

3.15 Creating 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.

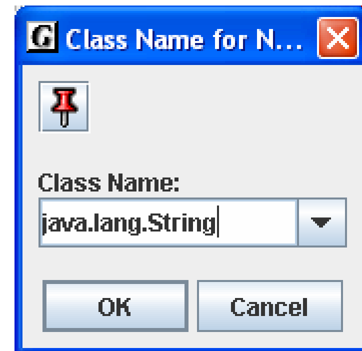


Figure 42. Creating an instance of String

3.16 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 (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 is it OK to end the process (Figure 43). 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**.

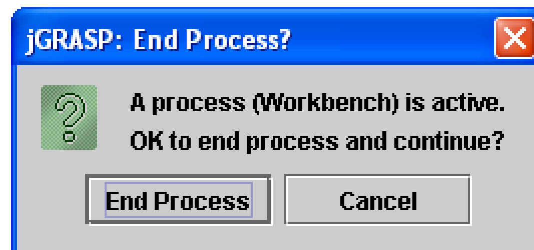


Figure 43. Making sure it is okay to exit

3.17 Closing a Project

If you leave one or more projects open when you exit jGRASP, when you restart jGRASP, they will be opened again. 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 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**.

After closing all projects, **Project: [default]** should be displayed at the top of the Desktop. All project information is saved when you close the project, as well as when you exit jGRASP.

3.18 Exiting jGRASP

When you have completed you 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.

Here are two ways to close or exit jGRASP:

(1) **The X Button** - You can exit jGRASP by clicking the Close button (**X**) in the upper right corner of the Desktop.

(2) **Desktop File Menu** – From the Desktop 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.

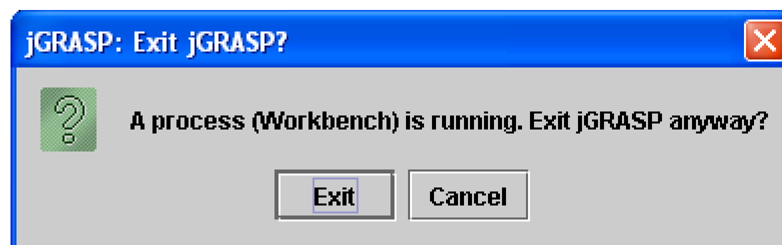


Figure 44. Making sure it is okay to exit

3.19 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. 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 icons on the UML toolbar.

  d. Invoke main() directly from the class diagram.

  e. Create three instances of Fiction and two of Novel.

  f. Invoke some of the methods for one or more of these 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 icon.

  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 a user designated group of files, which may be located in 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. If you have not created a project, **Project: [default]** should be displayed at the top of the Desktop, which indicates the jGRASP default project file is being used.

To use the UML and Object Workbench features of jGRASP, you must organize your Java files in a Project. UML generation and the Object Workbench are discussed in Sections 5 and 6. However, projects can be used independently of UML and Object Workbench features.

4.1 Creating a Project

On the Desktop toolbar, click **Project – New Project…** (Figure 45) to open the **New Project** dialog.

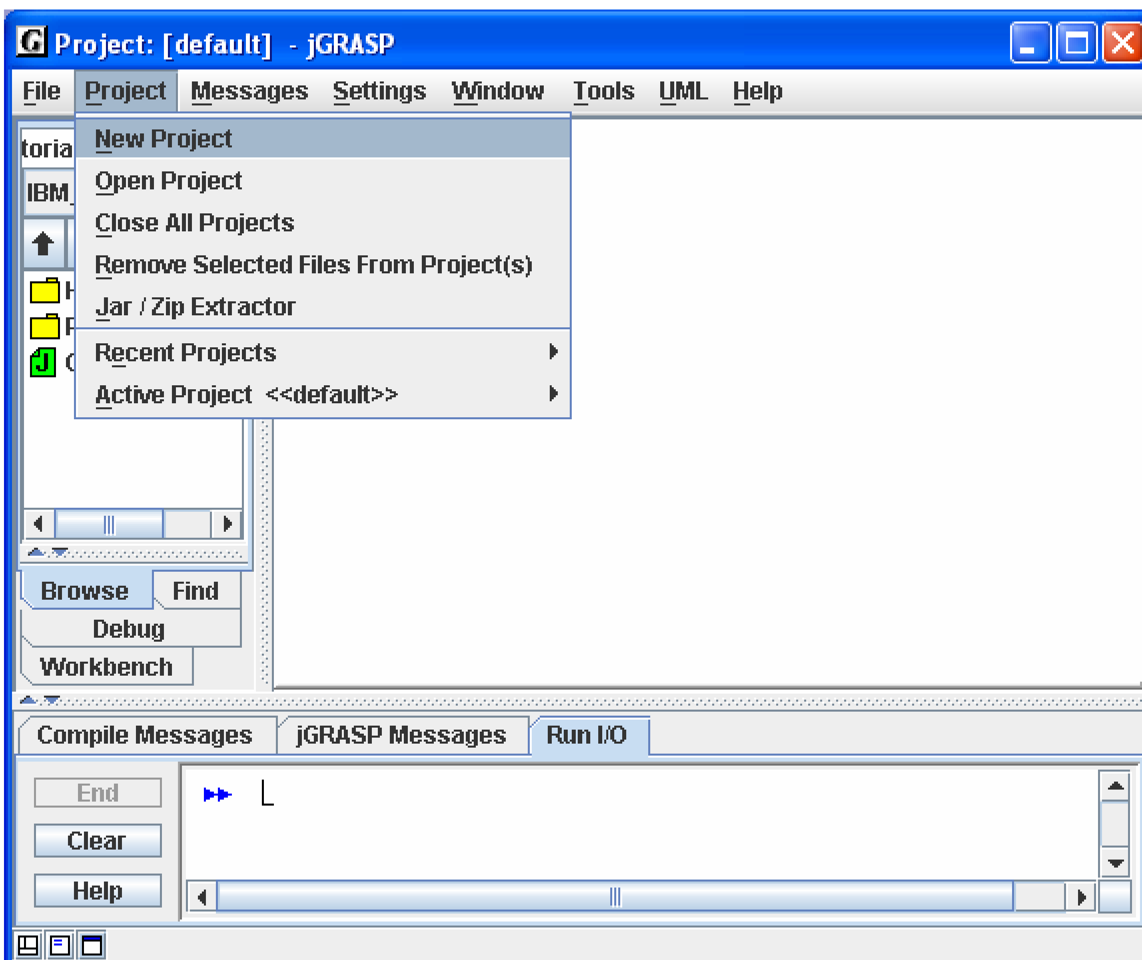


Figure 45. Creating a Project

Within the New Project dialog (Figure 46), 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 check ON the *Open UML Window* option. This will generate the UML class diagram and open the UML Window (see Section 5 for details).

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 in Java is *Classname*Project 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 PersonalLibaryProject.

After entering the project file name, click **Create** to save the project file. Notice the .gpj extension is automatically appended to the file name. As soon as the project is created, it becomes the current project, and the new project name replaces "[default]" at the top of the Desktop.

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.

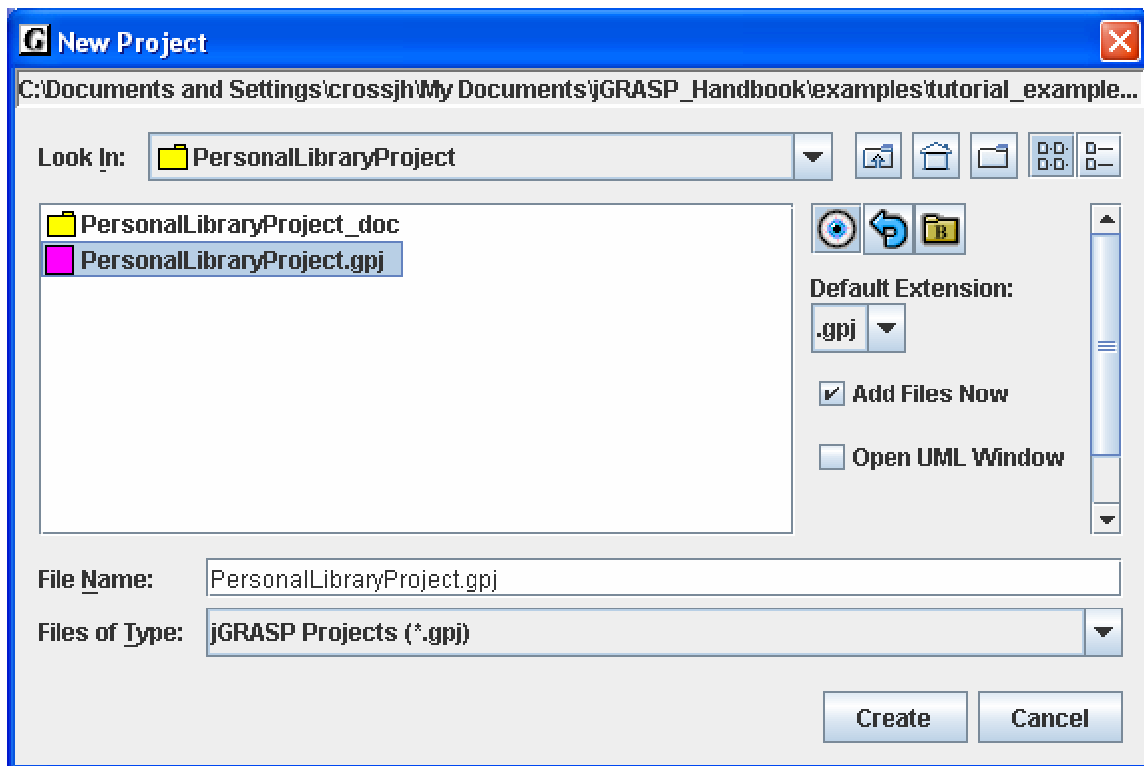


Figure 46. New Project window

4.2 Adding files to the Project

Beginning with v1.7, 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 47. After a project has been created, there are several ways to add Java files to it using the **Browse tab**.

(1) **In Browse Tab** - Drag the file (left click and hold) from directory at top to Open Project below.

(2) **From Browse Tab** - Drag the file from directory at top to UML Window (Section 5).

(3) **In Browse Tab** - Right click on the file and select **Add to Active Project – Relative Path**. (Figure 47).

(4) **From CSD window** – Click **File – Add to Active Project – Relative Path**.

Repeat this for each file to be added. You can also select multiple files (holding down the shift or control key), and add or drag the highlighted files all at once. To see the list of files in the project, select **Project** from among the tabs for **Browse**, **Project**, **Find**, and **Debug** at the bottom the left pane of the Desktop.

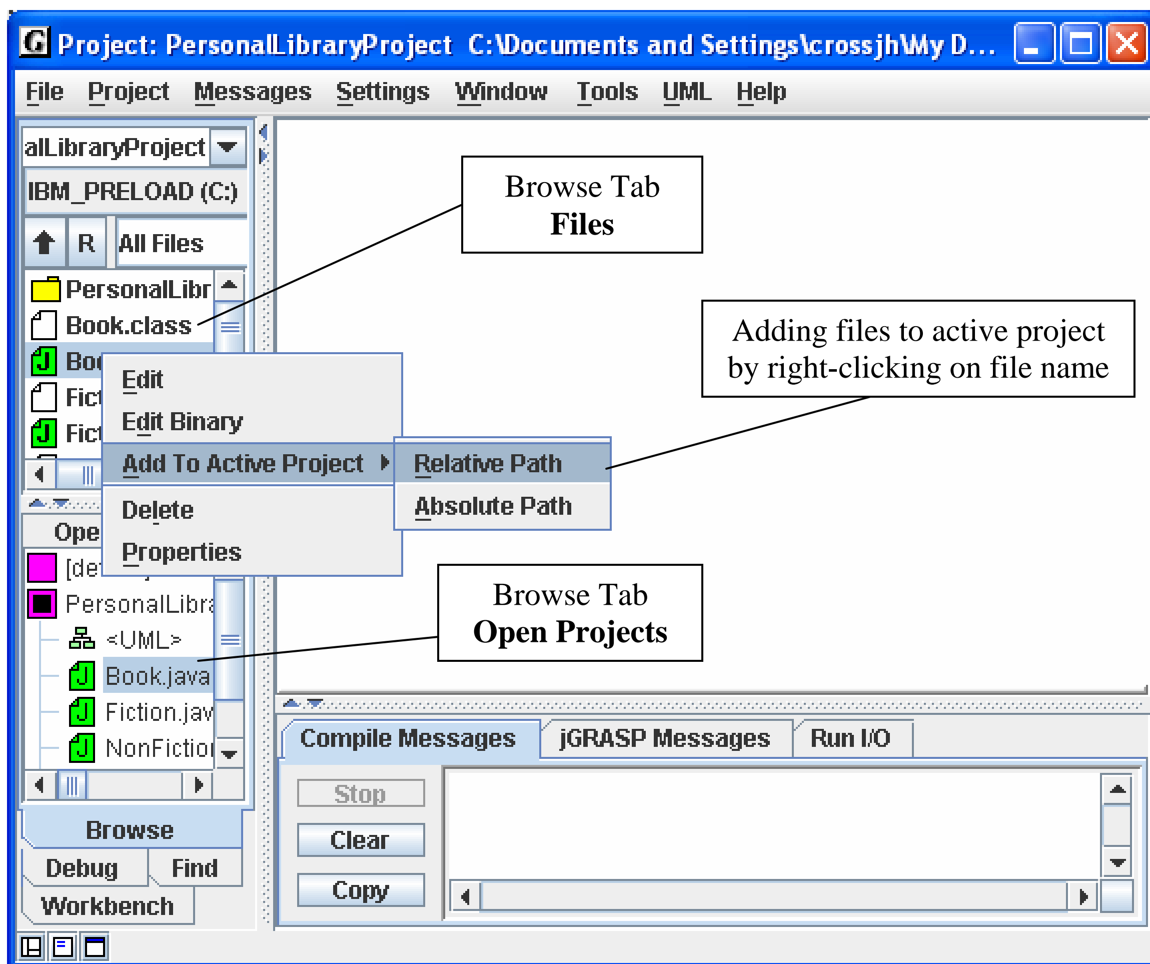


Figure 47. Adding a file to the Project

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 48. 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 the files section of the Browse tab and then pressing the *Delete* key.

