jGRASP

Tutorials

Tutorials for the jGRASP™ 1.8.7 Integrated Development Environment

James H. Cross II and Larry A. Barowski

Auburn University

September 2, 2009

The **Tutorials** have been updated with each new major release of jGRASP since their original inclusion in the **jGRASP Handbook** (Copyright © 2003 Auburn University).

James H. Cross II is a professor of computer science and software engineering at Auburn University.

Larry A. Barowski is a research associate in computer science and software engineering at Auburn University.

Printing Considerations

The Tutorials are formatted for 5.5 in. x 8.5 in. pages with a base font of 10 point Times New Roman. This improves on-screen viewing and facilitates *booklet printing* (four pages per sheet of 8.5 in. x 11 in. paper when printing on both sides). On Adobe Acrobat's Print dialog, select "Booklet Printing" under Page Scaling. The default Booklet Subset will be Both sides.

The Tutorials may also be printed "two pages per sheet" on 8.5 in. x 11 in. paper by selecting "Multiple pages per sheet" under "Page Scaling" on Adobe's Print dialog. This may yield a slightly smaller overall font than booklet printing.

If "Page Scaling" is set to "None" (the default) or "Shrink to printable area", the Tutorials will be printed "one page per sheet" with the font slightly larger than booklet printing. If "Page Scaling" is set to "Fit printable area", the Tutorials may be printed "one page per sheet" with the font even larger.

Regardless of the pages per sheet or page scaling you select, it is strongly recommended that you print a few test pages before printing a large number of pages. The options on your particular printer may be different from the ones described above.

Table of Contents

Overview of jGRASP and the Tutorials...................... 1

1 Installing jGRASP...1-1

1.1 The Install Files..1-2

1.2 Installing on Windows 95/98/2000/XP/Vista..1-2

1.3 Installing on Mac OS X 1-8

1.4 Installing on Other Systems (including x86 Linux, SPARC Solaris, and NetBSD/i386) 1-12

1.5 Compilers.. 1-12

1.6 Setting PATH and CLASSPATH 1-14

1.7 jGRASP Start Up Settings 1-15

1.8 Plug-Ins for jGRASP 1-16

2 Getting Started 2-1

2.1 Starting jGRASP...2-2

2.2 Quick Start - Opening a Program, Compiling, and Running.................2-3

2.3 Creating a New File 2-5

2.4 Saving a File 2-8

2.5 Building Java Programs - - Recap...2-9

2.6 Interactions (Java only).. 2-10

2.7 Generating a Control Structure Diagram... 2-12

2.8 Folding a CSD... 2-14

2.9 Line Numbers.. 2-15

2.10 Compiling a Program – A Few More Details.................................... 2-15

2.11 Running a Program - Additional Options.. 2-18

2.12 Using the Debugger (Java only).. 2-19

2.13 Opening a File – Additional Options 2-22

2.14 Closing a File 2-24

2.15 Exiting jGRASP... 2-25

2.16 Review and Preview of What’s Ahead 2-25

2.17 Exercises 2-27

3 Getting Started with Objects.................................3-1

3.1 Starting jGRASP...3-2

3.2 Navigating to Our First Example Project..3-3

3.3 Opening a Project and UML Window...3-4

3.4 Compiling and Running the Program from UML Window 3-5
3.5 Exploring the UML Window 3-6
3.6 Viewing the Source Code in the CSD Window 3-7
3.7 Exploring the Features of the UML and CSD Windows 3-8
3.7.1 Viewing the source code for a class 3-8
3.7.2 Displaying class information 3-8
3.7.3 Displaying Dependency Information 3-8
3.8 Generating Documentation for the Project 3-9
3.9 Using the Object Workbench 3-10
3.10 Opening a Viewer Window 3-13
3.11 Invoking a Method 3-14
3.12 Invoking Methods with Parameters That Are Objects 3-15
3.13 Invoking Methods on Object Fields 3-16
3.14 Showing Categories of Methods 3-17
3.15 Creating Objects from the CSD Window 3-18
3.16 Using Interactions 3-19
3.17 Running the Debugger on Invoked Methods 3-21
3.18 Creating an Instance from the Java Class Libraries 3-21
3.19 Exiting the Workbench 3-21
3.20 Closing a Project 3-22
3.21 Exiting jGRASP 3-22
3.22 Review of Toolbar Buttons 3-23
3.23 Exercises 3-24

4 Interactions 4-1

4.1 Starting Interactions 4-2
4.2 Interactions with Primitives 4-3
4.3 Interactions with Reference Types 4-8
4.4 Interactions with Your Own Classes 4-10
4.5 Working with Reference Types – Important Details 4-10
4.6 Interactions with the Debugger 4-10

5 The Control Structure Diagram (CSD) 5-1

5.1 An Example to Illustrate the CSD 5-2
5.2 CSD Program Components/Units 5-4
5.3 CSD Control Constructs 5-5
5.4 CSD Templates 5-10
5.5 Hints on Working with the CSD 5-11
5.6 Reading Source Code with the CSD 5-12
5.7 References 5-18

6 The Integrated Debugger..6-1

6.1 Preparing to Run the Debugger..6-2
6.2 Setting a Breakpoint..6-3
6.3 Running a Program in Debug Mode..6-3
6.4 Stepping Through a Program – the Debug Buttons..6-5
6.5 Stepping Through a Program – without *Stepping In*..6-7
6.6 Stepping Through a Program – and *Stepping In*..6-9
6.7 Opening Object Viewers..6-11
6.8 Debugging a Program..6-13

7 Projects..7-1

7.1 Creating a Project..7-2
7.2 Adding files to the Project..7-4
7.3 Removing files from the Project..7-5
7.4 Generating Documentation for the Project (Java only)..7-6
7.5 Jar File Creation and Extraction..7-8
7.6 Closing a Project..7-8
7.7 Exercises..7-9

8 UML Class Diagrams..8-1

8.1 Opening the Project..8-2
8.2 Generating the UML..8-2
8.3 Compiling and Running from the UML Window..8-4
8.4 Determining the Contents of the UML Class Diagram..8-5
8.5 Laying Out the UML Class Diagram..8-8
8.6 Displaying the Members of a Class..8-9
8.7 Displaying Dependencies Between Two Classes..8-10
8.8 Navigating to Source Code via the Info Tab..8-11
8.9 Finding a Class in the UML Diagram..8-11
8.10 Opening Source Code from UML..8-11
8.11 Saving the UML Layout..8-12
8.12 Printing the UML Diagram..8-12

9 The Workbench..9-1

9.1 Invoking Static Methods from the CSD Window..9-2
9.2 Invoking Static Methods from the UML Window..9-4
9.3 Creating an Object for the Workbench..9-7
9.4 Invoking a Method..9-9
9.5 Invoking Methods with Parameters Which Are Objects..9-10
9.6 Invoking Methods on Object Fields..9-10

9.7 Selecting Categories of Methods to Invoke 9-12
9.8 Opening Object Viewers 9-15
9.9 Running the Debugger on Invoked Methods 9-17
9.10 Exiting the Workbench 9-17

10 Viewers for Data Structures 10-1

10.1 Introduction 10-2
10.2 Opening Viewers 10-2
10.3 Setting the View Options 10-5
10.4 Selecting Among Views 10-7
10.5 Presentation Views for LinkedList, HashMap, and TreeMap 10-9
10.6 Presentation Views for Code Understanding 10-14
10.6.1 LinkedListExample.java 10-15
10.6.2 BinaryTreeExample.java 10-19
10.6.3 Configuring Views generated by the *Structure Identifier* 10-22
10.7 Using the Viewers from the Workbench 10-26
10.8 Summary of Views 10-33
10.9 Exercises 10-35

Overview of jGRASP and **the Tutorials**

jGRASP is a lightweight integrated development environment (IDE), created specifically to provide visualizations for improving the comprehensibility of software. jGRASP is implemented in Java, and thus, runs on all platforms with a Java Virtual Machine. jGRASP supports Java, C, C++, Objective-C, Ada, and VHDL, and it comes configured to work with several popular compilers to provide "point and click" compile and run functions. jGRASP 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 three important software visualizations: (1) *Control Structure Diagrams* (Java, C, C++, Objective-C, Ada, and VHDL) for source code visualization, (2) *UML Class Diagrams* (Java) for architectural visualization, and (3) *Dynamic Viewers* (Java) which provide runtime views for primitives and objects including traditional data structures such as linked lists and binary trees. jGRASP also provides an innovative *Object Workbench*, *Debugger*, and *Interactions* 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++, Objective-C, Java and VHDL. The CSD is intended to improve the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit. The CSD, designed to fit into the space that is normally taken by indentation in source code, is an alternative to flow charts and other graphical representations of algorithms. The CSD is a natural extension to architectural diagrams such as UML class diagrams.

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

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

The **Dynamic Viewers** for objects and primitives provide visualizations as the user steps through a program in debug mode or invokes methods for an object on the workbench. Textbook-like *Presentation* views are available for instances of classes that represent traditional data structures. When a viewer is opened, a *structure identifier* attempts to automatically recognize linked lists, binary trees, hash tables, and array wrappers (lists, stacks, queues, etc.) during debugging or workbench use. When a positive identification is made, an appropriate *presentation* view of the object is displayed. The *structure identifier* is intended to work for user classes, including textbook examples, as well as the most commonly used classes in the Java Collections Framework (e.g., ArrayList, LinkedList, HashMap, and TreeMap). A future *Viewer API* will allow users to create custom dynamic views of their own classes.

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

The **Integrated Debugger** works in conjunction with the CSD window, UML window, Object Workbench, and Interactions. The Debugger provides a seamless way for users to examine their programs step by step. The execution threads, call stack, and local 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 **Interactions** (new in jGRASP 1.8.7) feature allows users to enter most Java statements and expressions and then execute or evaluate them immediately. Interactions can be especially helpful when learning and experimenting with new elements in the Java language.

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

1 Installing jGRASP – Most users will skip this tutorial. However, it does provide details on the installation process as well as instructions for changing default startup settings for jGRASP. This tutorial also describes how to set the system path and the Java classpath from within jGRASP.

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 introduces interactions, the control structure diagram, and the debugger.

3 Getting Started with Objects – This tutorial is a good starting place for those interested in an *Objects First* approach to learning Java, but it assumes the reader will refer to the previous tutorial as needed. Projects, UML class diagrams, the Object Workbench, and Viewers are introduced.

The topics that are introduced in *Getting Started* and *Getting Started with Objects* are covered in more depth in the following seven tutorials. In most cases, these tutorials may be read as a topic becomes relevant to a user, rather than in the order indicated by their numbers.

4 Interactions – Although the Interactions feature is introduced in *Getting Started* and *Getting Started with Objects*, this tutorial provides examples for several common scenarios, including multi-line interactions and how to copy and paste interactions.

5 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 at any time. The latter part contains some helpful hints on getting the most out of the CSD.

6 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 their values change.

7 Projects – This tutorial discusses the concept of a project file (.gpj) in jGRASP which stores all information for a specific project. This includes the names (and paths) of each file in the project, the project settings, and the layout of the UML diagram. Some users may want to work in projects from the beginning while others want to deal with projects only when programs have multiple classes or files.

8 The UML Class Diagram – The focus of this tutorial is on generating a UML class diagram for a project and then using the diagram as a basis for creating instances for the workbench. This tutorial assumes the user understands the concept of a project and is able to create one (Tutorial 4).

9 The Workbench – This tutorial assumes the user is able to create a project (Tutorial 4) and work with UML class diagrams (Tutorial 5). The workbench provides an exciting way to approach object-oriented concepts and programming by allowing the user to create objects and invoke methods directly.

10 Viewers for Data Structures – This tutorial provides a more in-depth introduction to using Viewers with linked lists, binary trees, and other traditional data structures. Examples of *presentation* views are included for instances of non-JDK implementations for a linked list and binary tree as well as for instances of ArrayList, LinkedList, HashMap, and TreeMap.