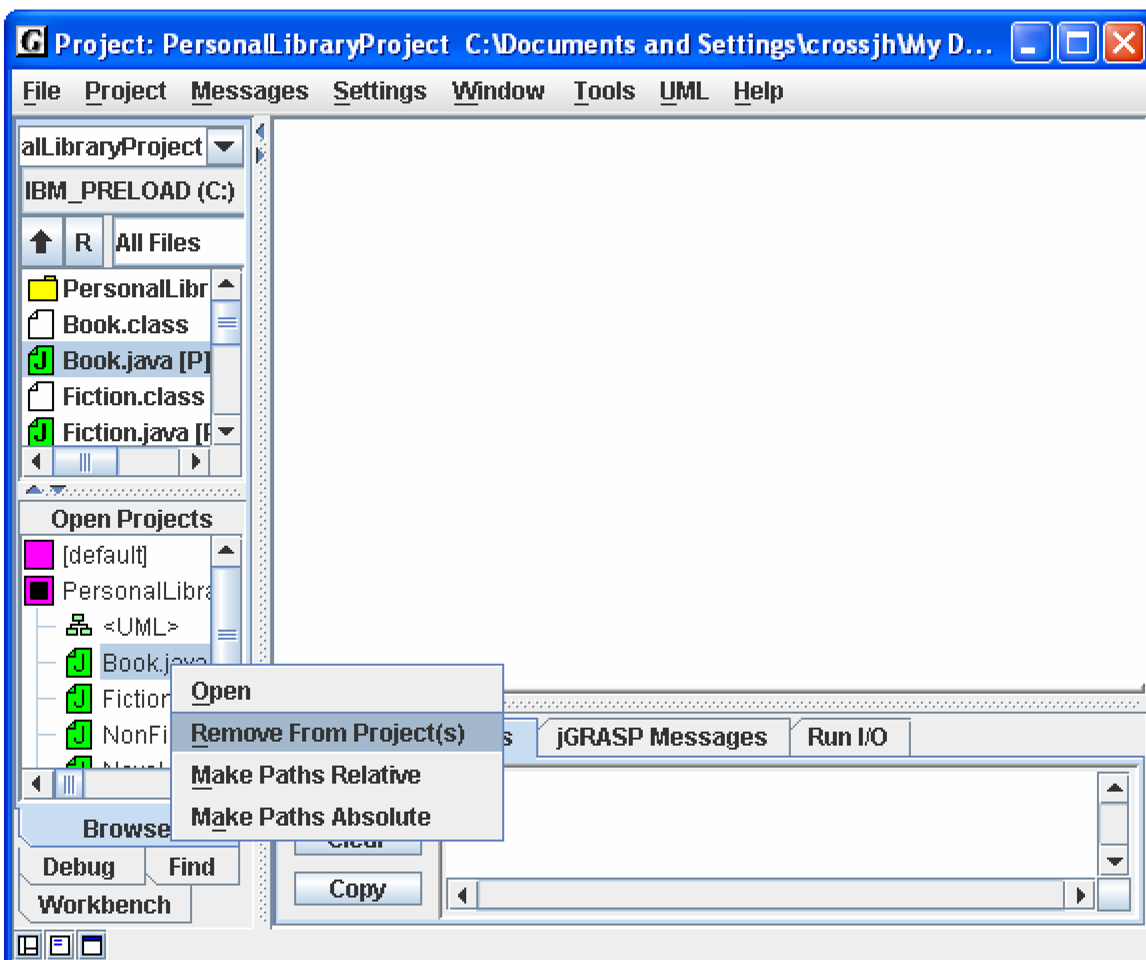


Figure 48. Removing a file from the Project

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 begin the process of generating the documentation, click on Desktop menu as shown in the Figure 49 below (**Project – Active Project <PersonalLibraryProject> -- Generate Documenation**). 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. So for the example, the default will be PersonalLibaryProject_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. Click **Generate** to start the process. jGRASP calls the javadoc utility, included with the J2SDK, to create a complete hyper-linked document within a few seconds.

This can be done on a file-by-file basis or for a complete project as shown in Figure 50 below. 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 more

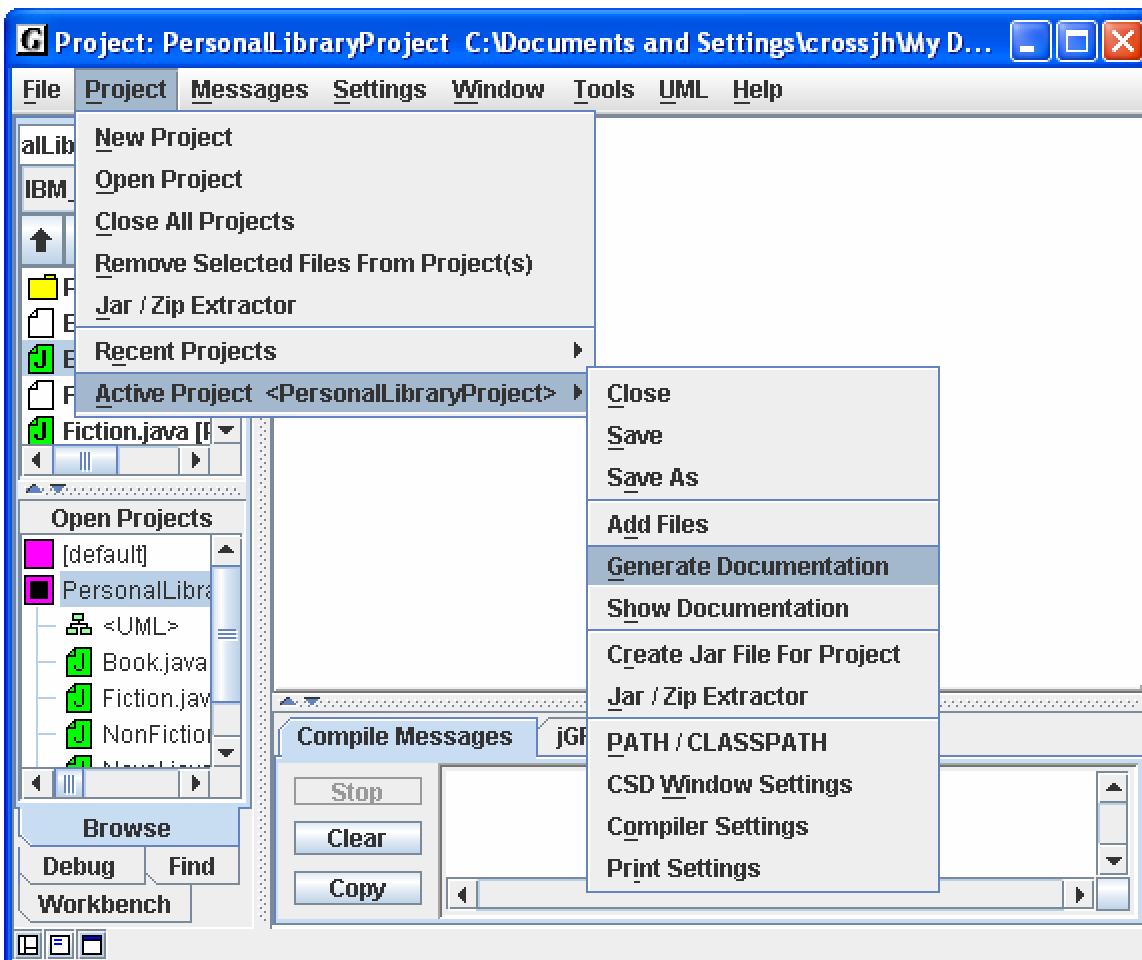


Figure 49. Generating Documentation for the Project

useful results, JavaDoc formal comments should be included in the source code. When the documentation is generated for an individual file, it is stored in a temporary directory for the duration of the jGRASP session. 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. If any changes are made to a project source file (and the file is resaved), jGRASP will indicate that the documentation needs to be regenerated; however, the user may choose to view the documentation files without updating them.

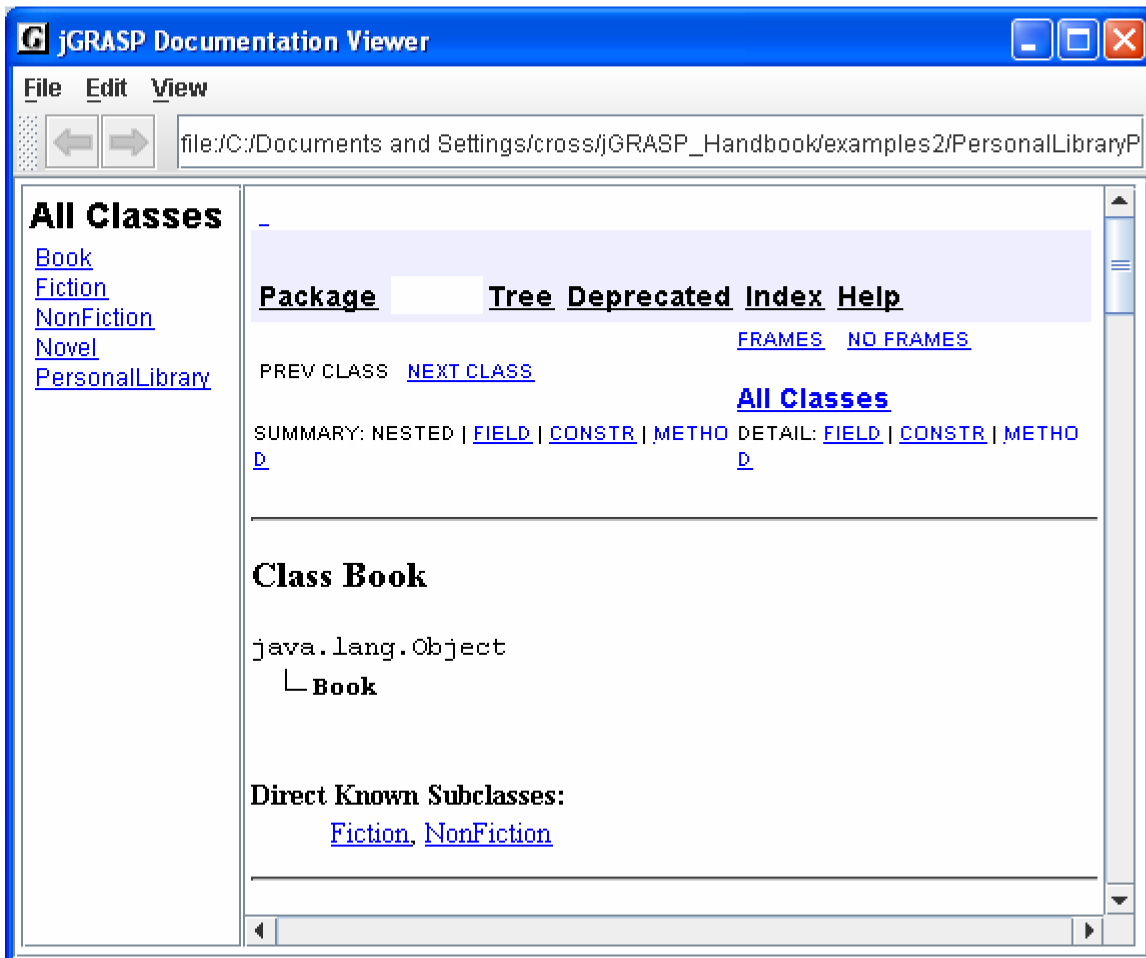


Figure 50. Project documentation

4.5 Jar file Creation and Extraction

jGRASP provides a utility for the creation and extraction of Java Archive files (JAR) for your project. The "Create Jar File for Project" option, which is found on the Project menu, allows you to create a single compressed file containing your entire project.

The "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 the ***jGRASP Handbook***, ***Part 2 – Reference***, and in jGRASP **Help**.

4.6 Active Project vs. Open Projects

Beginning with version 1.7, jGRASP allows you to have multiple projects open, but only one is designated as *active*. The small black square on the project icon in the project section of the Browse tab indicates the active project. By default, the last project you open is the active one. If you initiate your project operations from the Browse tab or from the UML window, it makes little difference which open project is active. However, if a Java file is in more than one open project and a project operation is initiated at the file level (e.g., from the CSD Window), the project settings, if any, from the active project will be used during the operation.

To change the active status from one project to another, right-click on the project name in the open projects section of the Browse tab and select **Make Active**. Of course, if you create a new project or open an existing project, then this project becomes the active one and its name is displayed at the top of the Desktop.

To open an existing project:

(1) From the Desktop toolbar - Click **Project – Open Project**.

(2) From the Browse Tab – Double-click on a project file in the files section of the Browse tab.

4.7 Closing a Project

(4) From the Desktop toolbar - Click **Project – Close All Projects**.

(5) From the Desktop toolbar - Click **Project – Active Project < > -- Close**.

(6) From the Browse Tab – Right-click on the project file name in the Open Projects section of the Browse tab and select **Close**.

After closing a project, **Project: [default]** should be displayed at the top of the Desktop. All project information is saved when you close the project, as well as when you exit jGRASP.

4.8 Exercises

(8) 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.

(9) After the new project is created, 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.

(10) Remove a file from PersonalLibraryProject2. After verifying the file was removed, add it back to the project.

(11) 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.

(12) Close the project.

(13) 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). The jGRASP project file is used to determine which user classes to include in the UML class diagram.

In order to generate a UML diagram, you must create a jGRASP project file (.gpj), as described in the previous section, if you haven't already done so. 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. In particular, 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.

The remainder of this section assumes you have created a project file, and that you have compiled your program files.

5.1 Opening the Project

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.

(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 additional help with opening a project, review the previous section.

5.2 Generating the UML

In Figure 34 below, PersonalLibraryProject is shown in the open projects section of the Browse tab along a **UML icon** and the files in the project. To generate the UML class diagram, double-click the UML icon. Alternatively, on the Desktop menu, click on **UML**, then **Generate/Update Class Diagram for Active Project**.

The UML Window should open with a diagram of all class files in the project as shown in Figure 50. 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 in 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** button.

If your project includes class inheritance hierarchies and/or other dependencies as in the example, then you should see the appropriate red and black dependency lines. Next, you

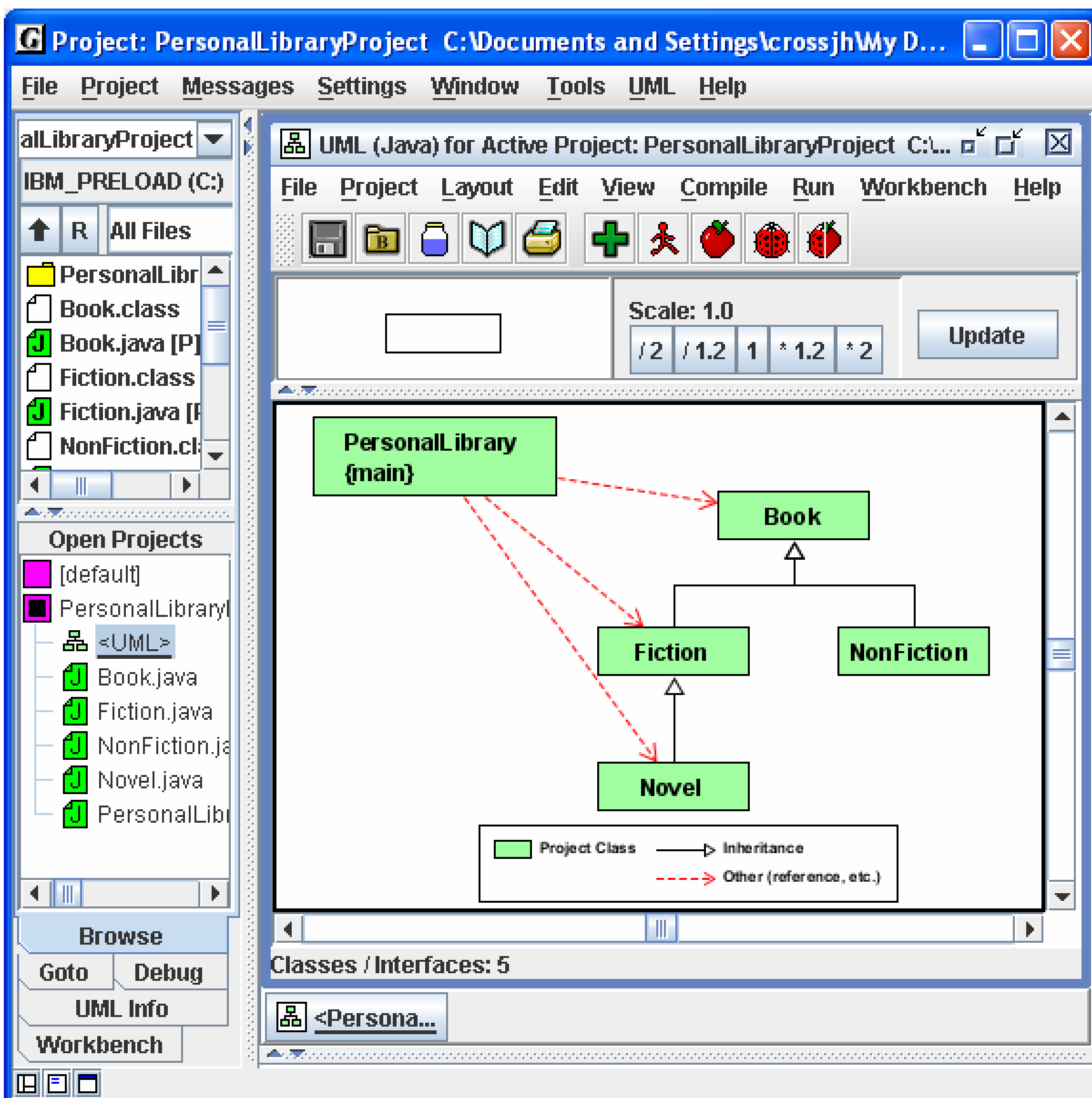


Figure 51. Generating the UML

will need to learn how to indicate which objects and dependencies you want in your UML diagram. However, if you are okay with the classes and dependencies shown in the diagram, then go to the next section and begin laying out your diagram.

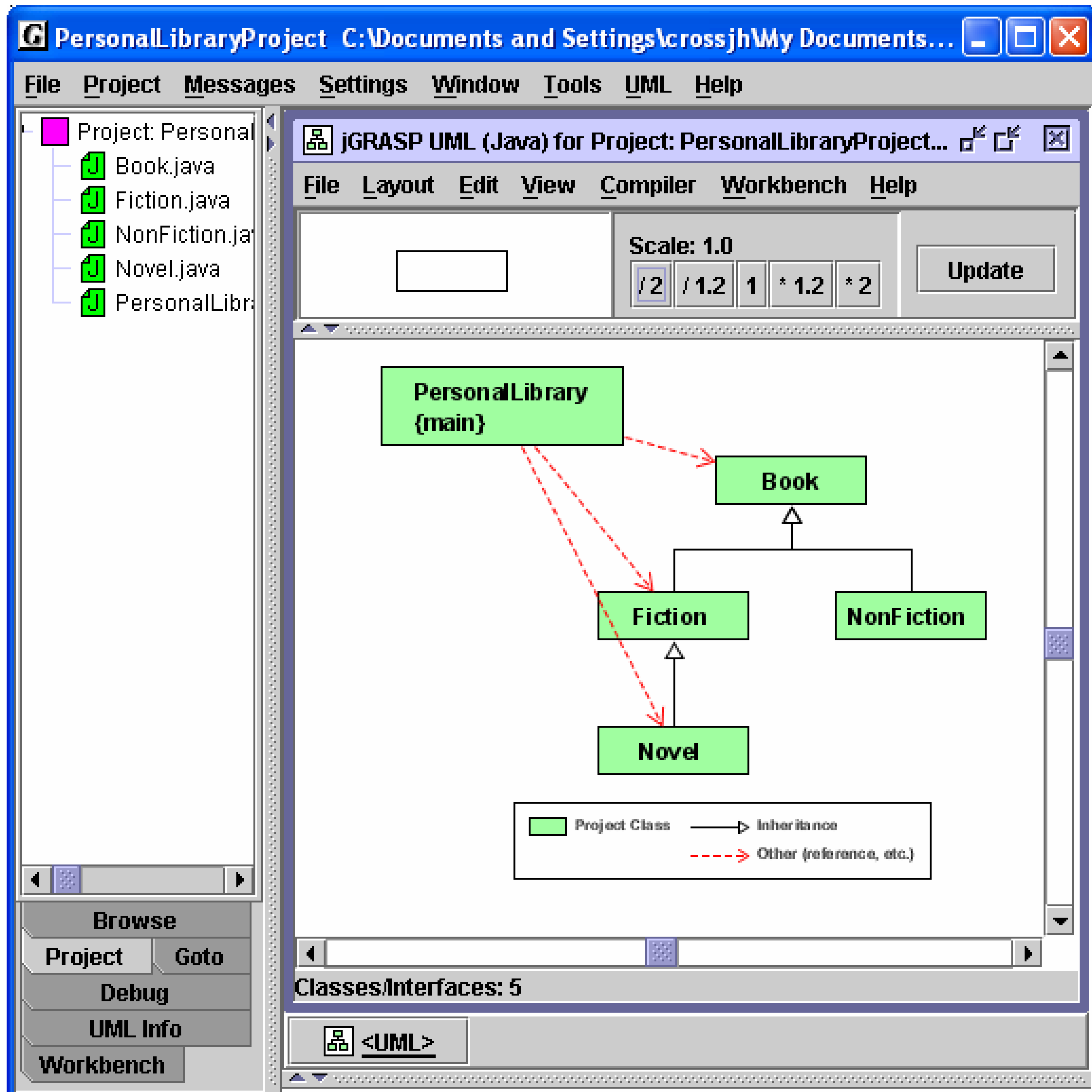


Figure 52. UML Window after initial Generate

5.3 Determining the Contents of the Class Diagram

jGRASP provides two approaches to controlling the contents/display of your UML diagram. The first (**Edit – 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). After you generated/updated the diagram based on these exclusions, the second approach (**View --**) allows you to make visible (or hide) certain categories of classes and dependencies in the UML diagram. Both approaches are described below.

Suppose you want to include the JDK classes (gray boxes) in your UML diagram (the default is to exclude them). Then you will need to edit 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 **Edit – Settings…**, which will bring up the **UML Settings** dialog. For example, to not exclude all JDK classes, under **Exclude by Type of Class**, uncheck (turn OFF) the checkbox that excludes **JDK Classes,** as shown in Figure 39. Note, synthetic classes are created by the Java compiler, and are usually not included in the UML diagram. After checking (or unchecking) the items you want excluded (or not), click the OK button, which should close the dialog and update the diagram. All JDK classes used by the project classes should now be visible in the diagram as gray boxes. This is shown in Figure 53 after the JDK classes have been dragged around. To remove them from the diagram, you would need to turn on the exclude option and update the diagram again. If you want to leave them in the diagram but not display them see the next paragraph. For more information see UML Settings in the Reference section.

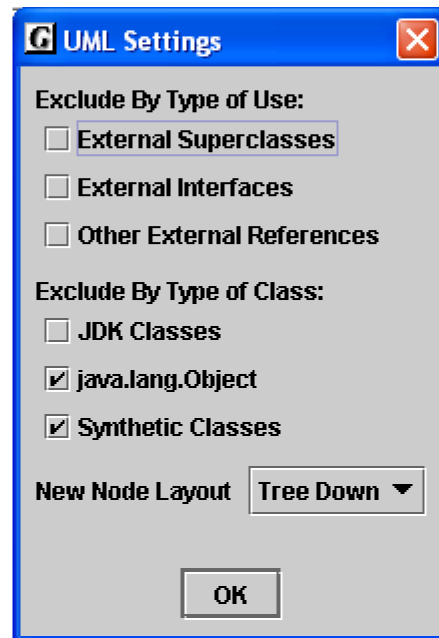


Figure 53. Editing the UML Settings

Making objects visible in the diagram - On the UML window menu, click on **View – Visible Objects**, then check or uncheck the items on the list as appropriate. 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. In general, you probably want all of the items on the list checked on as shown in Figure 54. Note that if items have been *excluded* from the

diagram, 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 the Reference section.

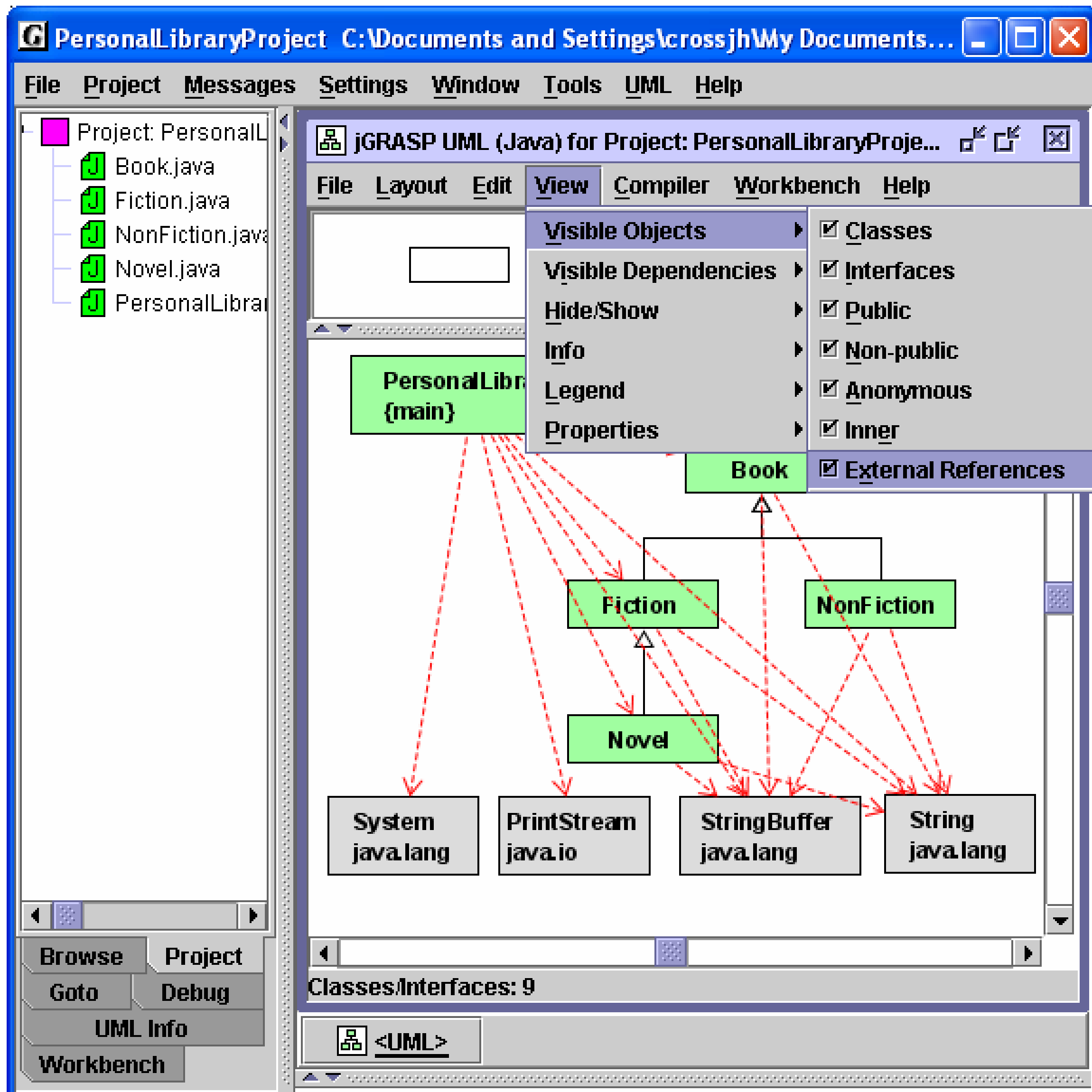


Figure 54. Making objects visible

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 child to the parent to from an *is-a relationship*. Red dashed lines with open arrowheads indicate **other dependencies**. These include the *has-a relationship* that indicates a class includes one or more instances of another class. If a class references an instance variable or method of another class, the red dashed arrow is drawn from the class where the reference is made to the class where the referenced item is defined. In general, you probably want to make all dependencies visible. as indicated in Figure 55.

Displaying the Legend - On the UML window menu, click on **View – Legend**, then set the desired options. Typically, you will want the following options checked on: Show Legend, Visible Items Only, and Small Font. Notice, the legend has been visible in the

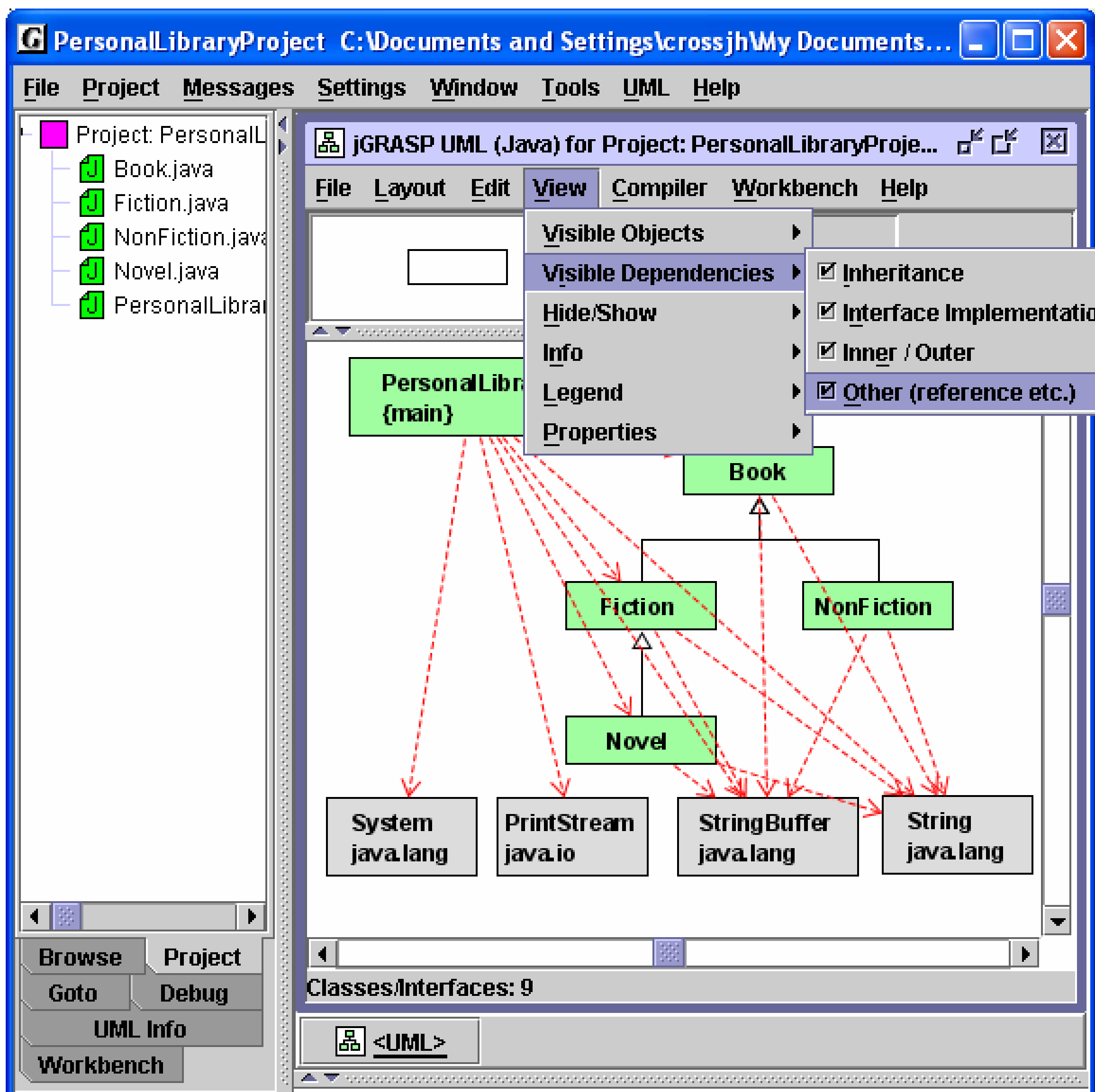


Figure 55. Making dependencies visible

all of UML figures. Before the JDK classes were excluded (Figure 54), they were included in the legend, but not after the Update. 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.