For additional information and to download jGRASP, please visit our web site (http://www.jgrasp.org).

1 Installing jGRASP

Among all of the jGRASP Tutorials, this one is expected to be the least read. Most users will download the jGRASP self-install file for their system, double-click the file, follow the instructions in the dialog, launch jGRASP, and be up and running. If you have successfully done this, you are ready to go on to the next chapter.

However, occasionally users need additional information when installing jGRASP and configuring it for their particular needs. This tutorial includes a description of the available *install* files, instructions for installing on Windows, Mac OS X, and Linux/UNIX, information on compilers, instructions for setting the system *path* and Java *classpath*, a description of the available jGRASP *startup* settings, and a list of available plug-ins for jGRASP. Most readers will need to refer to only a few of the sections below.

Since jGRASP is written in Java, you must have Java installed on your machine in order to run jGRASP. To compile and run Java programs, you will need to install the full Java 2 Platform Standard Edition (J2SE) Development Kit which is usually referred to as the **JDK.** See Section 1.5 for information on the JDK as well compilers for other languages.

Objectives – When you have completed this tutorial, you should be able to successfully install jGRASP on your computer, change the default compiler configuration, set *paths* and *classpaths*, modify the jGRASP *startup* settings, and be familiar with the available *plug-ins* for jGRASP.

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

1.1 The *Install* Files

1.2 Installing on Windows 95/98/2000/XP/Vista

1.3 Installing on Mac OS X

1.4 Installing on Other Operating Systems

1.5 Compilers

1.6 Setting PATH and CLASSPATH

1.7 jGRASP Start Up Settings

1.8 Plug-Ins for jGRASP

1.1 The Install Files

The current version of jGRASP is available from http://www.jgrasp.org in separate *install* files: one is self-extracting for **Microsoft Windows**, one is for **Mac OS X (10.4 or higher),** and the third is a generic **ZIP** file primarily intended for **Linux** and **UNIX** systems, although it can be used to install jGRASP on any system. Each of these is briefly described below. Beginning with version 1.8.6_02, jGRASP requires Java 1.5 (a.k.a. Java 5.0) or higher. If you must use an older version of Java, you will need to use jGRASP 1.8.6_01.

jGRASP exe – **Windows self-extracting exe file.** ***In order to run jGRASP and compile and run Java programs, the full JDK (rather than JRE) must be installed.***

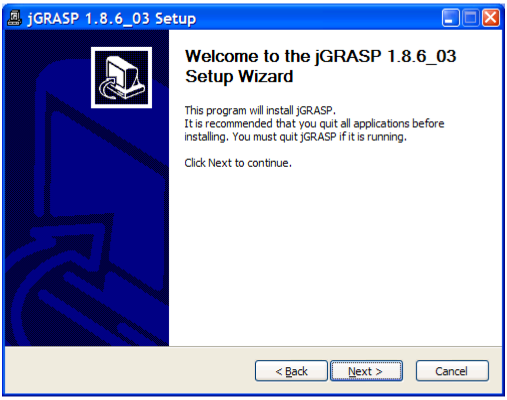
jGRASP pkg.tar.gz – **Mac OS X 10.4 or higher (PPC or Intel) tarred and gzipped package file** (requires admin access to install). J2SDK is pre-installed on Mac OS X machines, but you may want to upgrade if a newer version is available.

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

1.2 Installing on Windows 95/98/2000/XP/Vista

If you plan to compile and run Java programs on a Windows machine, the **jGRASP exe** install file is recommended. Prior to running jGRASP, you will also need to have installed the full JDK on your machine. If you are not planning to compile and run Java programs (e.g., you plan to compile and run programs written in C/C++ but not Java) then you will only need the JRE installed rather than the full JDK.

After you have downloaded the install file, simply double click on it, and you should see the jGRASP Setup dialog open as shown in Figure 1-1. Click “Next” to continue the installation process. The script will take you through the steps for installing jGRASP. If you are uncertain about a step, you should accept the default by clicking *Next* and/or pressing the ENTER key.



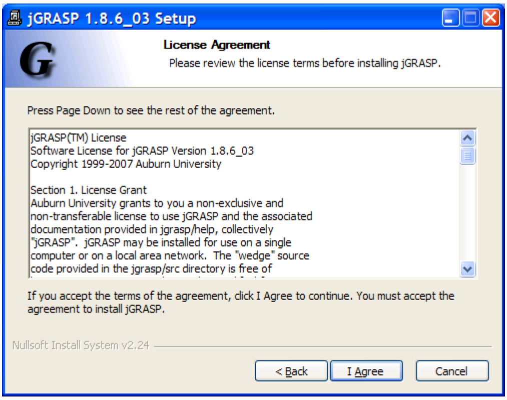
Welcome to the jGRASP 1.8.6_03 Setup Wizard

This program will install jGRASP.
It is recommended that you quit all applications before installing. You must quit jGRASP if it is running.

Click Next to continue.

Figure 1-1. jGRASP Setup dialog

The License Agreement is shown in Figure 1-2. You will need to scroll the dialog on your screen to see the entire license. After reviewing it, click "I Agree" to continue.



License Agreement

Please review the license terms before installing jGRASP.

Press Page Down to see the rest of the agreement.

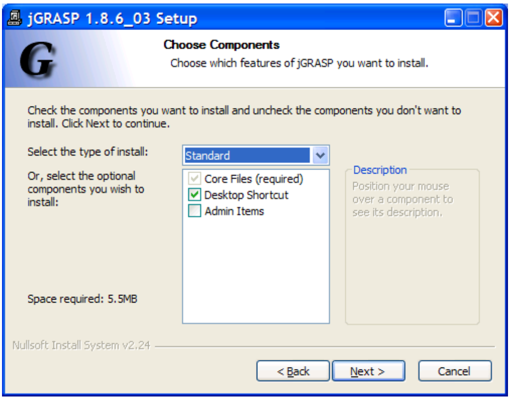
jGRASP(TM) License
Software License for jGRASP Version 1.8.6_03
Copyright 1999-2007 Auburn University

Section 1. License Grant
Auburn University grants to you a non-exclusive and
non-transferable license to use jGRASP and the associated
documentation provided in jgrasp/help, collectively
"jGRASP". jGRASP may be installed for use on a single
computer or on a local area network. The "wedge" source
code provided in the jgrasp/src directory is free of

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install jGRASP.

Figure 1-2. License Agreement

The next screen, Figure 1-3, allows you to select the components you want to install. Most users should simply install the "Standard" group of components, which is the default. However, if you need to provide common settings for all users on a network, you should select the "Administrator" group (or "Admin Items"). When these are included, you will be asked to provide an admin folder name in a later screen (Figure 1.5).



jGRASP 1.8.6_03 Setup

Choose Components

Choose which features of jGRASP you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Standard

Or, select the optional components you wish to install:

- ☑ Core Files (required)
- ☑ Desktop Shortcut
- ☐ Admin Items

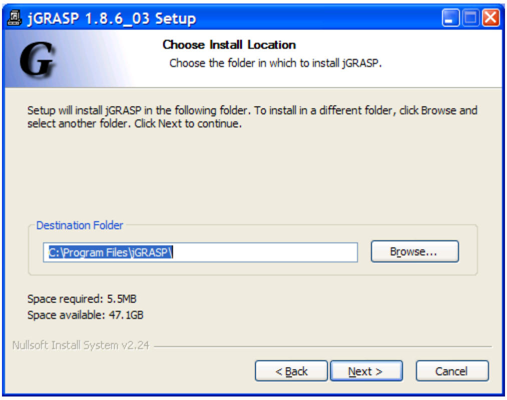
Description

Position your mouse over a component to see its description.

Space required: 5.5MB

Figure 1-3. Selecting components to install

The screen in Figure 1-4 allows you to indicate the folder where jGRASP is to be installed. The Windows default is “Program Files” so most users should just click “Next” to continue



jGRASP 1.8.6_03 Setup

Choose Install Location

Choose the folder in which to install jGRASP.

Setup will install jGRASP in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

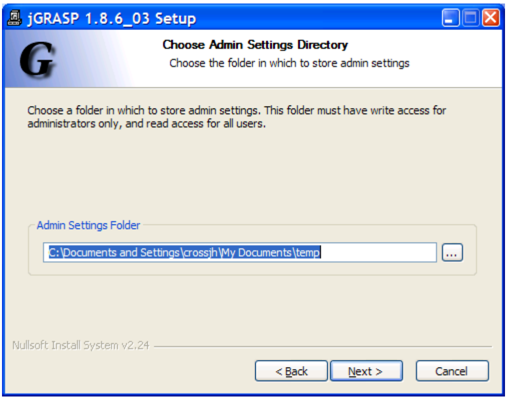
Destination Folder: C:\Program Files\jGRASP\

Space required: 5.5MB
Space available: 47.1GB

Figure 1-4. Choosing the *Install* directory

The screen in Figure 1-5 will only be displayed if you selected the “Admin Items” above in Figure 1-3. Most users should not install Admin Items. However, if you include Admin items, you need to provide a directory for the common settings for all users on the target network. Use a full path to specify the directory. This directory must be accessible and readable by all users, and writable only for the system administrators. You should select a directory location outside the jGRASP distribution, so that you can continue to use the settings after upgrading.

Administrators (i.e., anyone with write access in the common settings directory) will have additional options on the “Settings” menu when they run jGRASP. These include CSD, Compiler, and Print settings.



jGRASP 1.8.6_03 Setup

Choose Admin Settings Directory

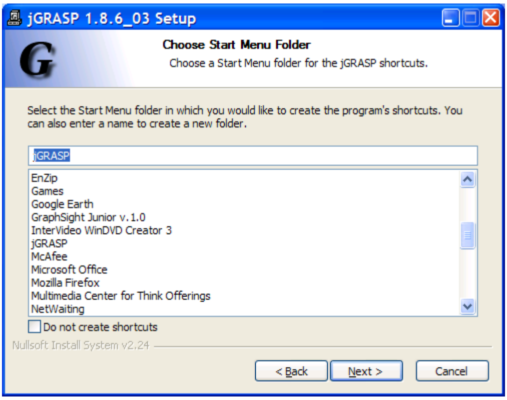
Choose the folder in which to store admin settings

Choose a folder in which to store admin settings. This folder must have write access for administrators only, and read access for all users.

Admin Settings Folder: C:\Documents and Settings\crossjh\My Documents\temp

Figure 1-5. Choosing the *Admin* directory if “Admin Items” are included in Figure 1-3

The screen in Figure 1.6 allows you to specify the Start Menu folder for the jGRASP shortcuts. Normally, this folder would be named “jGRASP” as indicated and you should click “Next” to continue.



jGRASP 1.8.6_03 Setup

Choose Start Menu Folder

Choose a Start Menu folder for the jGRASP shortcuts.

Select the Start Menu folder in which you would like to create the program's shortcuts. You can also enter a name to create a new folder.