5.4 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, on UML window menu, click on **Layout – Tree Down** to perform the operation, or right-click on a selected class then on the pop up menu select **Layout – Tree Down**.

With a one or more classes selected, you can move them as a group. Figure 55 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 of 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.

5.5 Displaying the Members of a Class

If you are working with an example laying out your UML diagram as described in the section above, when you select a class, you may have noticed that the class members (fields, constructors, and methods) were displayed in **Info** tab of the left pane of the UML Window. In Figure 56, 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 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.

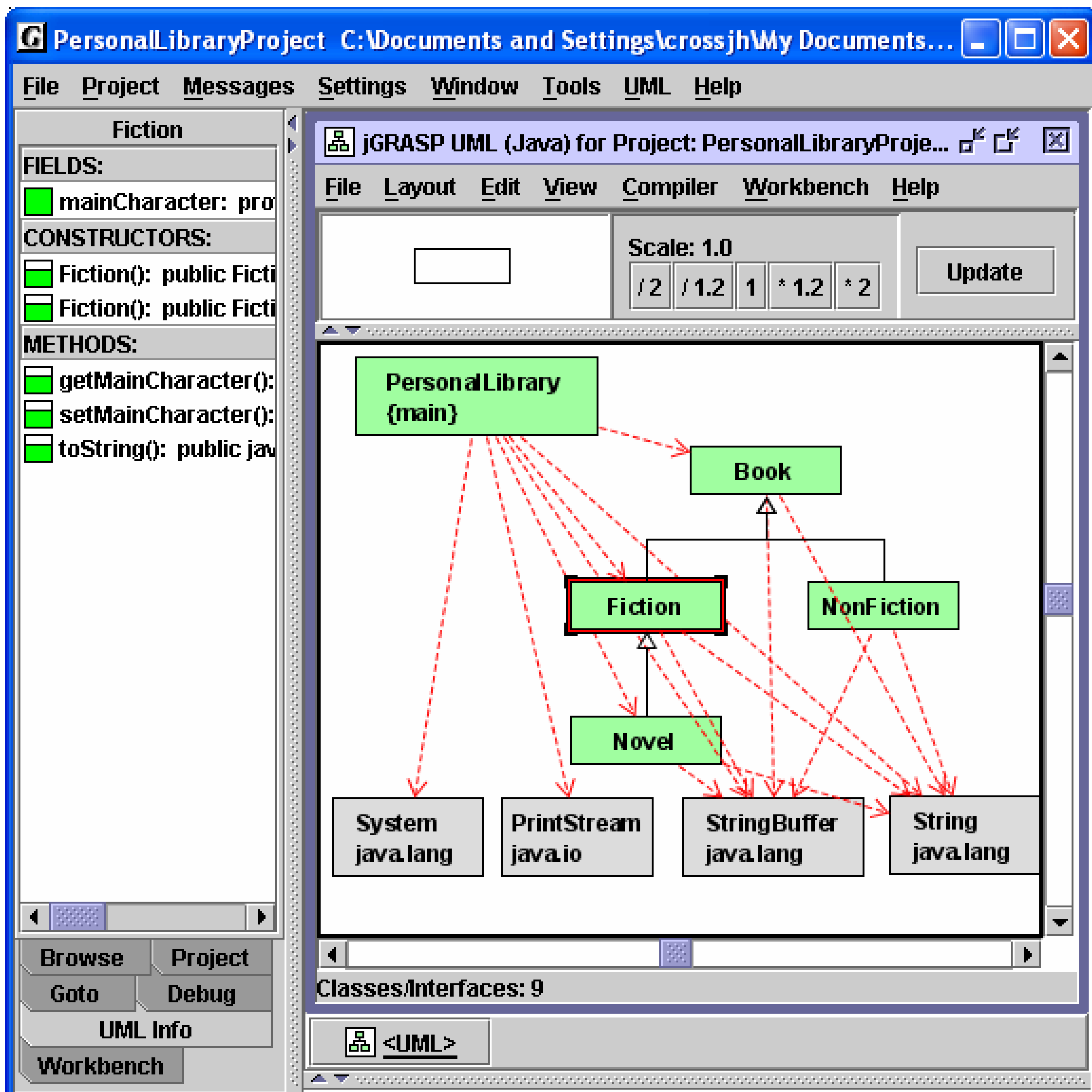


Figure 56. Displaying class members

5.6 Displaying Dependencies Between Two Classes

An arrow or edge between two classes in the UML diagram indicates that there are one or more dependencies in the direction of the arrow. In Figure 57, the edge drawn from PersonalLibrary to Fiction has been selected, indicated by the large arrorhead. The list of dependencies in the Info tab of the left pane includes one constructor (Fiction) and one method (getMainCharacter). These are the resources that PersonalLibrary uses from Fiction. Reviewing all of the dependencies among the classes in your object-oriented program will usually prove insightful and provide you with a more in-depth understanding of the source code.

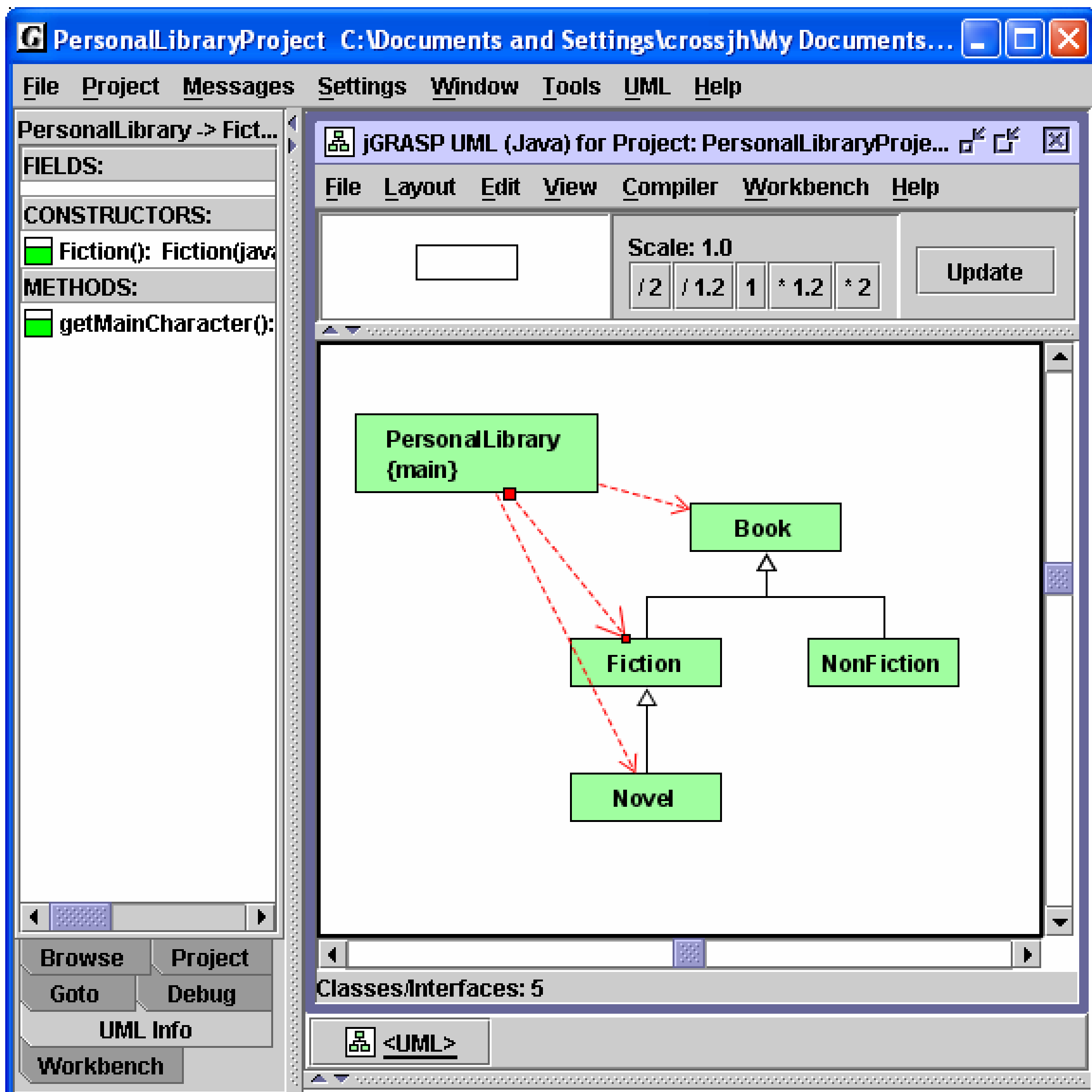


Figure 57. Displaying the dependencies between two classes

5.7 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 diagram.

5.8 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. For example, when the PersonalLibrary class is double-clicked, the corresponding CSD Window is opened on the Desktop as shown in Figure 58.

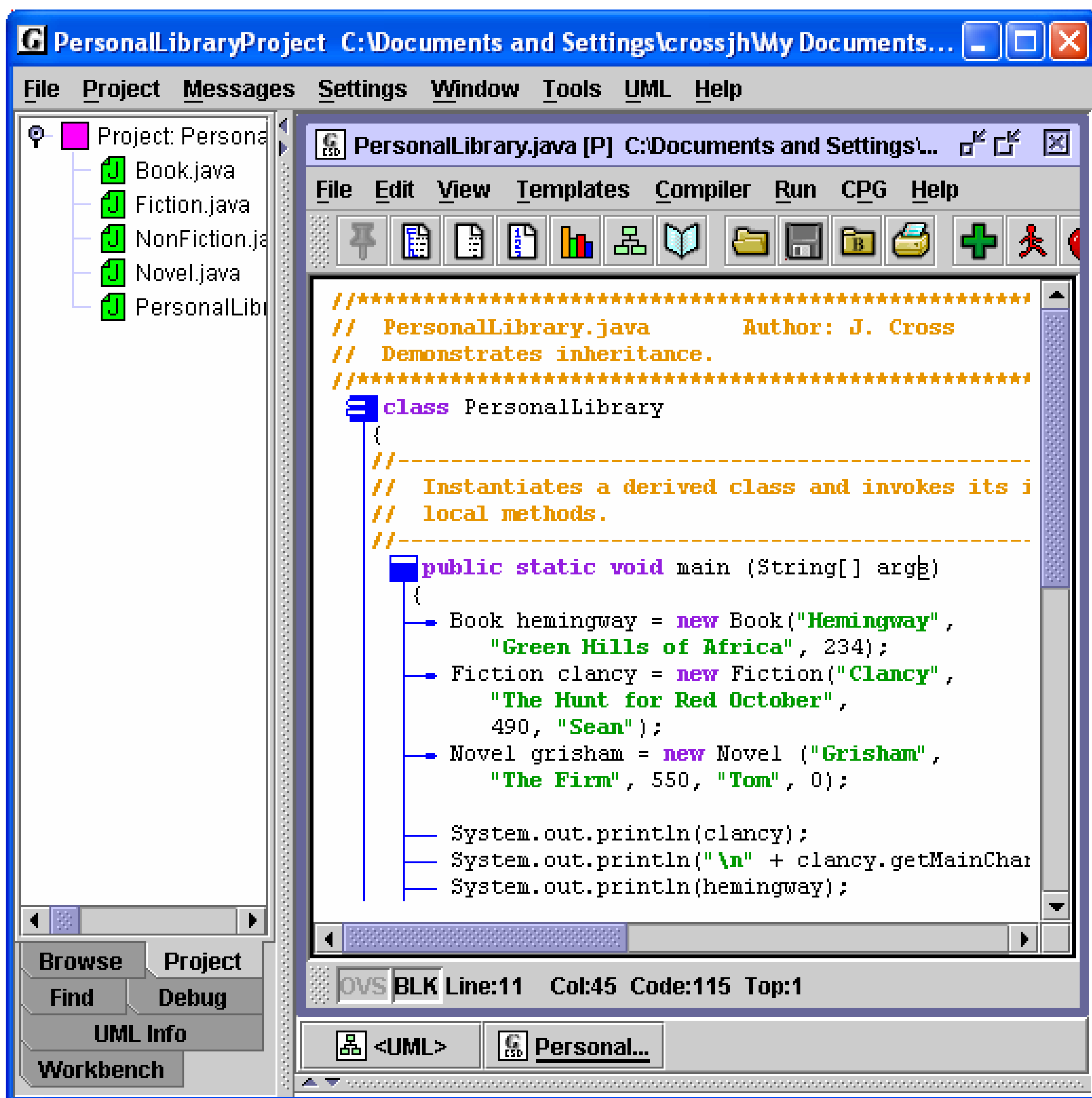


Figure 58. Opening CSD Window from UML

5.9 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 the .java and .class files, and if you added the source files with *relative paths*, then you should be able to ship the files around (e.g., email them to your instructor).

5.10 Printing the UML Diagram

On UML window menu, click on **Print – Print Preview** to see show 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 Dependency Diagrams in the ***jGRASP Handbook***, ***Part 2 – Reference,*** or in jGRASP **Help**.

6 The Object Workbench

Beginning with Version 1.6, jGRASP provides an Object Workbench that works in conjunction with the UML class diagram, as well as the integrated debugger. The Object Workbench is a useful approach for learning the fundamental concepts of classes and objects. The user can create instances of any class in the UML diagram as well as instances of any Java class. 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 class or *static* 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, without the need for a driver program intended to test each method.

6.1 Invoking Static Methods

We begin with a typical first program, Hello.java, which has a *main* method. We have created a project file, HelloProject, added Hello.java to the project, then generated the UML class diagram. Also, we have elected to display the Java classes used by the Hello class (**Edit** -- **Settings** -- uncheck "Exclude JDK classes") as shown in Figure 59.