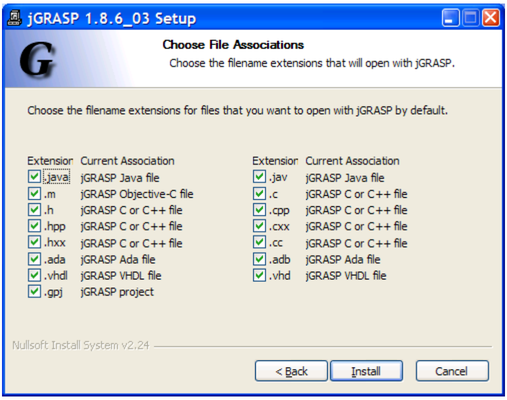
jGRASP

EnZip
Games
Google Earth
GraphSight Junior v.1.0
InterVideo WinDVD Creator 3
jGRASP
McAfee
Microsoft Office
Mozilla Firefox
Multimedia Center for Think Offerings
NetWaiting

□ Do not create shortcuts

Figure 1-6. Choosing the *Start Menu* folder

Figure 1.7 shows the file associations supported in jGRASP. By default jGRASP includes those file extensions that are not already associated with other applications. After selecting (or unselecting) as appropriate, click “Install” to begin the install process.



jGRASP 1.8.6_03 Setup

Choose File Associations

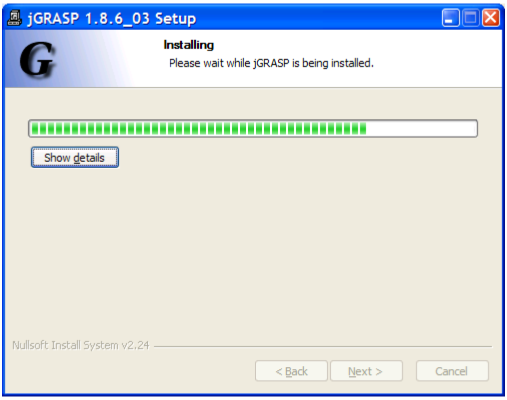
Choose the filename extensions that will open with jGRASP.

Choose the filename extensions for files that you want to open with jGRASP by default.

Extension	Current Association	Extension	Current Association
☑ .java	jGRASP Java file	☑ .jav	jGRASP Java file
☑ .m	jGRASP Objective-C file	☑ .c	jGRASP C or C++ file
☑ .h	jGRASP C or C++ file	☑ .cpp	jGRASP C or C++ file
☑ .hpp	jGRASP C or C++ file	☑ .cxx	jGRASP C or C++ file
☑ .hxx	jGRASP C or C++ file	☑ .cc	jGRASP C or C++ file
☑ .ada	jGRASP Ada file	☑ .adb	jGRASP Ada file
☑ .vhdl	jGRASP VHDL file	☑ .vhd	jGRASP VHDL file
☑ .gpj	jGRASP project		

Figure 1-7. Choosing *File Associations*

Figure 1-8 shows the progress of the installation and Figure 1-9 indicates that installation is complete. Click “Finish” to close the Install wizard.



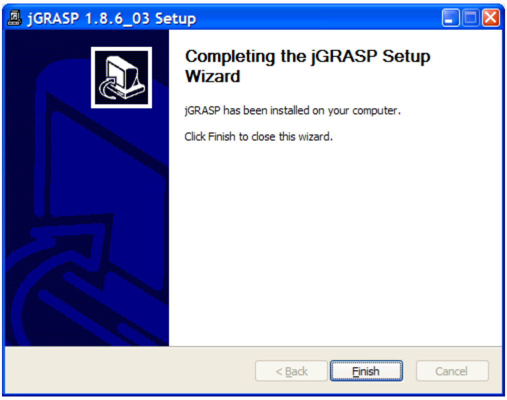
jGRASP 1.8.6_03 Setup

Installing

Please wait while jGRASP is being installed.

Nullsoft Install System v2.24

Figure 1-8. Installing jGRASP



jGRASP 1.8.6_03 Setup

Completing the jGRASP Setup Wizard

jGRASP has been installed on your computer.

Click Finish to close this wizard.

Figure 1-9. Installation complete

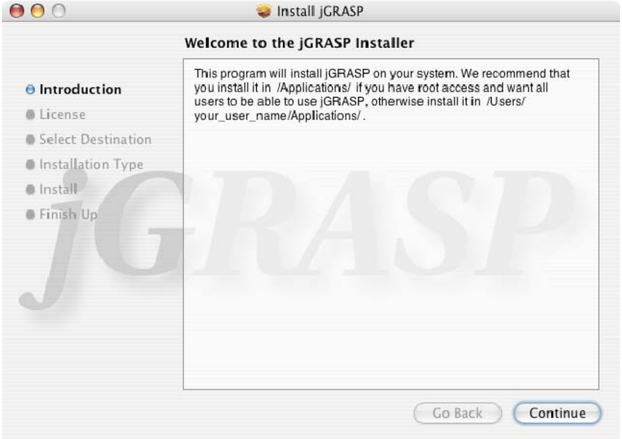
You should find the jGRASP icon on your desktop, and jGRASP should also be listed on the Windows **Start** > **All Programs** menu.

You can start jGRASP by double clicking the icon on your Windows desktop or via the Windows **Start** menu. See “Getting Started” for details.

1.3 Installing on Mac OS X

To install jGRASP on a Mac OS X machine, an administrator password is required. When you download jGRASP, the install file (.pkg.tar.gz) should unzip and untar automatically. If this does not happen, you can use Stuffit Expander [or from a terminal, use "gunzip jgrasp*.tar.gz" then "tar xf jgrasp*.tar"]. You should now be able to double click on the .pkg file to continue the installation.

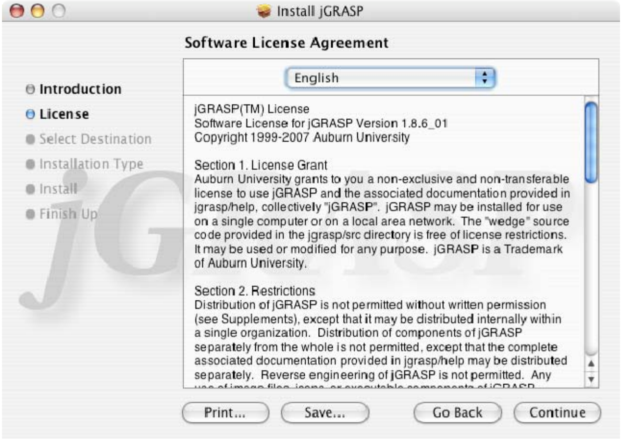
Figure 1-10 shows the introduction screen of the jGRASP Installer with a recommendation regarding the folder in which jGRASP should be installed. In most cases, you will want to install jGRASP in the /Applications/ folder so that anyone who has an account on your machine can use jGRASP.



This program will install jGRASP on your system. We recommend that you install it in /Applications/ if you have root access and want all users to be able to use jGRASP, otherwise install it in /Users/your_user_name/Applications/ .

Figure 1-10. jGRASP Setup dialog

The software license for jGRASP is shown in Figure 1-11. After you have reviewed it, click “Continue”. In the next screen, Figure 1-12, you must agree to the terms of software license by clicking “Agree” in order to continue the installation process. If you click “Disagree” the installation will be cancelled.



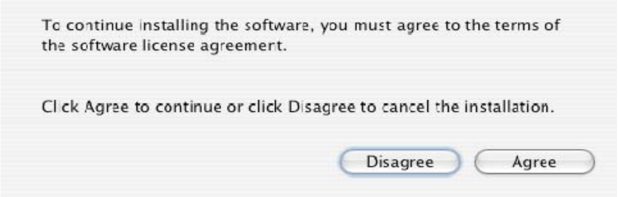
Software License Agreement

jGRASP(TM) License
Software License for jGRASP Version 1.8.6_01
Copyright 1999-2007 Auburn University

Section 1. License Grant
Auburn University grants to you a non-exclusive and non-transferable license to use jGRASP and the associated documentation provided in jgrasp/help, collectively "jGRASP". jGRASP may be installed for use on a single computer or on a local area network. The "wedge" source code provided in the jgrasp/src directory is free of license restrictions. It may be used or modified for any purpose. jGRASP is a Trademark of Auburn University.

Section 2. Restrictions
Distribution of jGRASP is not permitted without written permission (see Supplements), except that it may be distributed internally within a single organization. Distribution of components of jGRASP separately from the whole is not permitted, except that the complete associated documentation provided in jgrasp/help may be distributed separately. Reverse engineering of jGRASP is not permitted. Any

Figure 1-11. License Agreement

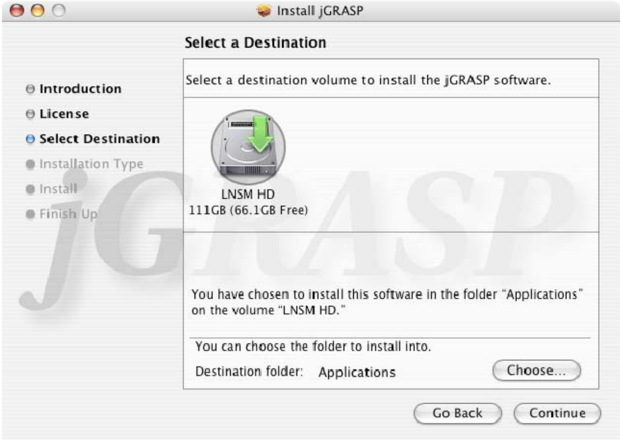


To continue installing the software, you must agree to the terms of the software license agreement.

Click Agree to continue or click Disagree to cancel the installation.

Figure 1-12. Selecting components to install

The screen in Figure 1-14 allows you to select a destination volume and folder for the installation. In the example, the defaults are shown.



Select a Destination

Select a destination volume to install the jGRASP software.

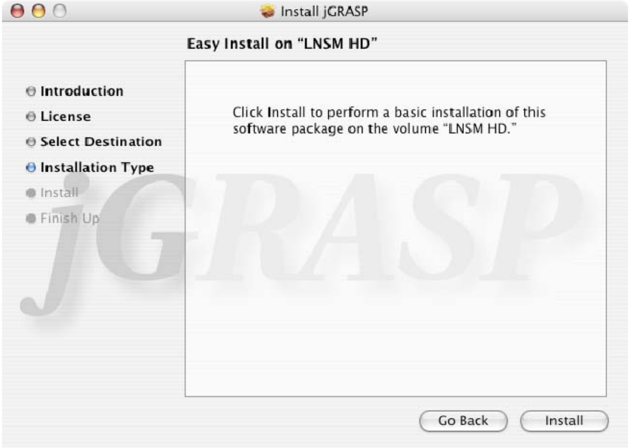
You have chosen to install this software in the folder "Applications" on the volume "LNSM HD."

You can choose the folder to install into.

Destination folder: Applications

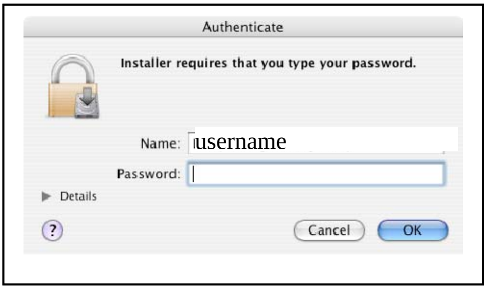
Figure 1-14. jGRASP Setup dialog

When you see the screen in Figure 1-15, just click next. Figure 1-16 shows the "Authenticate" dialog. As indicated above, installing software under Mac OS X requires administrator privileges so you will need to enter your user name and password in order to complete the installation process.



Click Install to perform a basic installation of this software package on the volume "LNSM HD."

Figure 1-15. License Agreement



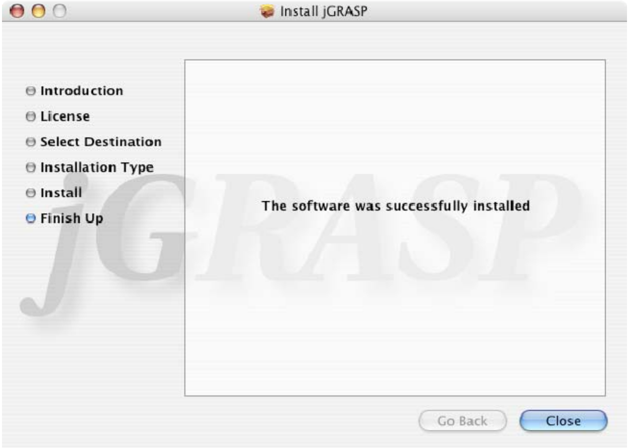
Installer requires that you type your password.