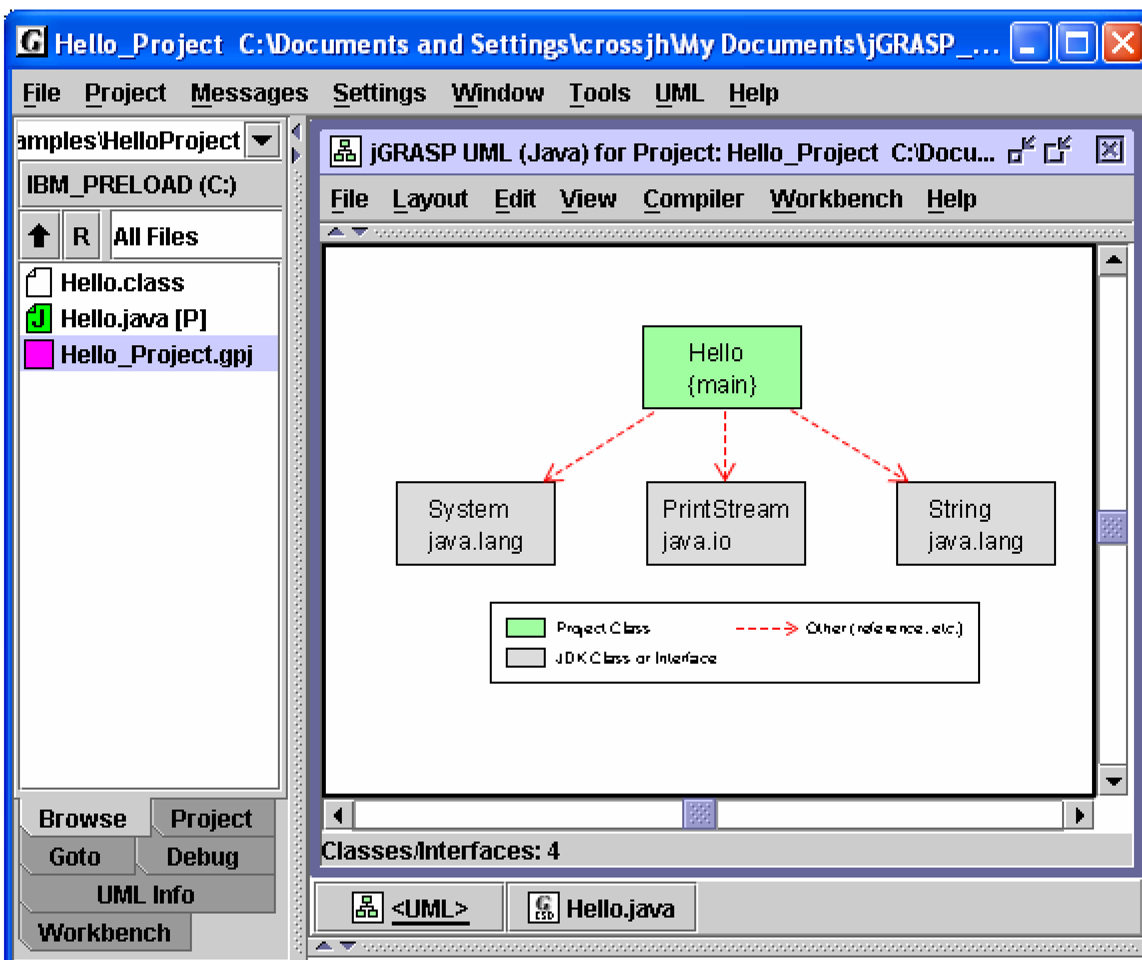


Figure 59. Creating an Object for the Workbench

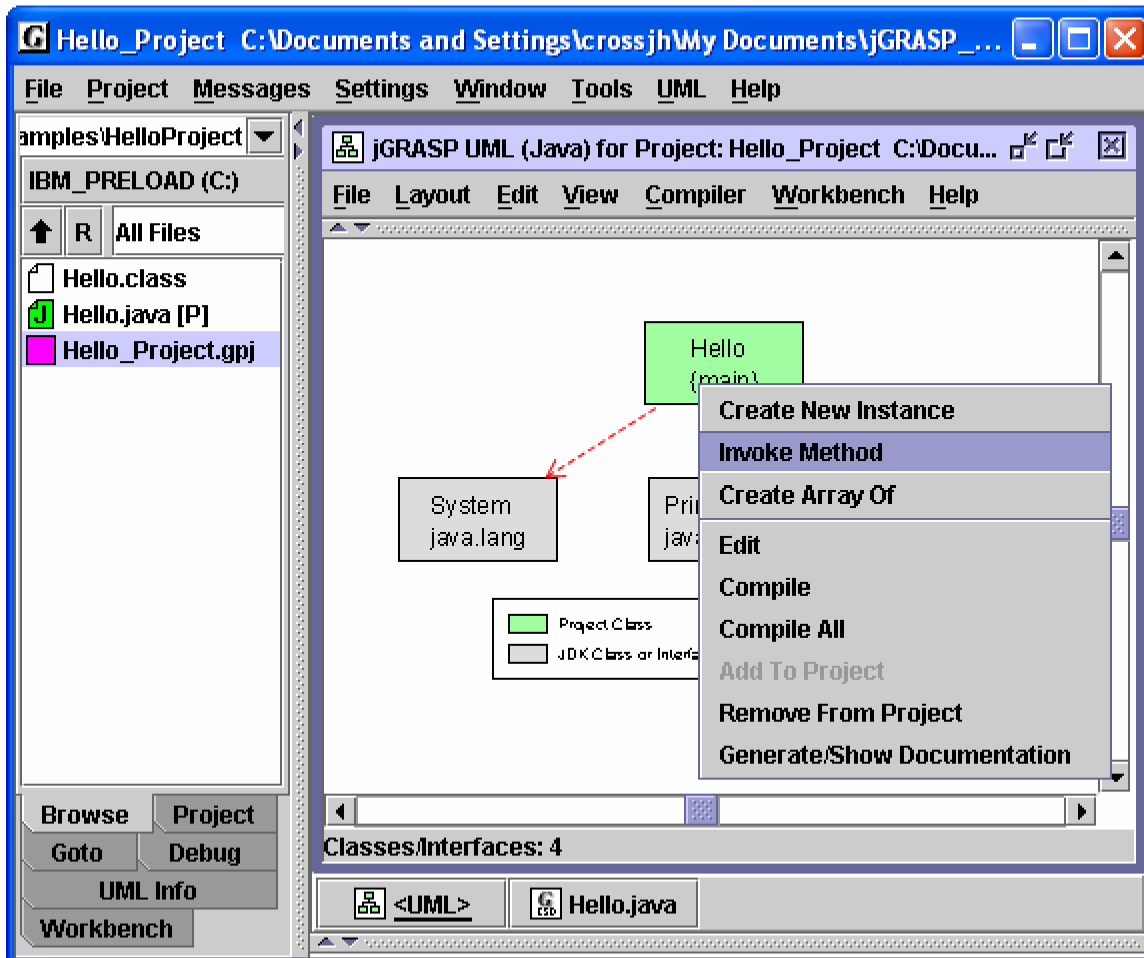


Figure 60. Invoking a static method from a class

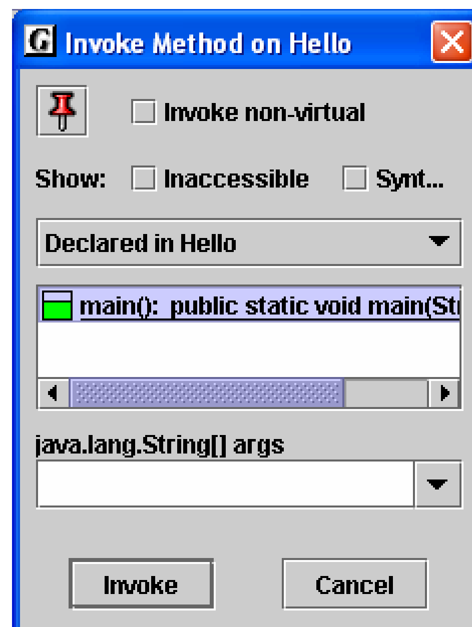


Figure 61. Invoking *main*

Since *main* is a static method, it is associated with the class rather than an instance of the class. Therefore, we right-click on the Hello class symbol, then select **Invoke Method** (Figure 60). 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 61). We can leave the *java.lang.String[] args* blank since our *main* method is not expecting any command line arguments to be passed into it.

Now you are ready to invoke the main method by clicking **Invoke** in the lower left corner of the dialog. Figure 62 shows the results of the invocation. First, notice that a dialog pops up entitled, "Result of Hello.main…" with the

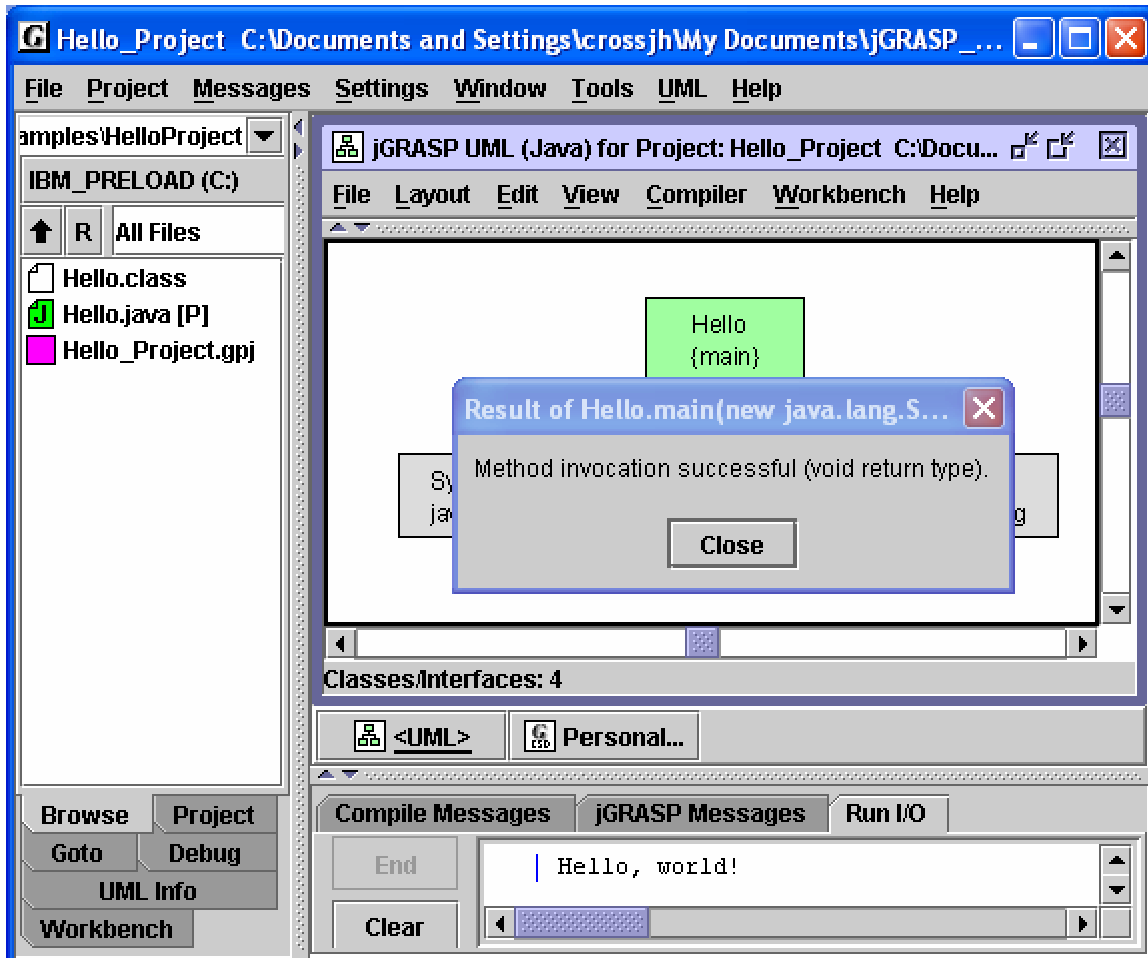


Figure 62. Invoking a static method from a class

message "Method invocation successful (void return type)." You will see a Result dialog each time you invoke a method from the workbench. In most cases it will contain an actual value of either a primitive type or reference type since most methods do not have a return type of "void" as does our *main* method.

In addition to a return value, a method invocation may produce standard output or even pop up a GUI frame of its own. In our example, *main* produces standard output with the statement, *System.out.println("Hello, world")*, so these results can be seen in the Run I/O pane in the lower part of the jGRASP desktop.

As stated at the beginning of this section, static methods are associated with a particular class rather than a specific object of the class. Hence, static methods are invoked by selecting a class in the UML diagram. If you attempt to invoke a method for a class that has no static methods, the Invoke Method dialog will indicate this. When a class has no static methods, but rather has instance methods, you will need to create an instance of the class. In the next several sections, you will learn how to create instances or objects of a class from the UML diagram and then how invoke a particular object's methods from the Workbench tab pane by selecting the newly created object.

6.2 Creating an Object for the Workbench

Now we move to a more interesting example which contains multiple classes. Figure 52 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 PersonalLibary and selecting **Invoke Method**, then selecting and Invoking *main*). However, the focus of this section is to create an object for 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 63. A list of constructors will be displayed in a dialog box.

If a parameterless constructor is selected as shown in Figure 64, 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 65. The arguments (values of the parameters) should be filled in prior to clicking **create**.

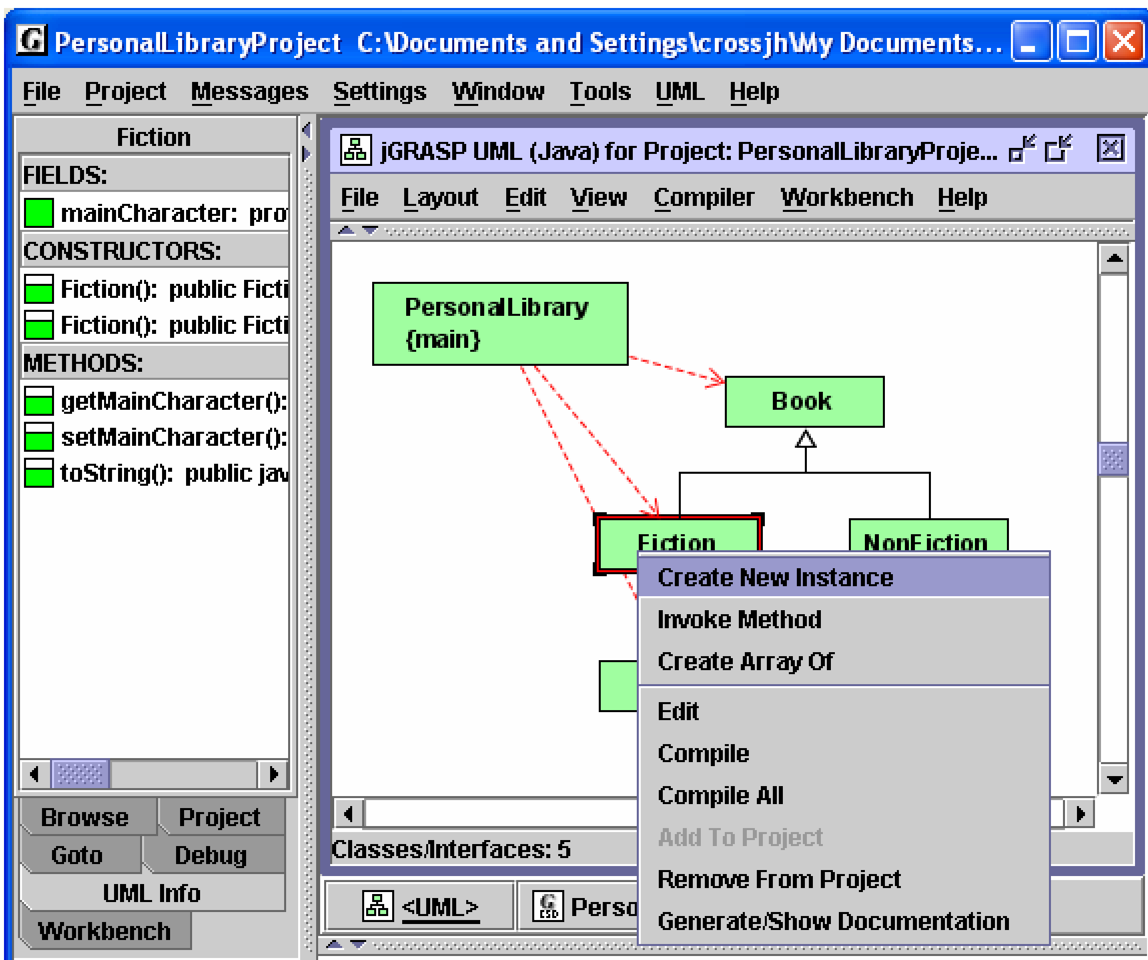


Figure 63. Creating an Object for the Workbench

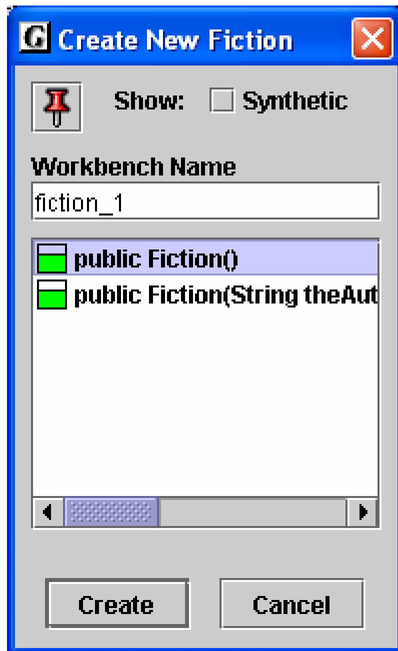


Figure 64. Selecting a constructor

Figure 65. Constructor with parameters

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 remain open. This is handy for creating multiple instances of the same class.

In Figure 66, the Workbench tab is shown after two instances of Fiction and one of Novel have been created. The first object, fiction_1, has been expanded so that the fields (mainCharacter, author, title, and pages) can be viewed. Since the first three fields are instances of the String class, they to can also be expanded. You should also note that mainCharacter is color coded green since it is the only field declared locally in Fiction. The other fields are color coded orange to indicate they were inherited from a parent, which in this case was Book.

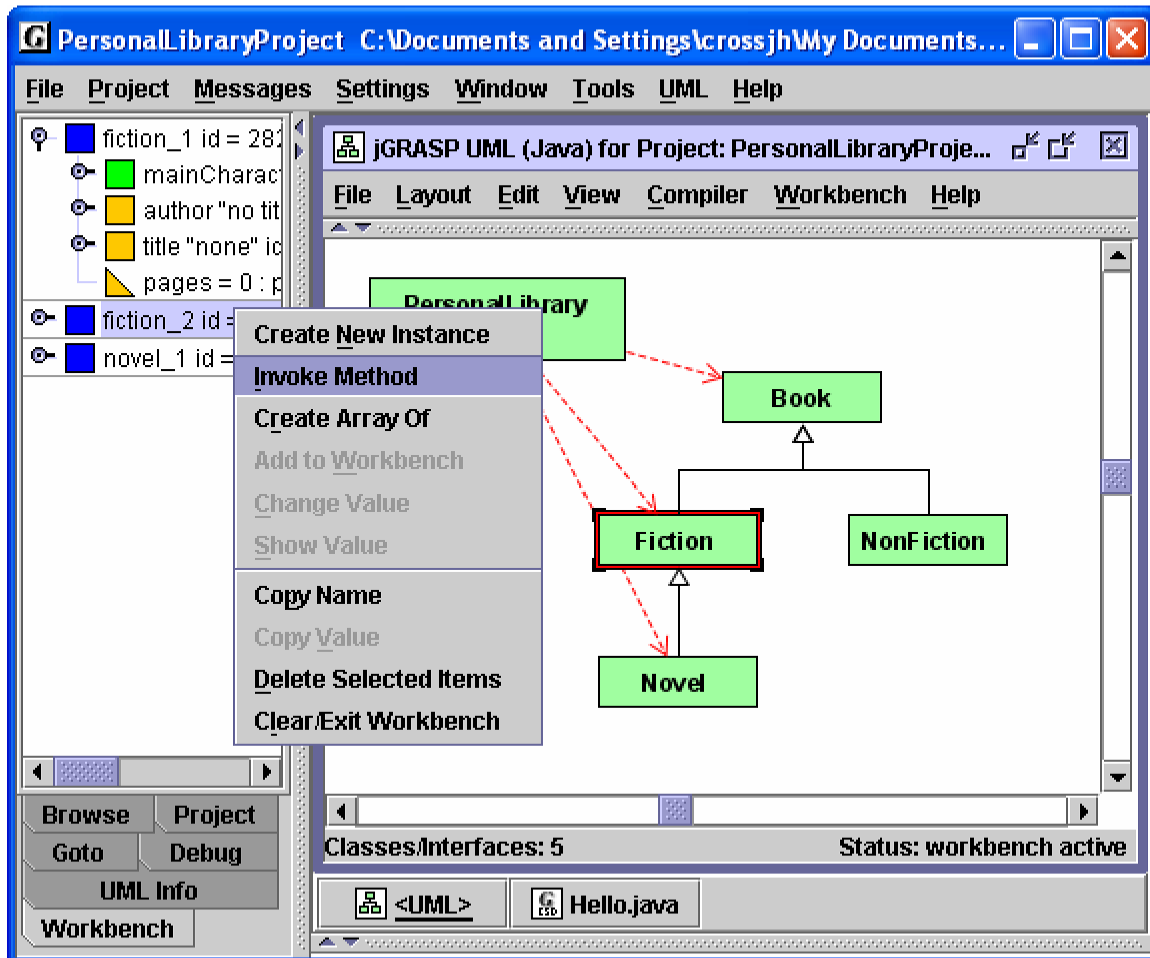


Figure 66. Workbench with two instances of Fiction

6.3 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 66, fiction_2 has been selected, followed by a right mouse click, and then Invoke Method has been selected. A list of local methods will be displayed in a dialog box as shown in Figure 67. You may also display inherited methods by selecting the appropriate parent. After one of the methods is selected and the parameters filled in as necessary, then click **Invoke**. This will execute the method and display the return value (or void) as well as output, if any, in the

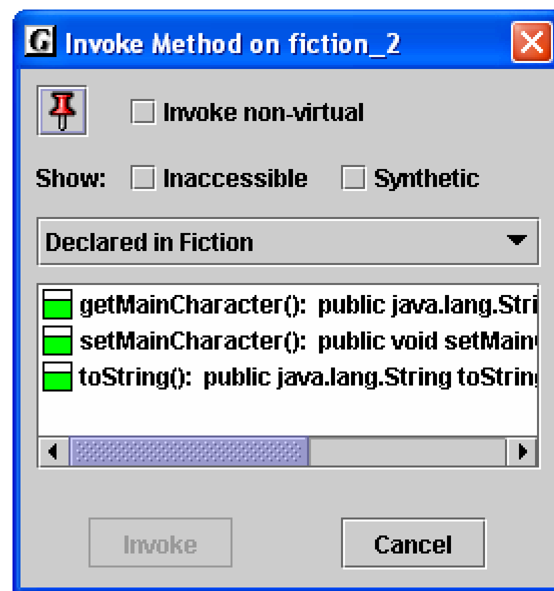


Figure 67. Selecting a method

usual way. If the method was to update a field (e.g., setMainCharacter()), the effect of the invocation would be 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 for invoking multiple methods for the same object.

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 toStirng(). By carefully reviewing the results of the method invocations, we are essentially testing our program.

6.4 Invoking Methods with 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 of that are primitive types and/or strings, these can be entered directly, with any strings enclosed in double quotes. If a parameter requires an object, then you must create an object instance for the workbench first. Then you can simple drag the object (actually a copy) from the workbench to the parameter field in the Invoke Method dialog.

6.5 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 66, fiction_1 has been expanded to show its fields (mainCharacter, author, title, and pages). Since the field mainCharacter is itself object of the class String, you can invoke any of the String methods. For example, right-click on mainCharacter, select **Invoke Method**. When the dialog pops up (Figure 68), 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 69). 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.

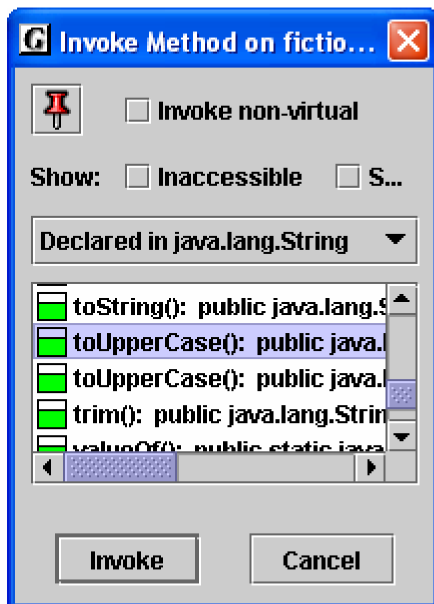


Figure 68. Invoking a String method

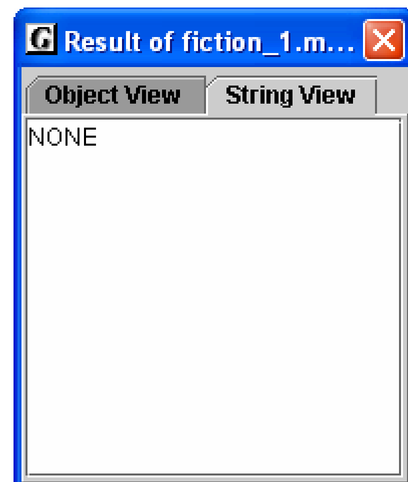


Figure 69. Result of fiction_1.mainCharacter. toUpperCase()

6.6 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 66) and select **Invoke Method**, the dialog in Figure 70 pops up. However, only the toString() method is listed because it is the only one declared in Novel. To view inherited methods, find the pull-down menu located above the list. Notice it is currently set to "Declared in Novel". Right-clicking on the menu reveals all of the superclasses of Novel (Figure 71). Selecting "Declared in superclass Fiction" lists all methods inherited from Fiction (Figure 72). Notice the orange color coding indicating "inherited" similar to the fields on the workbench. However, in this case, toString() is gray to indicate it has been overridden by the toString() method declared in Novel.

Figure 70. Invoking a method for novel_1

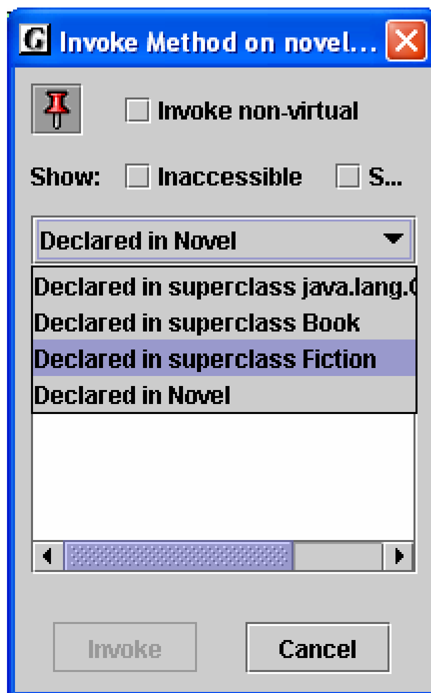


Figure 71. Viewing superclasses for novel_1

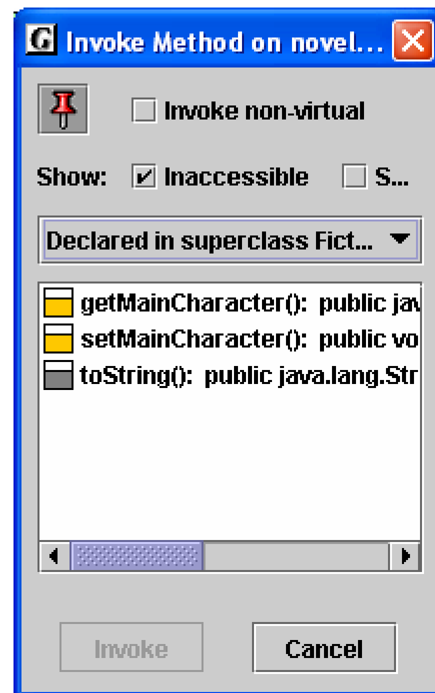


Figure 72. Viewing superclasses for novel_1

6.7 Running the Debugger on Invoked Methods

When objects are on the workbench, the workbench is running Java in debug mode. 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. At this time, you can single step through the program, examine fields, resume, etc. in the usual way. See the Tutorial on "Debugging" for more details.

6.8 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 is it OK to end the process (Figure 73). When you try to exit jGRASP, you will get a similar message (Figure 74). 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**.

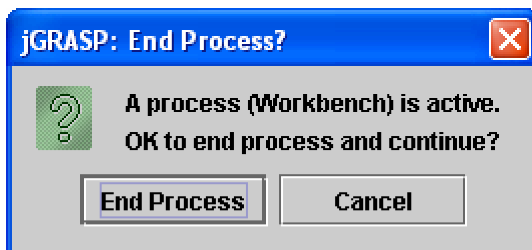


Figure 73. Making sure it is okay to exit

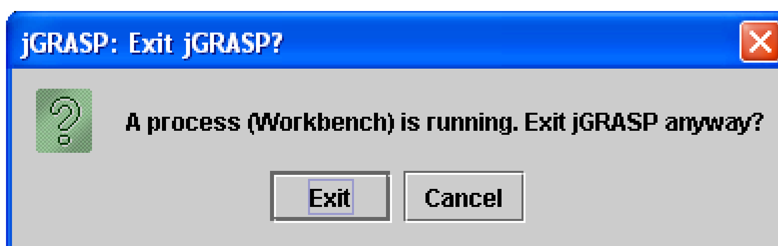


Figure 74. Making sure it is okay to exit

7 The Integrated Debugger

Your skill set for writing programs would not be complete without knowing how to use a debugger. While the connotation of a debugger is that its purpose is to assist in 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 Desktop and which 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.