Name: username

Password:

Figure 1-16. Selecting components to install

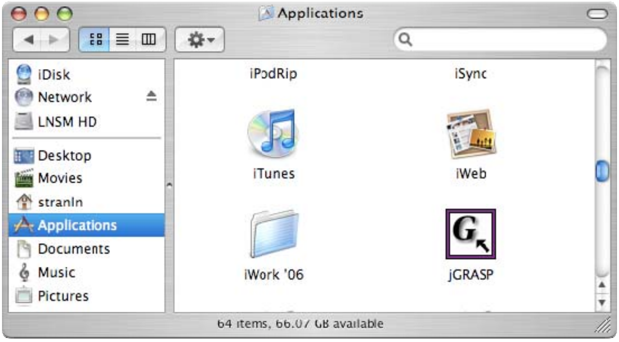
After the jGRASP installation completes, you should see the dialog below.



The software was successfully installed

Figure 1-17. Installation is complete

jGRASP should now be available via the Applications folder.



64 items, 66.07 GB available

Figure 1-18. jGRASP start button

The first time you run jGRASP, the CSD font will be installed on your system, and a soft link to the jGRASP startup script (for command line execution) will be created in /usr/bin or your $HOME/bin directory.

1.4 Installing on Other Systems (including x86 Linux, SPARC Solaris, and NetBSD/i386)

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

While users will find the .zip installation file suitable for Linux and UNIX systems, it will also work on Windows Mac OS X systems. Since the installation file can be unzipped anywhere, the user should note the directory.

1.5 Compilers

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

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

Ada (GNAT)

http://www.cygwin.com (includes gnatmake)

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

http://www.cygwin.com

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

FORTRAN (GNU/Cygnus)

http://www.cygwin.com (includes g77, i.e., GNU Fortran)

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

Java (J2SE JDK, Jikes)

http://java.sun.com/javase/downloads/index.jsp

Assembler (MASM)

http://www.masm32.com/masmdl.htm

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

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

1.6 Setting PATH and CLASSPATH

If you plan to use Java packages other than the standard Java Libraries, these will need to be added to your computer's Java *classpath*. You may also need to add the directories containing various compilers and/or tools to the system path of your machine. If you will not be compiling from the command line, you can make these changes in jGRASP, and the steps for setting the system path and classpath are similar. Since most users will be setting the classpath rather than system path, the steps for doing this are described below.

Go to **Settings** > **PATH/CLASSPATH** > **Workspace** from the control panel (jGRASP desktop menu).

Select the **CLASSPATHS** tab in the settings dialog if it is not already selected, then click the **New** button.

In the "New CLASSPATH / Doc Path" dialog, click the **Browse** button for "Path or JAR File" and navigate to the JAR file or the folder containing the target Java package root, and click the **Choose** button.

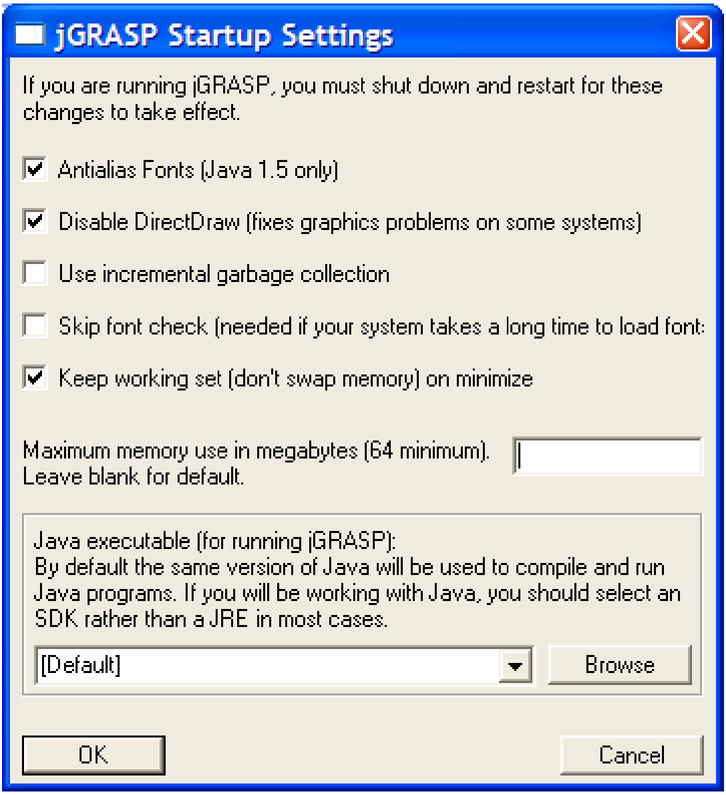
(Optional) If javadoc documentation is provided and you want to set the documentation path for the classes in the JAR or package folder, click the **Browse** button for "Documentation Path", select the folder containing the associated javadoc documentation, and click the **Choose** button.

Finally, click **OK** on the "New CLASSPATH / Doc Path" dialog, and **OK** on the settings dialog.

Once you have one or more classpath entries listed, these can be turned on and off with the associated check boxes. They can also be re-ordered by dragging an entry up or down the list. The order of the entries is important since the classpaths are searched from top to bottom. The same is true for system paths which are entered in the PATH tab of the dialog.

1.7 jGRASP Start Up Settings

For Windows and Mac OS X, jGRASP provides a dialog that allows the user to set jGRASP startup settings. The dialog can be opened from the jGRASP group in the Window's **Start** > **All Programs** menu or the Mac OS X applications directory. If jGRASP is already running, you can open the dialog by clicking **Settings** > **jGRASP Startup Settings** on the jGRASP main menu.



If you are running jGRASP, you must shut down and restart for these changes to take effect.

☑ Antialias Fonts (Java 1.5 only)

☑ Disable DirectDraw (fixes graphics problems on some systems)

☐ Use incremental garbage collection

☐ Skip font check (needed if your system takes a long time to load font:

☑ Keep working set (don't swap memory) on minimize

Maximum memory use in megabytes (64 minimum). Leave blank for default.

Java executable (for running jGRASP):
By default the same version of Java will be used to compile and run Java programs. If you will be working with Java, you should select an SDK rather than a JRE in most cases.

[Default]

Figure 1-19. jGRASP Startup Settings

Each of the check boxes in the dialog is self-explanatory, and the default settings, shown in Figure 1-19, should be suitable for most users. The last item is a drop down list of all the JRE and JDK version that were found on your machine. This setting is used to indicate which version of Java to use for jGRASP itself. Unless indicated by other jGRASP settings, the same version of Java will be used to compile and run Java programs. "[Default]" indicates that jGRASP should use the most recent version of Java found on your machine. If you want to specify a particular version of Java, you may select an entry on the drop down list. Note that if you plan to compile Java programs, then you should select a JDK version rather than a JRE. The disadvantage of selecting a specific JDK is that when a new version of Java is installed, you will have to change the startup setting if you want jGRASP to use it. Hence, "[Default]" is the entry that most users will want. After making changes jGRASP will need to be restarted for the new settings to take effect.

1.8 Plug-Ins for jGRASP

Beginning with version 1.8.7, jGRASP includes "plug-ins" for several useful tools. In order to use a plug-in, the associated tool must be installed on your machine. At startup, jGRASP looks for the tools associated with the plug-ins, and if found, provides access to the tools via the **Tools** menu on the control panel. If you install a tool after jGRASP is already running, you will need to restart jGRASP. If jGRASP does not find the tool or if you want to customize a tool's settings, select the associated Configure option. For example, for Checkstyle, select **Tools** > **Checkstyle** > **Configure**. This will open the **Checkstyle Tool Settings** dialog which will allow you to set Checkstyle's home directory, select a particular **Checks** file, set flags, etc. This allows jGRASP to locate the tool on your machine and then interact with it via the **Compile Messages** tab in the lower window of jGRASP. In the case of Checkstyle, after the tool has been configured, options for **Check File** and **Check Directory** will be available on the Checkstyle menu. **Check File** checks the file in the CSD window that has focus, and **Check Directory** checks all files in the directory containing the file for the CSD window in focus. If the file in the CSD window with focus is in an open project, a **Check Project Files** options will also be available.

Current plug-ins include the following:

(1) **Checkstyle** – automates style checking against a specified "check file". For best compatibility with jGRASP, you are encouraged to install Checkstyle 5. See http://checkstyle.sourceforge.net for details.

(2) **DCD (Dead Code Detector)** – finds never used code in your Java programs. See https://dcd.dev.java.net for details.

(3) **FingBugs** – uses static analysis to look for potential bugs in Java code. http://findbugs.sourceforge.net

2 Getting Started

For the examples in this section, Microsoft Windows and Java will be used. However, much of the information applies to other operating systems and 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 personal computer, you should see the jGRASP icon on the Windows desktop.

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

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

2.1 Starting jGRASP

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

2.3 Creating a New File

2.4 Saving a File

2.5 Building Java Programs - - Recap

2.6 Interactions (Java only)

2.7 Generating a Control Structure Diagram

2.8 Folding a CSD

2.9 Line Numbers

2.10 Compiling a Program – A Few More Details

2.11 Running a Program - Additional Options

2.12 Using the Debugger (Java only)

2.13 Opening a File – Additional Options

2.14 Closing a File

2.15 Exiting jGRASP

2.16 Review and Preview of What's Ahead

2.17 Exercises

2.1 Starting jGRASP

jGRASP

If you are working in a Microsoft Windows environment, you can start jGRASP by double clicking its icon on your Windows desktop. If you don't see the jGRASP icon on the desktop, try the following: click ***Start*** > ***All Programs*** > **jGRASP** (folder) > **jGRASP**.

Depending on the speed of your computer, jGRASP may take between 10 and 30 seconds to start up. The jGRASP virtual **Desktop**, shown below, is composed of a Control Panel with a menu and toolbar across the top and three resizable panes. The *left pane* has tabs for **Browse**, **Debug, Find,** and **Workbench** (Project tab is combined with the Browse tab beginning in version 1.7). The Browse tab, which is the default when jGRASP is started, lists the files in the current directory. The large *right pane* is for UML and CSD Windows. The *lower pane* has tabs for jGRASP messages, Compile messages, Run I//O. and Interactions. The panes can be resized by moving the horizontal or vertical partitions that separate them. Select the partition with the mouse (left-click and hold down) then drag the partition to make a pane larger or