7.1 Preparing to Run the Debugger

In preparation to use the debugger, click **Compiler** on the menu of the CSD Window to be sure **Debug Mode** is checked. If the box in front of Debug Mode is not checked, click on the box. When you click on Compiler 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. Then right-click to display a set of options that includes **Toggle Breakpoint**. For example, in Figure 48 the cursor is on 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.

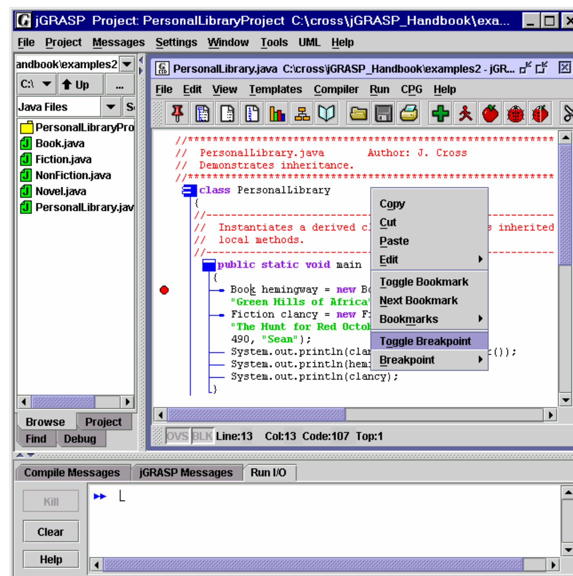


Figure 75. Setting a breakpoint

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. Then right-click to display a set of options

that includes **Toggle Breakpoint**. For example, in Figure 75 the cursor is on 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.

7.3 Running a Program in Debug Mode

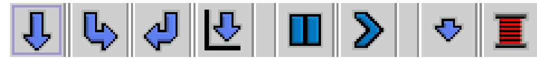
After compiler your program in Debug Mode and setting one or more breakpoints, you are ready to run your program with the debugger. You can start the debugger in one of two ways:

(1) click **Run** – **Debug** on the CSD Window menu, or

(2) click the debug symbol 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 is stopped at the breakpoint, and finally, on the left side of the jGRASP desktop the debugger pane is popped to the top. Each of these can be seen in Figure 50. Notice the debugger pane is further divided into three subpanes labeled **Threads**, **Call Stack**, and **Variables/Settings**. Each of the debugger subpanes can be resized by selecting and dragging one of the horizontal or vertical borders. This has been done in some the figures that follow. The **Threads** subpane lists all of the active threads running in the program. In the example, the red thread icon indicates the program is stopped in *main*, and green indicates a thread is running. Beginners and intermediate users can ignore the thread pane. However, advanced users should find it quite useful for starting and stopping individual threads in their programs. The **Call Stack** subpane is useful to all levels of users since is 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/Settings** subpane shows the details of the current state of the program. Finally, when a line of source code is highlighted, it means that the line is about to be executed.

7.4 Stepping Through a Program



After the program stops at the breakpoint (Figure 76), the you can use the icons at the top of the debug pane to *single step*, *step into* a method call, *step out* of a method, *run to the cursor*, *pause* the current thread, *resume*, and *suspend* new thread, while watching the call stack and contents of variables change dynamically. 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

Figure 76. Desktop after debugger is started

pass, we will "step" through the program without "stepping into" any of the method calls,

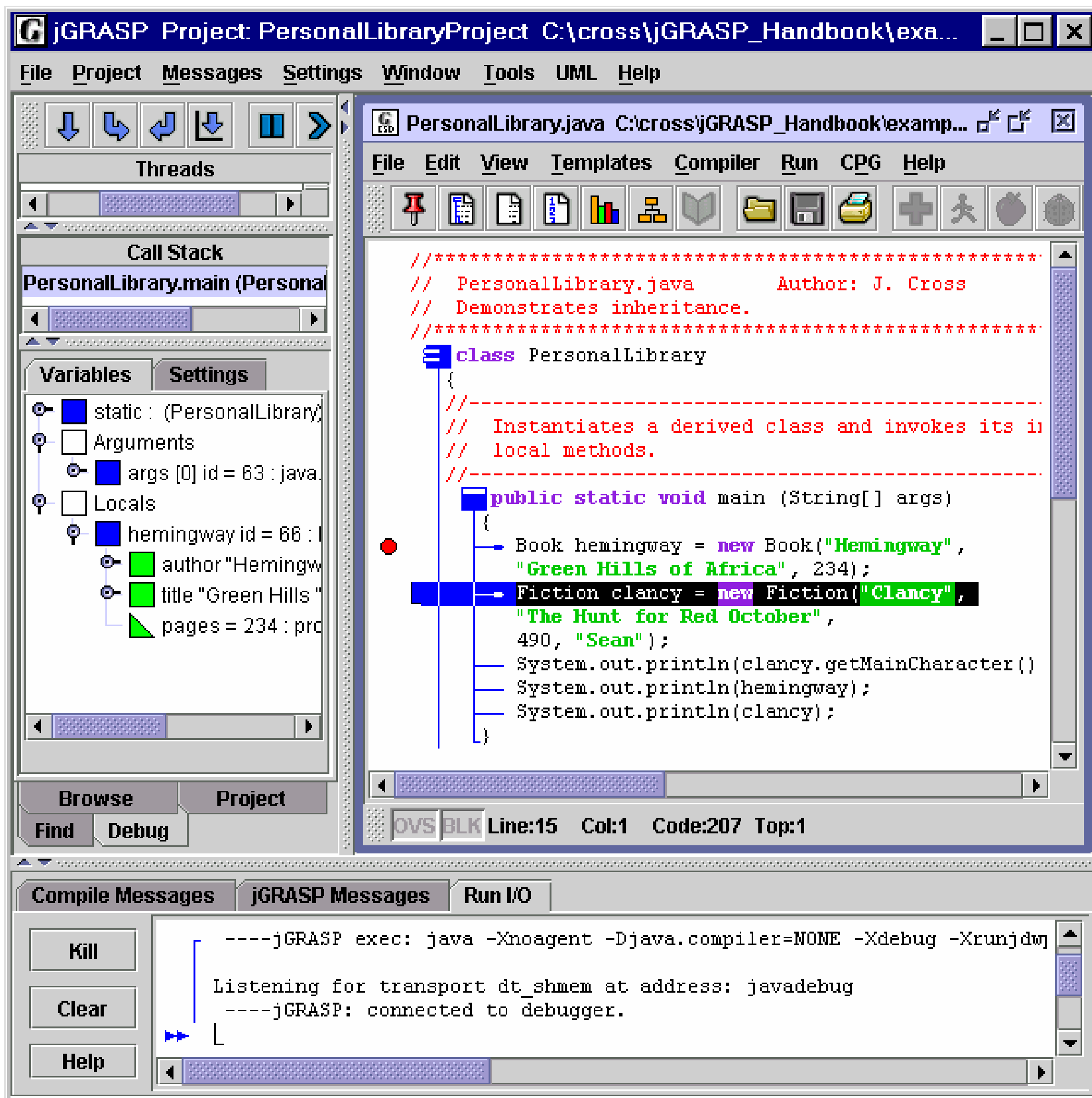


Figure 77. Desktop after hemingway (Book) object is created

and we will concentrate on the Variable section. Initially, **Variables/Settings** pane indicates no local variables have been declared. Figure 77 shows the results of "stepping" to the next statement. Notice that under Locals in the **Variable/Settiings** pane, 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 77, hemingway has been opened to show the author, title, and pages fields. Each of the String instances (e.g., author) can be opened to view the individual characters. Notice that all the fields in hemingway are green, which indicates they were declared in the class Book.

When an array is opened in the debugger, only the first ten elements (indexed 0 to 9) are displayed. To see other elements, left-click the array to select it, then click one more time. Note, this is not a double-click, but rather two single clicks. The first time you do this there may be a short delay, but a slider bar will popup that allows you to display a range of any ten items.

After executing the next statement, Figure 78 shows an instance of the Fiction class called clancy that has been created. In the figure, clancy has been opened to reveal its fields. The field “mainCharacter” is green, indicating it is defined in Fiction. The other fields (author, title, and pages) are amber, which indicates these fields were inherited from Book.

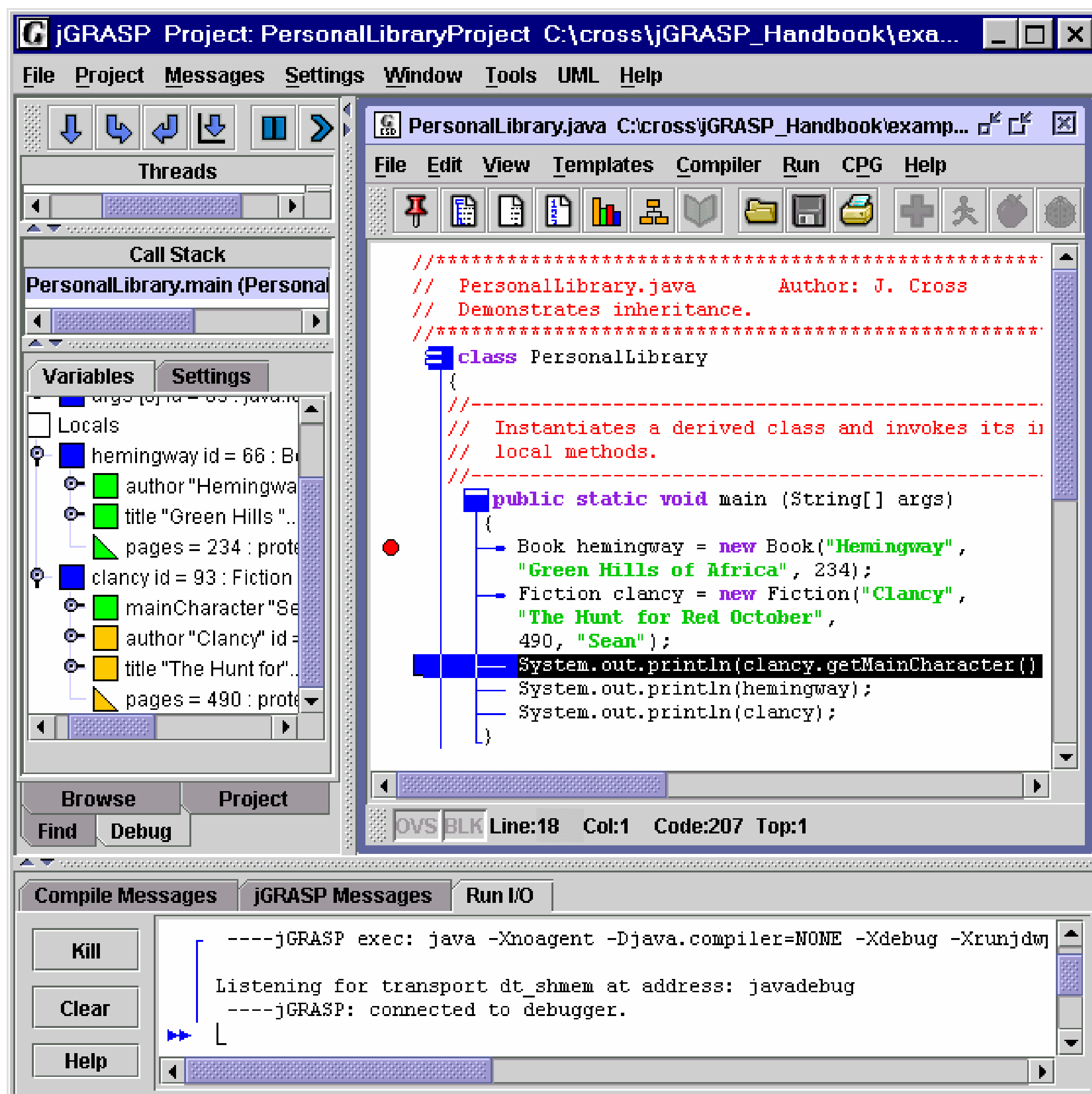


Figure 78. After next step and "clancy" created

As you continue to step though 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 pane and its subpanes should become blank, indicating that the program is no longer running.

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 by clicking **Run – Debug** on the CSD Window menu or by clicking the debug symbol on the toolbar. After arriving at the breakpoint, we "step in" and the constructor for class Book pops up in the CSD Window (Figure 79). You can then step through this method in

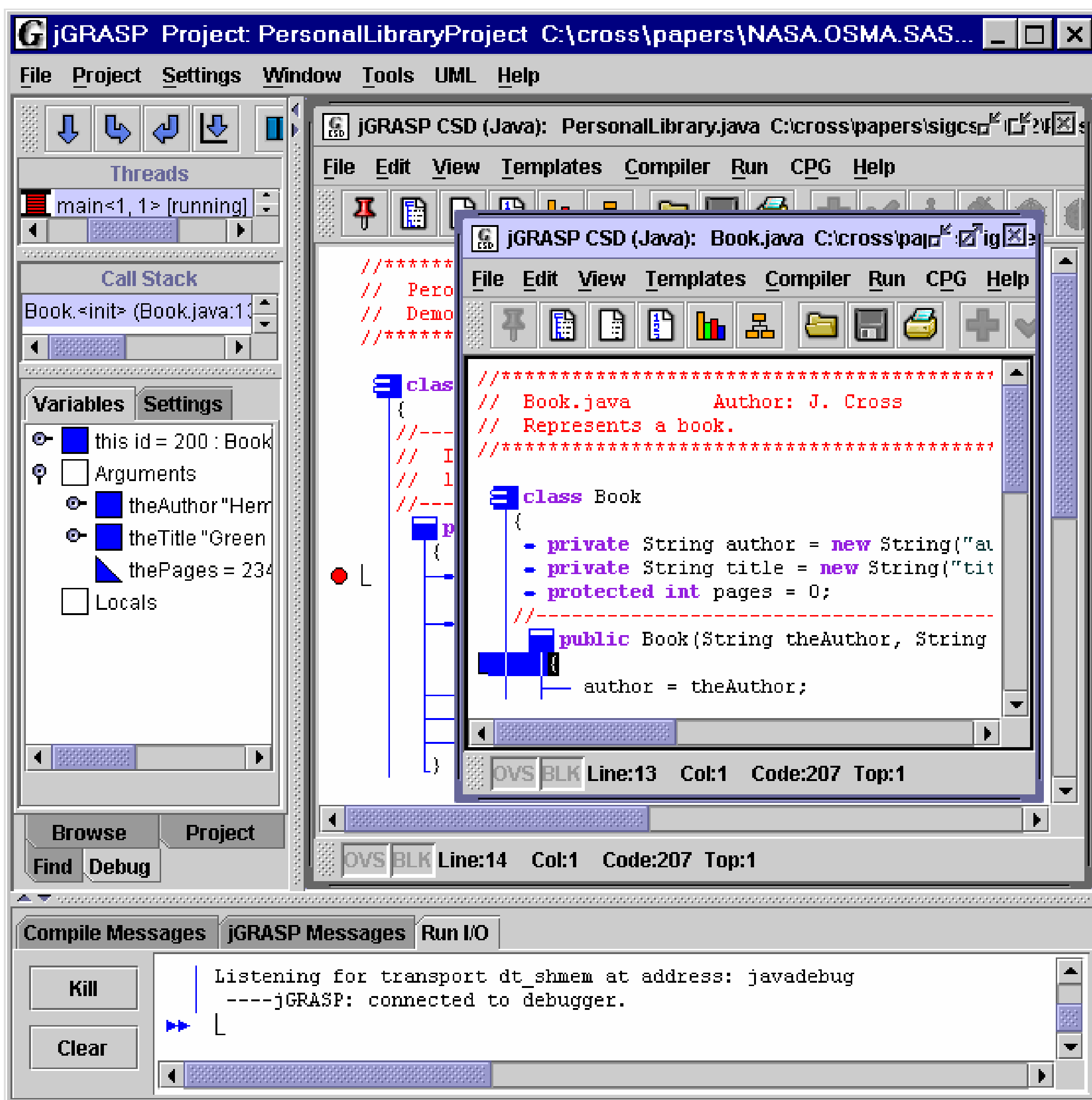


Figure 79. After next stepping into the Book constructor

the usual way, eventually returning to the statement in the main program that called the constructor.

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.

7.5 Debugging a Program

You have, no doubt, noticed that the previous discussion was only indirectly related to the activity of actually 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 must ensure that the variables 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.

For additional details, see Integrated Java Debugger in the ***jGRASP Handbook***, ***Part 2 – Reference,*** or in jGRASP **Help**.

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.

8.1 An Example to Illustrate the CSD

Figure 80 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.

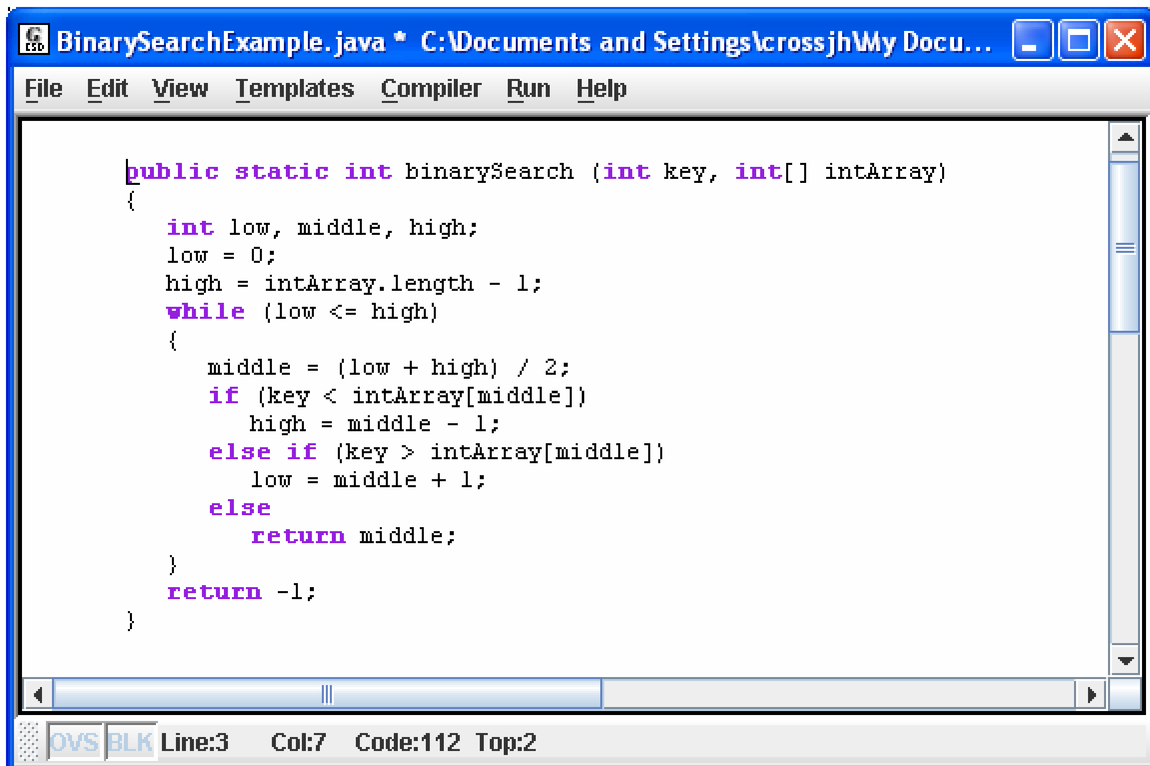


Figure 80. binarySearch method without CSD

Figure 81 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 icon 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.

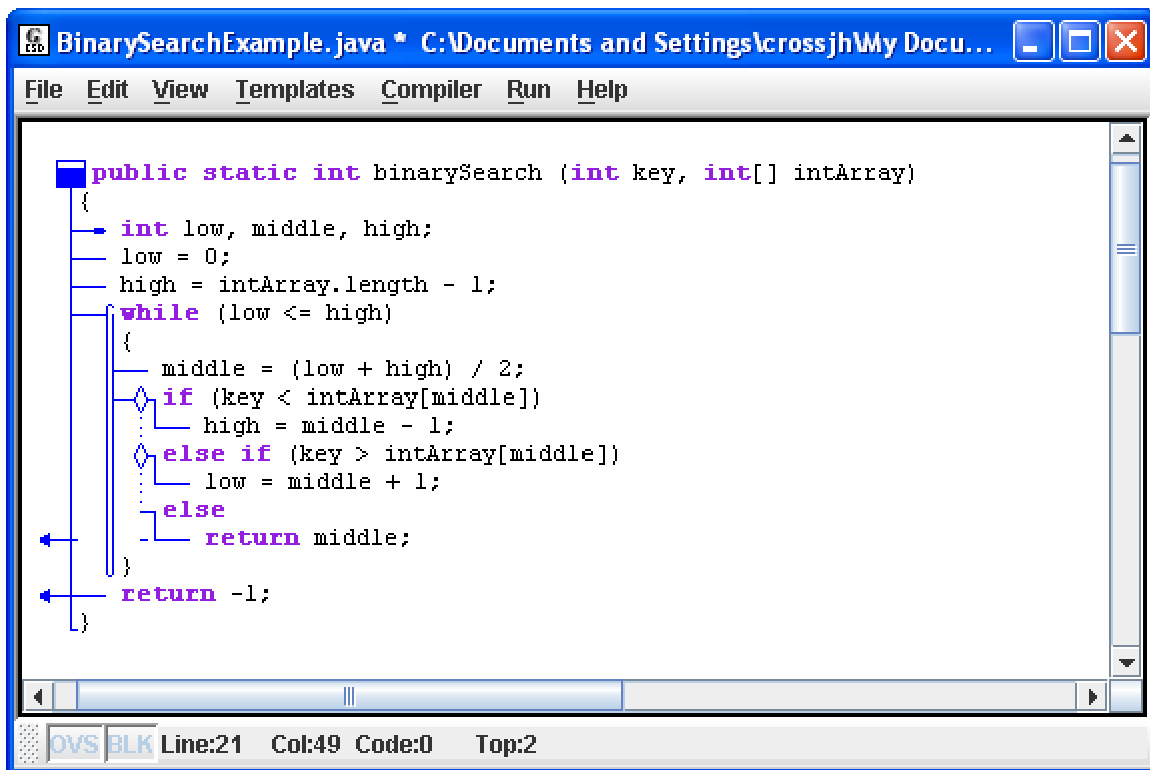


Figure 81. binarySearch with CSD

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, Nasi-Schneiderman chart, Warnier-Orr diagram, and the action diagram [Cross 1998]. In a 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.

<table>
<tr><td>Sequence</td><td><pre>
   ;
   ;
   ;
</pre></td><td>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.</td></tr>
<tr><td>Selection
if</td><td><pre>
if (cond)
   ;
</pre></td><td>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.</td></tr>
<tr><td>if..else</td><td><pre>
if (cond)
   ;
else
   ;
</pre></td><td>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.</td></tr>
<tr><td>if..else..if</td><td><pre>
if (cond)
   ;
else if (cond)
   ;
else
   ;
</pre></td><td>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.</td></tr>
</table>

<table>
<tr>
<td>Selection
(cont'd)
switch</td>
<td>

```
switch(item)
{
   case a:
      ;
      break;
   case b:
      ;
      break;
   default:
      ;
}
```

</td>
<td>The semantics of the switch statement are different from those of if statements. The expr (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 expr 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.</td>
</tr>
<tr>
<td>switch
(when break is omitted)</td>
<td>

```
switch (expr)
{
   case 1:
      ;
      break;
   case 2:
      ;
      ;
   case 3:
      ;
      ;
   case 4:
      ;
      ;
}
```

</td>
<td>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).

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.</td>
</tr>
</table>

Iteration		

while loop
(pre-test)
	`while(cond)`
	`{`
	`    ;`
	`}`
		The CSD notation for the ***while*** statement is a loop construct represented by the double line, which is continuous except for the small gap on the line with the ***while***. 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 ***while*** will be executed zero or more times.

for loop
(discrete)
	`for(i=0;i<j;i++)`
	`{`
	`    ;`
	`}`
		The ***for*** statement is represented in a similar way. The ***for*** 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 ***for*** *index* is initialized to 0, the *condition* is I < j, and the *index increment* is i++. The sequence within the ***if*** will be executed zero or more times.

do loop
(post-test)
	`do`
	`{`
	`    ;`
	`}`
	`while(cond);`
		The ***do*** 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.

break in loop	`while (cond)`
	`{`
	`    ;`
	`    if (cond)`
	`        break;`
	`    ;`
	`}`
		The ***break*** statement can be used to transfer control flow out of any loop (***while***, ***for***, ***do***) 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 ***if*** statement. If the ***break*** is used alone (e.g., without the ***if*** statement), the statements in the loop body beyond the ***break*** will never by executed.

Iteration
(cont'd)
continue
	`do`
	`{`
	`    ;`
	`    if (cond)`
	`        continue;`
	`    ;`
	`}`
	`while (cond);`
		The ***continue*** 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 (***while***, ***for***, ***do***).

Exception Handling	`try`
	`{`
	`    ;`
	`}`
	`     catch(E)`
	`    {`
	`        ;`
	`    }`
	`finally`
	`{`
	`    ;`
	`}`
		In Java, the control construct for exception handling is the ***try..catch*** statement with optional ***finally*** clause. In the example at left, if stmt1 generates an exception E, then control is transferred to the corresponding ***catch*** clause. After the catch body is executed, the ***finally*** clause (if present) is executed. If no exception occurs in the try block, when it completes, the ***finally*** clause (if present) is executed.
		
		The ***try..catch*** statement can have multiple ***catch*** clauses, one for each exception to be handled.

With a return	`try`
	`{`
	`    ;`
	`    ;`
	`    return;`
	`}`
	`     catch(E)`
	`    {`
	`        ;`
	`    }`
	`finally`
	`{`
	`    ;`
	`}`
		By definition, the ***finally*** clause is always executed not matter how the ***try*** block is exited. In the example at left, a ***return*** 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 ***try*** block. The CSD uses this same convention for ***break*** and ***continue*** when these cause a ***try*** block to exited.
		
		When try blocks are nested and ***break***, ***continue***, and ***return*** 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.

8.4 CSD Templates

In Figure 82, 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.

ConstructsOfCSD2.java C:\Documents and Settings\crossjh\My Documents...

File Edit View Templates Compiler Run Help

```
//  Sequence: default
   ;
   ;
   ;

//  Selection: if
if (cond)
   ;

//  Selection: if..else
if (cond)
   ;
else
   ;

//  Selection: if..else..if
if (cond)
   ;
else if (cond)
   ;
else
   ;

//  Selection: switch
switch(item)
{
   case a:
      ;
      break;
   case b:
      ;
      break;
   default:
      ;
}
```

```
//  Iteration: while
while(cond)
{
   ;
}

//  Iteration: for
for(index=0;index<j;index++)
{
   ;
}

//  Iteration: do
do
{
   ;
}
while(cond);

//  Exception Handling
//   try..catch..finally
try
{
   ;
}
catch(ETYPE EXCEPTN)
{
   ;
}
finally
{
   ;
}
```

OVS BLK Line:83 Col:31 Code:0 Top:9 TopB:63

Figure 82. 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 that 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 it 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 ***class***. Note, the file should be saved with the name of the class, which in this case is MyClass.	`public class MyClass`
	`{`
	`}`
		`public class MyClass`
		`{`
		`}`

2. Now, inside the class, we enter the text for a ***method*** called myMethod, and then re-generate the CSD by pressing **F2**.	`public class MyClass`
	`{`
	    `myMethod()`
	    `{`
	    `}`
	`}`
		`public class Hello`
		`{`
		    `myMethod()`
		    `{`
		    `}`
		`}`

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

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.

<table>
<tr>
<td>Reading Sequence
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.</td>
<td><pre>s(1);
s(2);
s(3);</pre></td>
</tr>
<tr>
<td>Reading Selection
Now combining the sequence with selection (if.. else), after s(1), we enter the if statement marked by the diamond. If the condition is true, 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 false, 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).</td>
<td><pre>s(1);
if (cond)
   s(2);
else
   s(3);
s(4);</pre></td>
</tr>
</table>

Reading Selection with Nesting	

As above, after s(1), we enter the ***if*** 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 *false*, we read down the dotted line (the false path) to s(4). If cond2 is false, we read down the dotted line to the ***else*** 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).	`s(1);`
	`if (cond1)`
	`   if (cond2)`
	`      s(2);`
	`   else`
	`      s(3);`
	`s(4);`

Reading Selection with Even Deeper Nesting

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).

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.

If s(4) is executed, we can see immediately that s(7) follows.

In fact, from any statement, regardless of the level of nesting, the CSD makes it easy to see which statement is executed next.

```
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 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

Consider the fragment at right with s(1) and s(2) in the body of the ***if*** statement.

```
s(1);
if (cond)
   s(2);
   s(3);
```

After the CSD is generated, the reader can see how the compiler will interpret the code, and add the missing braces.

```
s(1);
if (cond)
   s(2);
s(3);
```

<table>
<tr><td>Here is another common mistake made glaring by the CSD.

Most likely, the semi-colon after the condition was unintended. However, the CSD shows what there rather than what was intended.</td><td><pre>if (cond);
   s(2);
   s(3);

if (cond);
s(2);
s(3);</pre></td></tr>
<tr><td>Similarly, the CSD provides the correct interpretation of the while statement.

Missing braces . . .
.</td><td><pre>while (cond)
   s(2);
   s(3);

while (cond)
   s(2);
s(3);</pre></td></tr>
<tr><td>Similarly, the CSD provides the correct interpretation of the while statement.

Unintended semi-colon . . .</td><td><pre>while (cond);
   s(2);
   s(3);

while (cond);
s(2);
s(3);</pre></td></tr>
</table>

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

```java
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");
      }
   }

};
```


```java
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.

8.7 References

[Cross 1998] J. H. Cross, S. Maghsoodloo, and T. D. Hendrix, "Control Structure Diagrams: Overview and Initial Evaluation," J*ournal 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.