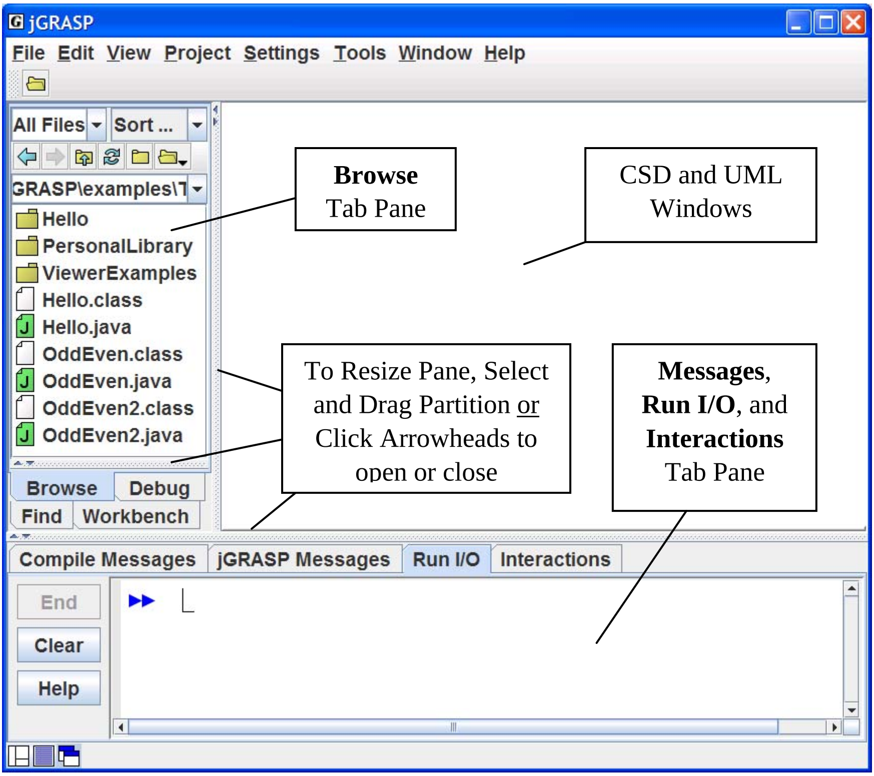


Figure 2-1. The jGRASP Virtual Desktop

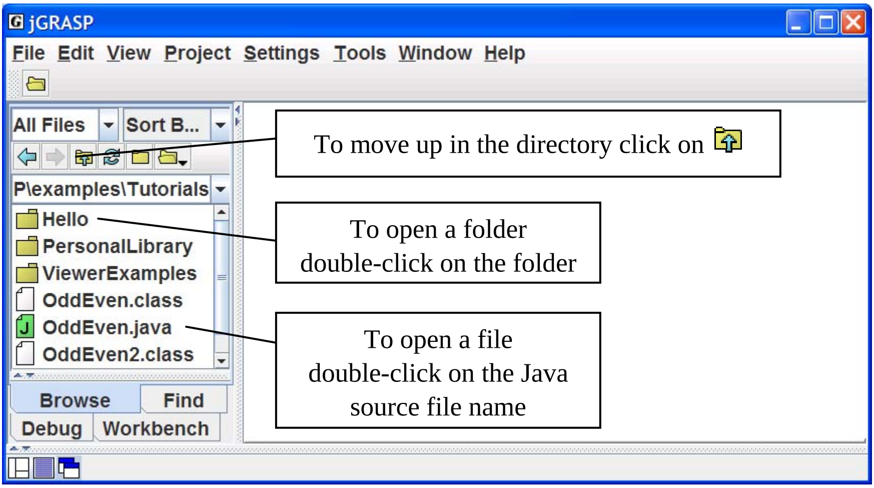
smaller. You can also click the arrowheads on the partition to open and close the pane.

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

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

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

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



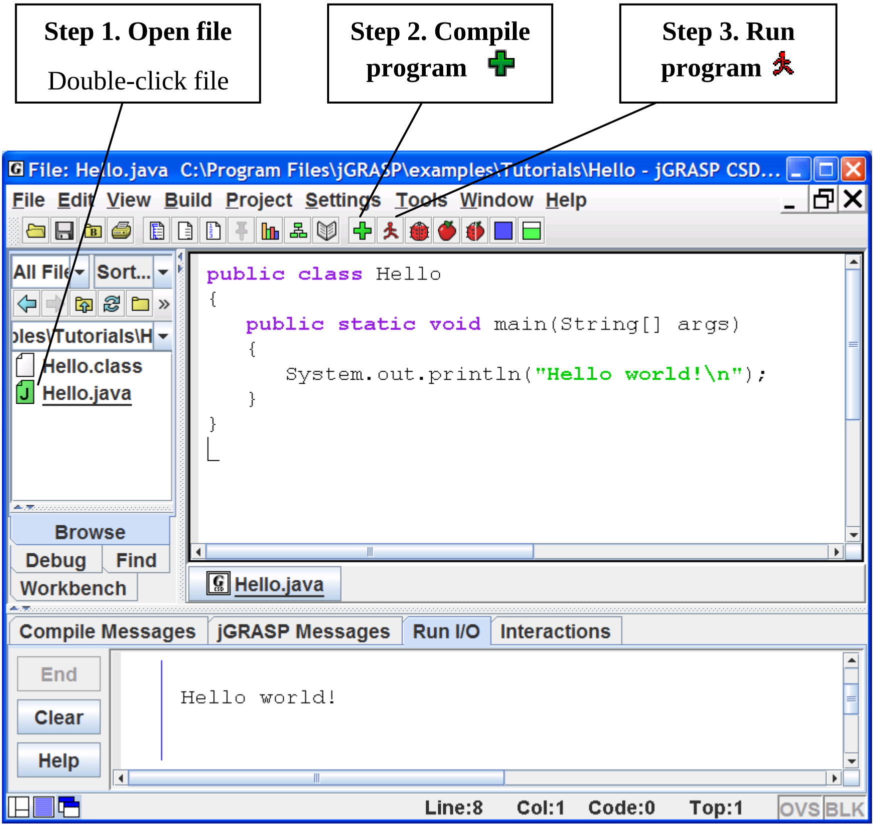
To move up in the directory click on

To open a folder double-click on the folder

To open a file double-click on the Java source file name

Figure 2-2. The jGRASP Virtual Desktop

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



Step 1. Open file

Double-click file

Step 2. Compile program ✚

Step 3. Run program

```
public class Hello
{
   public static void main(String[] args)
   {
      System.out.println("Hello world!\n");
   }
}
```

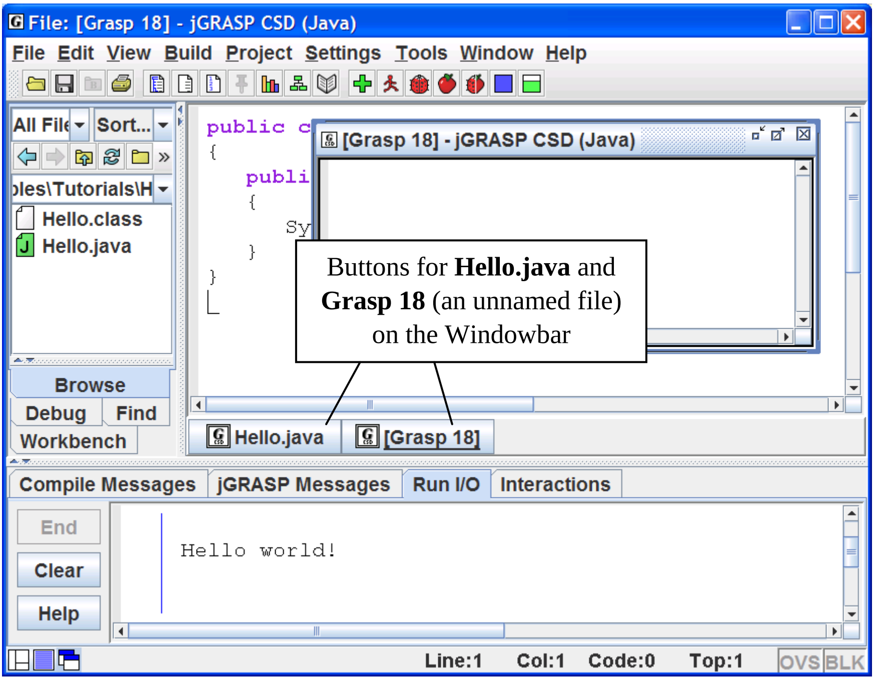
```
Hello world!
```

Figure 2-2. After loading file into CSD Window

2.3 Creating a New File

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

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



```
public c
{
    publi
    {
        Sy
    }
}
```

Buttons for **Hello.java** and **Grasp 18** (an unnamed file) on the Windowbar

```
Hello world!
```

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

In the upper right corner of the CSD window are three buttons that control its display. The first button minimizes the CSD window; the second button maximizes the CSD window or if it is already maximized, restores the CSD window to its previous size. The third button closes the CSD window. You may also make the Desktop itself full screen by clicking the appropriate button in the upper corner of it.

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

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

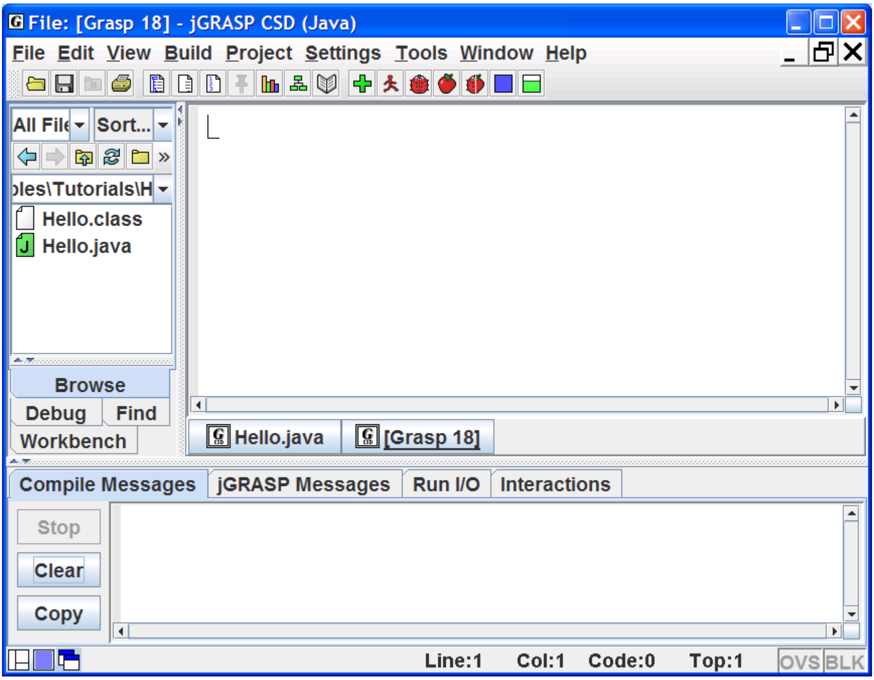
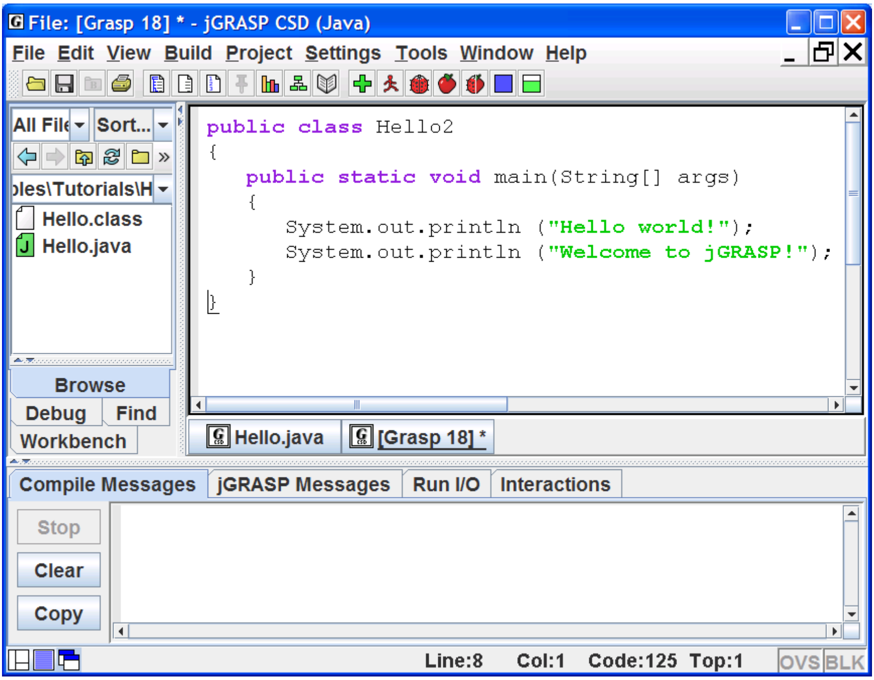


Figure 2-5. CSD Window maximized in Desktop

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

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

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



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

Figure 2-6. CSD Window with program entered

2.4 Saving a File

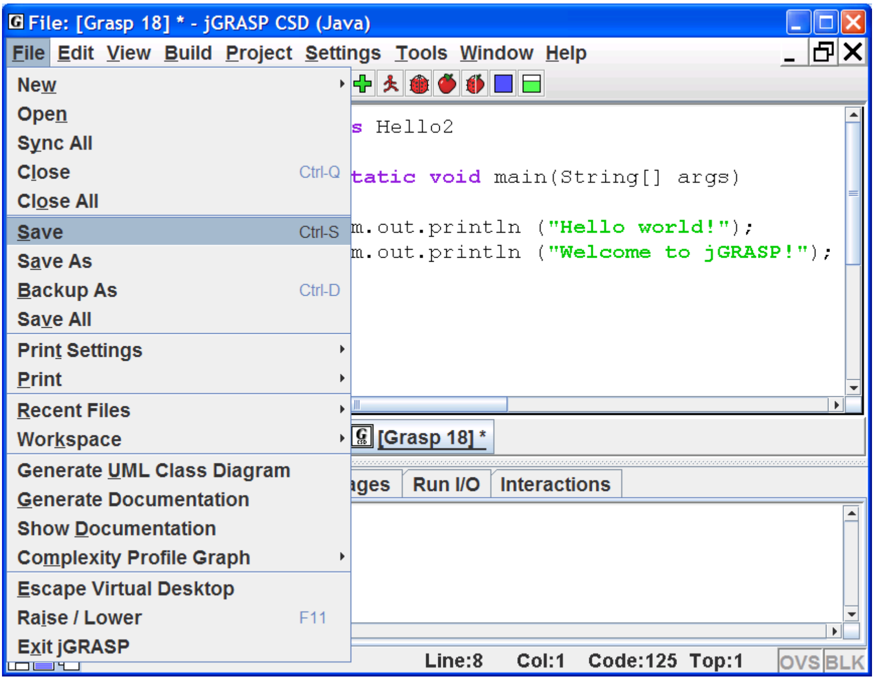
You can save the program as "Hello2.java" by doing any of the following:

(1) Click the Save button 💾 on the toolbar, or

(2) Click **File** > **Save** on menu (see Figure 2-7), or

(3) Press Ctrl-S (i.e., while pressing the Ctrl key, press the “s” key).

If the file has not been saved previously, the Save dialog box pops up with the name of the file set to the name of the class file. Note, in Java, the file name must match the class name (i.e., class Hello2 must be saved as Hello2.java). Be sure you are in the correct directory. If you need to create a new directory, click the folder button on the top row of the Save dialog. When you are in the proper directory and have the correct file name indicated, click the *Save* button on the dialog. After your program has been saved, it should be listed in the Browse tab (see Figure 2.8 on the next page). If you do not see the program in the Browse tab, you may need to navigate to the directory where the file was saved or click B on the toolbar to change the **Browse** tab to the directory of the current file.



```
s Hello2

tatic void main(String[] args)

m.out.println ("Hello world!");
m.out.println ("Welcome to jGRASP!");
```

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

2.5 Building Java Programs - - Recap

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

(1) Enter your source code into a CSD window and then save the program in a file whose name consists of the Java class name and the ".java" extension (e.g., MyProgram.java). You should try to enter your program in chunks so that it will always be compilable.

(2) ✚ Compile the source program (e.g., MyProgram.java) to create the byte code file with a ".class" extension (e.g., MyProgram.class). After attempting to compile your program, you may need to make corrections via the edit window (step 1) based on the messages provided by the compiler and then compile the program again. Note that the .class file is not created until you have a "clean" compile (i.e., no error messages).

(3) 🏃 Run your program. In this step, the byte code or .class file produced by the compiler is executed by the Java Virtual Machine. After you run your program, you should inspect the output (if any) to make sure the program did what you intended. At this point, you may need to find and correct mistakes (bugs). After making the corrections in the edit window (step 1), you will need to compile your program again (step 2). Later, we will use the debugger to step through a program so we can see what happens after each individual statement is executed.

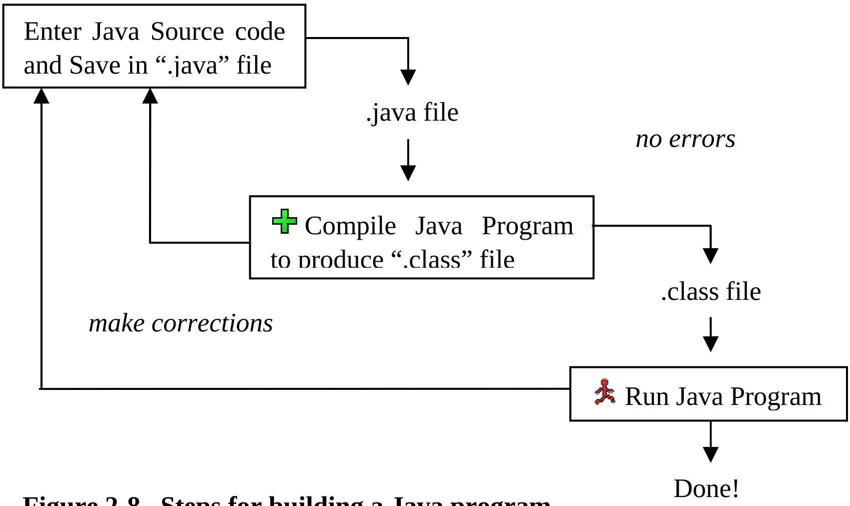

Enter Java Source code and Save in ".java" file
.java file
no errors
✚ Compile Java Program to produce ".class" file
.class file
make corrections
🏃 Run Java Program
Done!

Figure 2-8. Steps for building a Java program

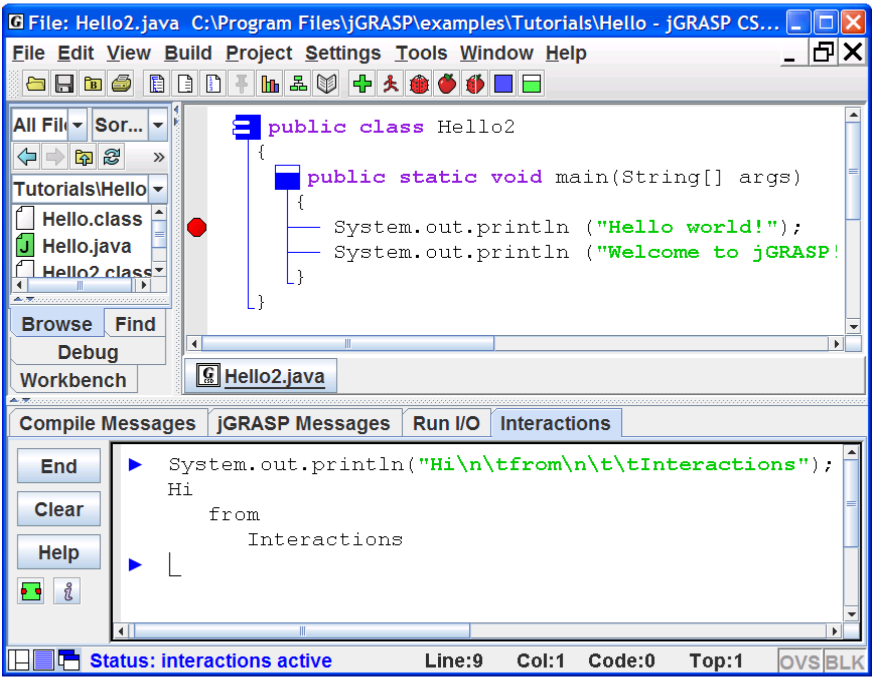
2.6 Interactions (Java only)

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

Consider the following statement that prints a String which includes escape sequences for newline (\n) and tab (\t).

```
System.out.println(“Hello \n\tfrom \n\t\tInteractions!”);
```

Of course you could write a short program that includes this statement, save it, compile it, and run it in order to see the results of executing the statement. However, it may be more convenient to type this statement into the Interactions tab, press ENTER, and quickly see the results as shown below in Figure 2-8.



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

```
▶ System.out.println("Hi\n\tfrom\n\t\tInteractions");
  Hi
      from
          Interactions
▶
```

Status: interactions active   Line:9   Col:1   Code:0   Top:1

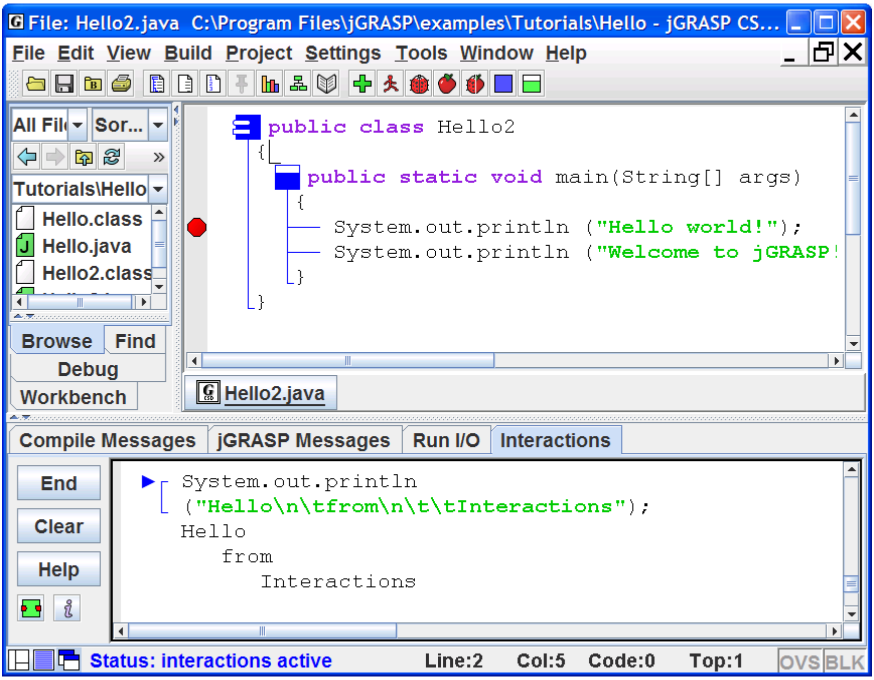
Figure 2-8. Using Interactions

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

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

```
System.out.println                    Shift-ENTER
 ("Hello\n\tfrom\n\t\tInteractions"); ENTER
```

If you simply press ENTER at the end of the first line, Interactions will attempt to execute the incomplete statement and you will get an error message. Figure 2-9 shows the statements above with delayed execution.



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

```
System.out.println
("Hello\n\tfrom\n\t\tInteractions");
Hello
    from
        Interactions
```

Status: interactions active Line:2 Col:5 Code:0 Top:1 OVS BLK

Figure 2-9. Multiple line statement with delayed execution

In the next tutorial, *Getting Started with Objects*, we will see how to use Interactions to create objects on the workbench and then use them in statements and expressions. Of course, you can also interact with local variables of a program running in debug mode after it stops at a breakpoint.

2.7 Generating a Control Structure Diagram

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

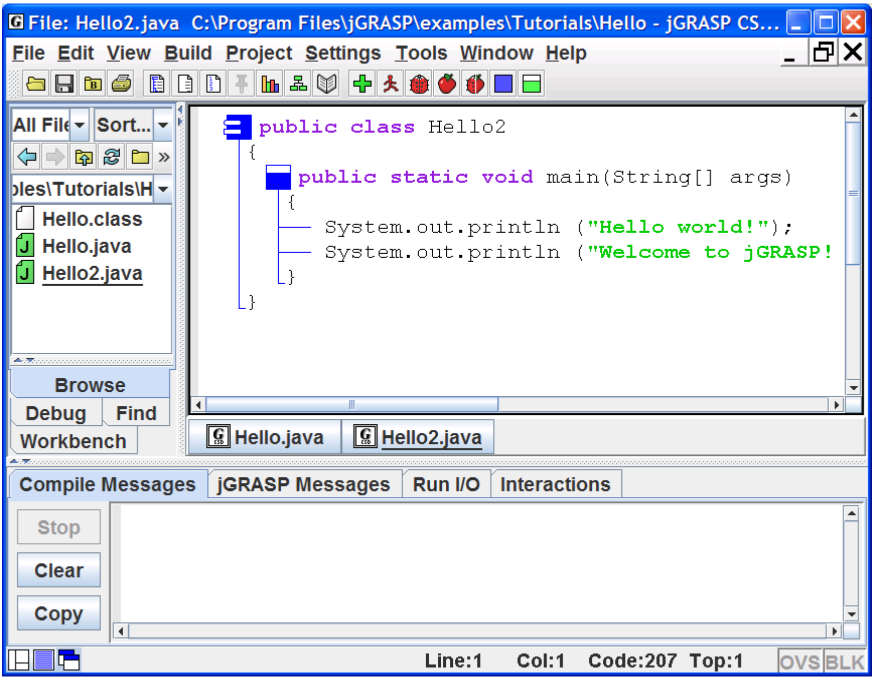
(1) Click the Generate CSD button , or

(2) Click **View > Generate CSD** on the menu, or

(3) Press the F2 key.

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

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.



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

Figure 2-10. After CSD is generated

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

If you are unable find and correct the error, you should try compiling the program since the compiler may provide a more detailed error message (see Compiling below).

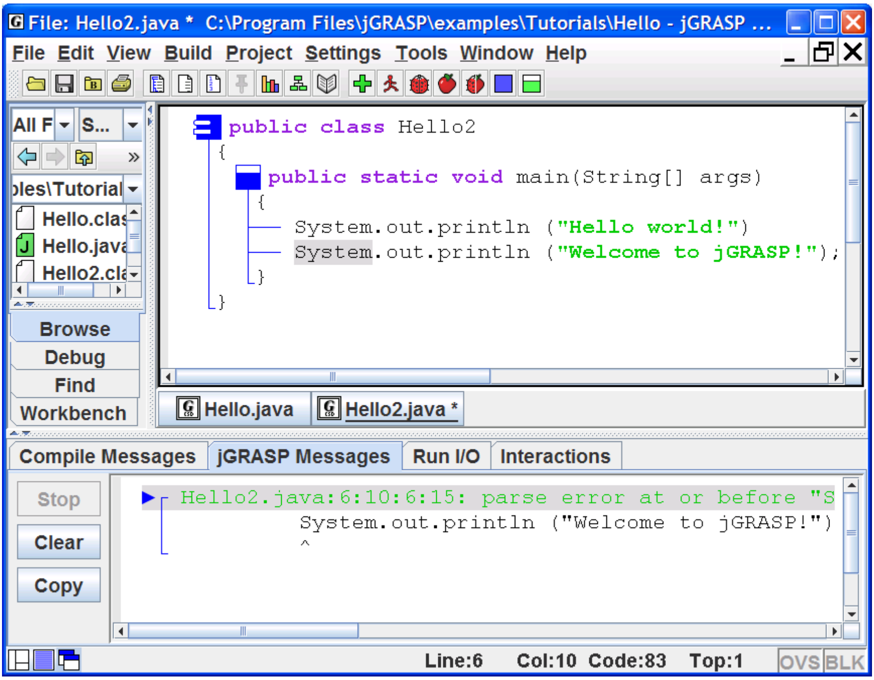
You can remove the CSD by doing any of the following:

(1) Click the Remove CSD button 🗎, or

(2) Click **View** > **Remove CSD** on the menu, or

(3) Press Shift-F2.

Note that it is not necessary to remove the CSD before compiling or saving a program. Your programs will always be saved as plain text. Many users never remove the CSD. In fact, many turn on Auto Generate (**View** > then check ON **Auto Generate CSD**) so that they will always have the CSD with their code.



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

```
Hello2.java:6:10:6:15: parse error at or before "S
            System.out.println ("Welcome to jGRASP!")
            ^
```

Line:6 Col:10 Code:83 Top:1 OVS BLK

Figure 2-11. Syntax error detected

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

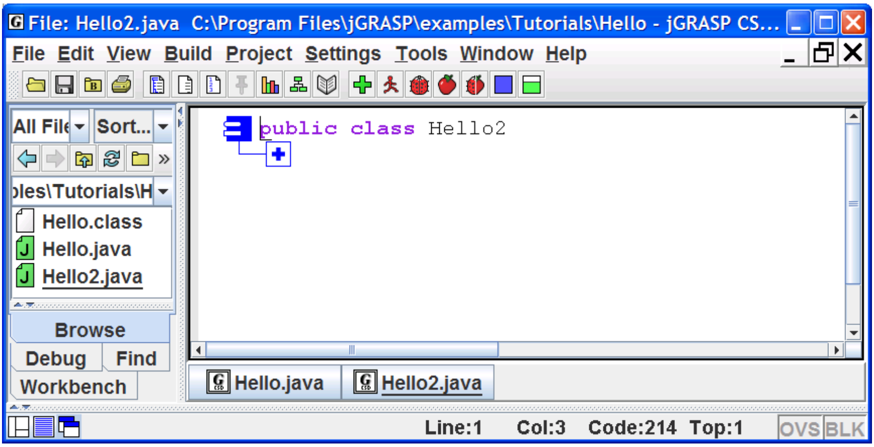
TIP: As you enter a program, try to enter it in “chunks” that are syntactically correct. For example, the following is sufficient to generate the CSD.

```
public class Hello
{
}
```

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

2.8 Folding a CSD

Folding is a CSD feature that becomes increasingly useful as programs get larger. After you have generated the CSD, you can fold your program based on its structure. For example, if you double-click on the class symbol in the program, the entire program is folded (Figure 2-12). Double-clicking on the class symbol again will unfold the program completely. If you double-click on the “plus” symbol, the first layer of the program will be unfolded. Large programs can be unfolded layer by layer as needed. Although the example program has no loops or conditional statements, these may be folded by double-clicking the corresponding CSD control constructs. For other folding options, see the **View** > **Fold** menu.



```
public class Hello2
```

Figure 2-12. Folded CSD

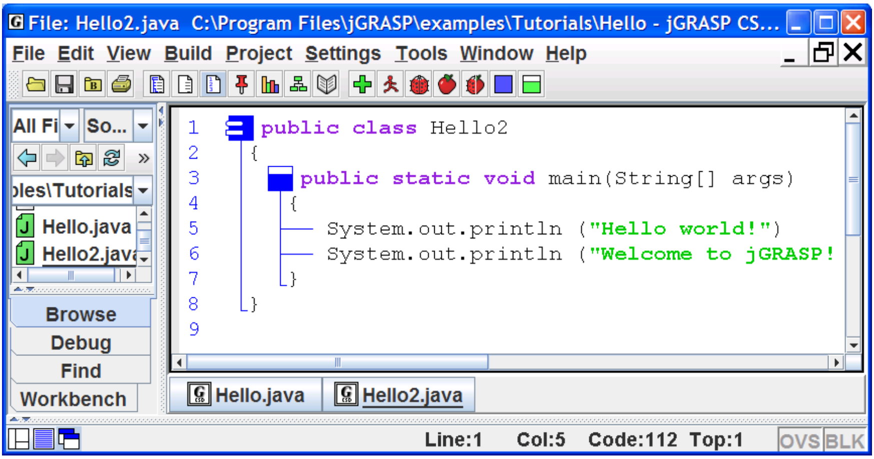
2.9 Line Numbers

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

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

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

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


File: Hello2.java C:\Program Files\jGRASP\examples\Tutorials\Hello - jGRASP CS...
File Edit View Build Project Settings Tools Window Help
All Fi
So...
ples\Tutorials
Hello.java
Hello2.java
Browse
Debug
Find
Workbench
Hello.java
Hello2.java
OVS BLK

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

Line:1 Col:5 Code:112 Top:1

Figure 2-13. Line numbers in the CSD Window

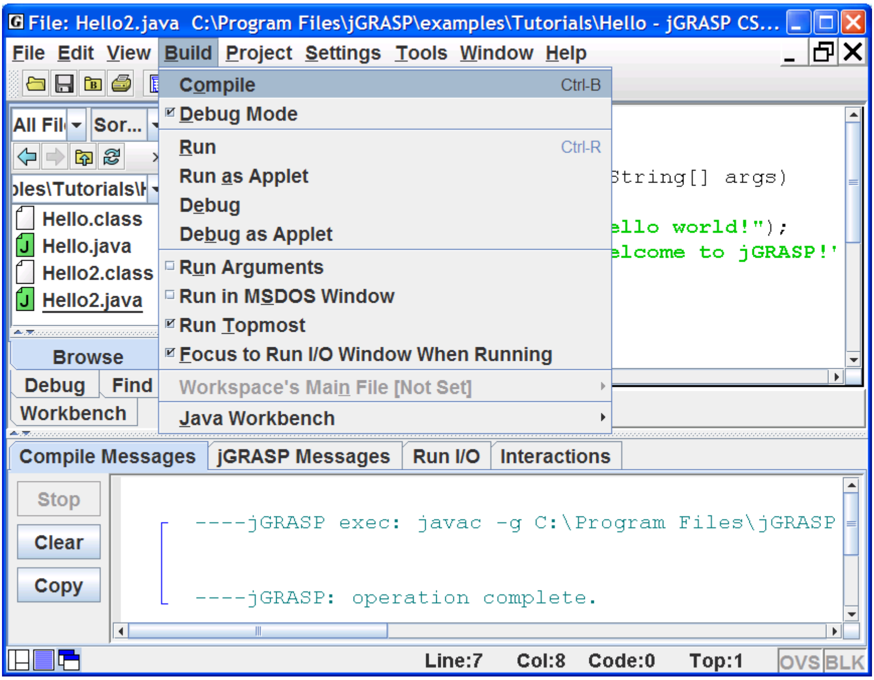
2.10 Compiling a Program – A Few More Details

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

Compile a Java program in jGRASP by clicking the Compile button or by clicking on the Compiler menu: **Build > Compile** (Figure 2-14).

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

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



```
String[] args)
ello world!");
elcome to jGRASP!'
```

```
----jGRASP exec: javac -g C:\Program Files\jGRASP

----jGRASP: operation complete.
```

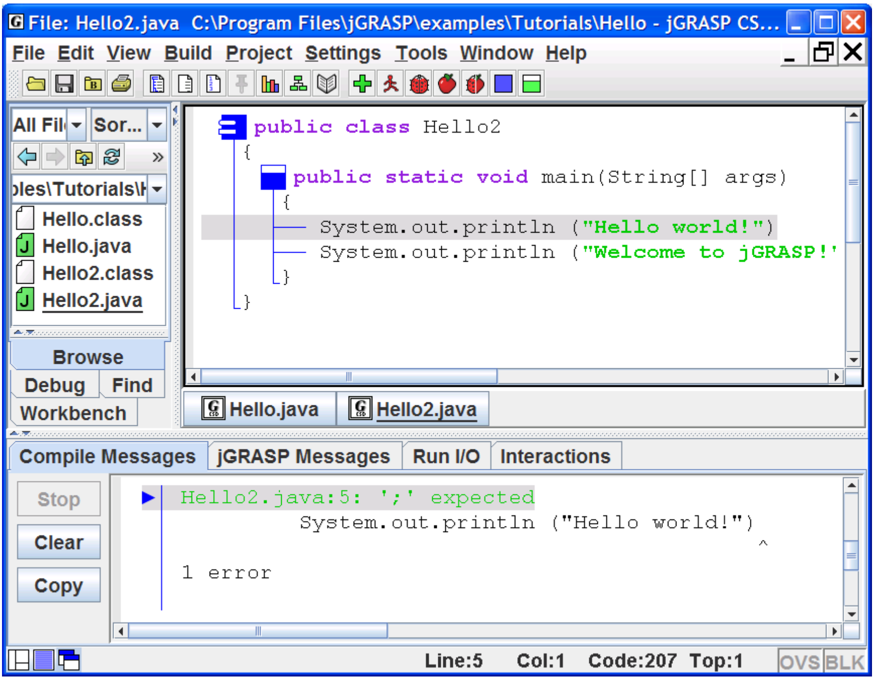
Figure 2-14. Compiling a program

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 2-14. Now you are ready to "Run" the program (see Section *2.11 Running A Program – Additional Options*).

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

Figure 2-15 shows a program with a missing ")" in the first println statement. The error description is highlighted in the Compiler Message tab, and jGRASP automatically scrolls the CSD window to the line where the error most likely occurred and highlights it. If multiple errors are indicated, you should correct all that are obvious and then compile the program again. Sometimes correcting one error can clear up several error messages.

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



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

```
Hello2.java:5: ';' expected
         System.out.println ("Hello world!")
                                            ^
1 error
```

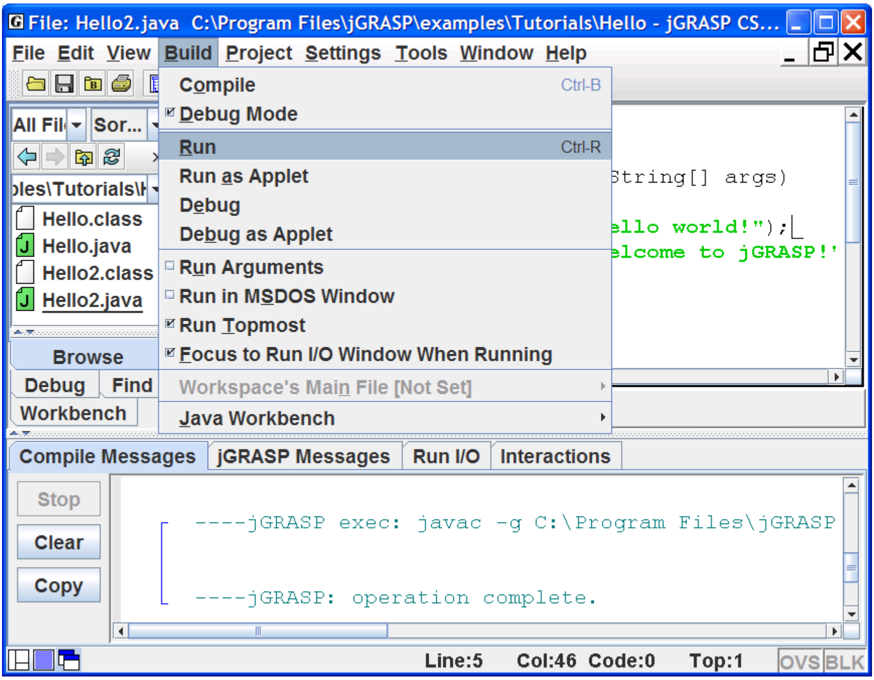
Figure 2-15. Compile time error reported

2.11 Running a Program - Additional Options

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

To run the program, click **Build** > **Run** on the toolbar (Figure 2-16). The options on the Build menu allow you to run your program: as an application (**Run**), as an Applet (**Run as Applet**), as an application in debug mode (**Debug**), and as an Applet in debug mode (**Debug as Applet**). Other options allow you to pass Run arguments, Run in an MS-DOS window rather than the jGRASP Run I/O message pane, and Run Topmost to keep frames and dialogs of the program on top of jGRASP components.

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

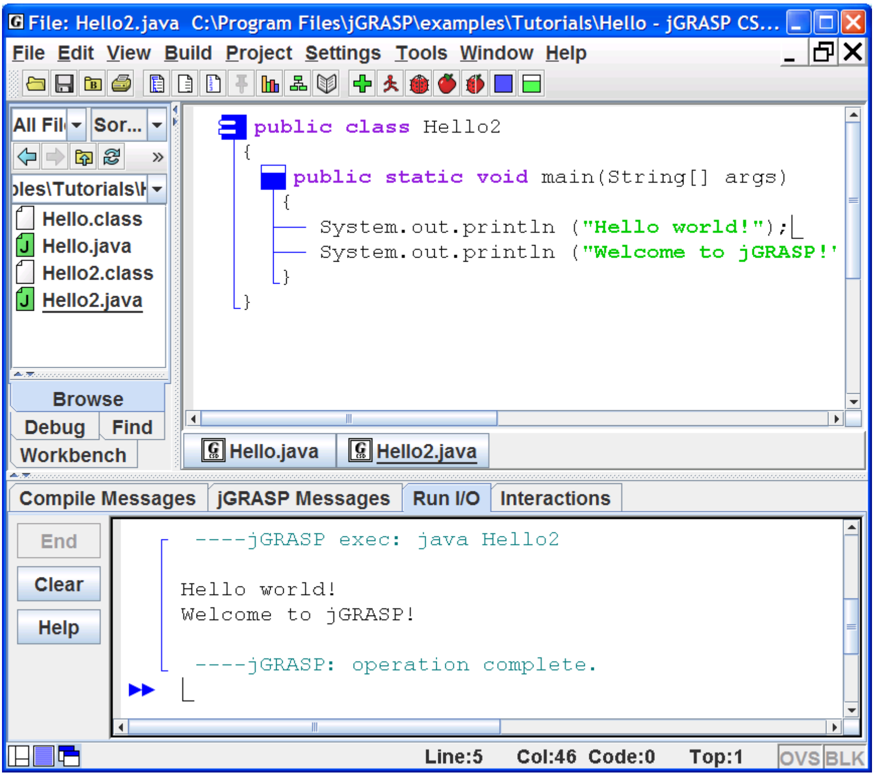


```
String[] args)
ello world!");
elcome to jGRASP!'
```

```
----jGRASP exec: javac -g C:\Program Files\jGRASP

----jGRASP: operation complete.
```

Figure 2-16. Running a program

Output – If a program has any standard input and/or output, the Run I/O tab in the lower pane pops to the top of the Desktop. In Figure 2-17, the output from running the Hello2 program is shown in Run I/O tab.



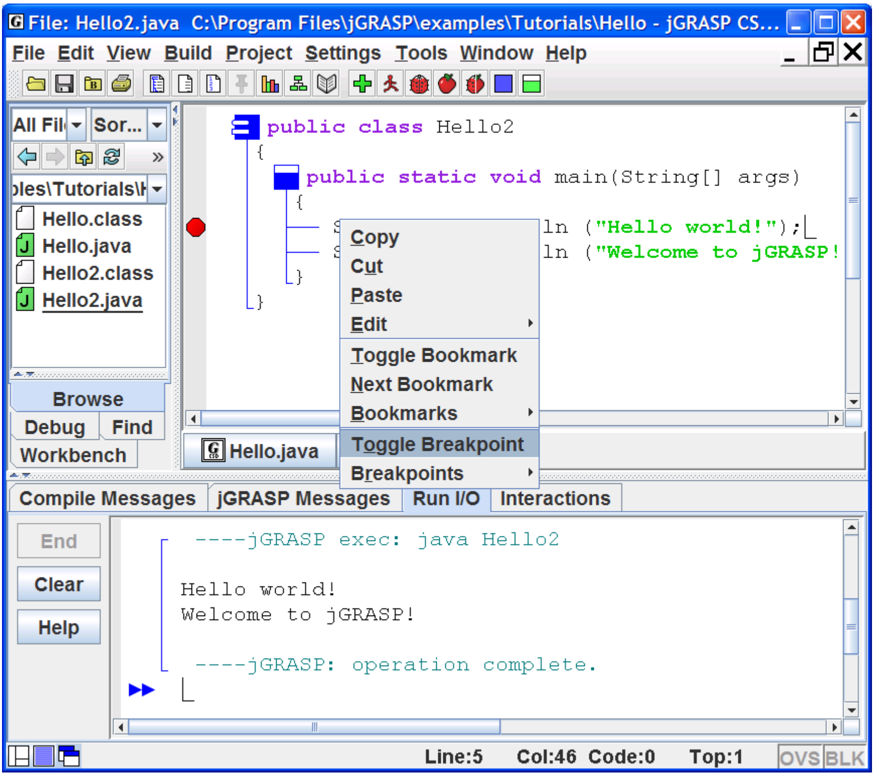
```
public class Hello2
{
   public static void main(String[] args)
   {
      System.out.println ("Hello world!");
      System.out.println ("Welcome to jGRASP!'
   }
}
```

```
 ----jGRASP exec: java Hello2

Hello world!
Welcome to jGRASP!

 ----jGRASP: operation complete.
```

Figure 2-17. Output from running the program

2.12 Using the Debugger (Java only)

jGRASP provides an easy-to-use visual Debugger for Java that allows you to set one or more breakpoints in your program, run the debugger, then after the program reaches a breakpoint, step through your program statement by statement. To set a breakpoint, hover the mouse over the gray column to the left of the line where you want to set the breakpoint. When you see the red breakpoint symbol, left-click the mouse to set the breakpoint. You can also set a breakpoint by left-clicking on the statement where you want your program to stop, then right-clicking to select **Toggle Breakpoint** (Figure 2-18). Alternatively, after left-clicking on the line where you want the breakpoint, click **View > Breakpoints > Toggle Breakpoint**. You should see the red octagonal breakpoint symbol ● appear in the gray area to the left of the line. The statement you select must be an executable statement (i.e., one that causes the program to do something). In the Hello2 program below, a breakpoint has been set on the first of the two *System.out.println* statements, which are the only statements in this program that allow a breakpoint.



```
public class Hello2
{
   public static void main(String[] args)
   {
      S                   ln ("Hello world!");
      S                   ln ("Welcome to jGRASP!
   }
}
```

```
----jGRASP exec: java Hello2

Hello world!
Welcome to jGRASP!

----jGRASP: operation complete.
```

Figure 2-18. Setting a breakpoint

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

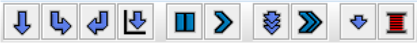
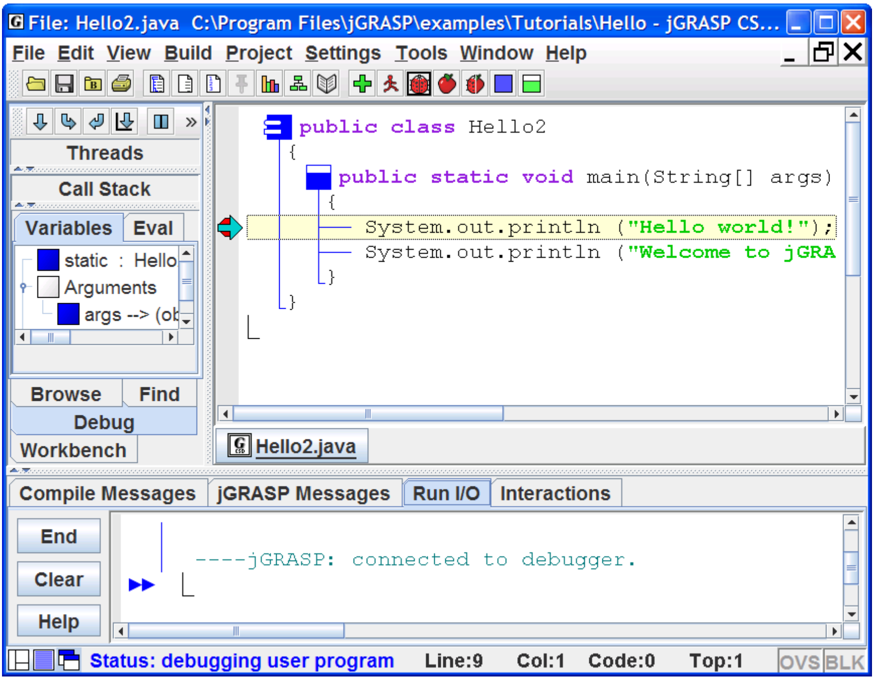


Figure 2-19. Debugger control buttons

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



File: Hello2.java C:\Program Files\jGRASP\examples\Tutorials\Hello - jGRASP CS...

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

static : Hello
Arguments
args --> (ol

```
----jGRASP: connected to debugger.
```

Status: debugging user program   Line:9   Col:1   Code:0   Top:1

Figure 2-20. Stepping with the Debugger

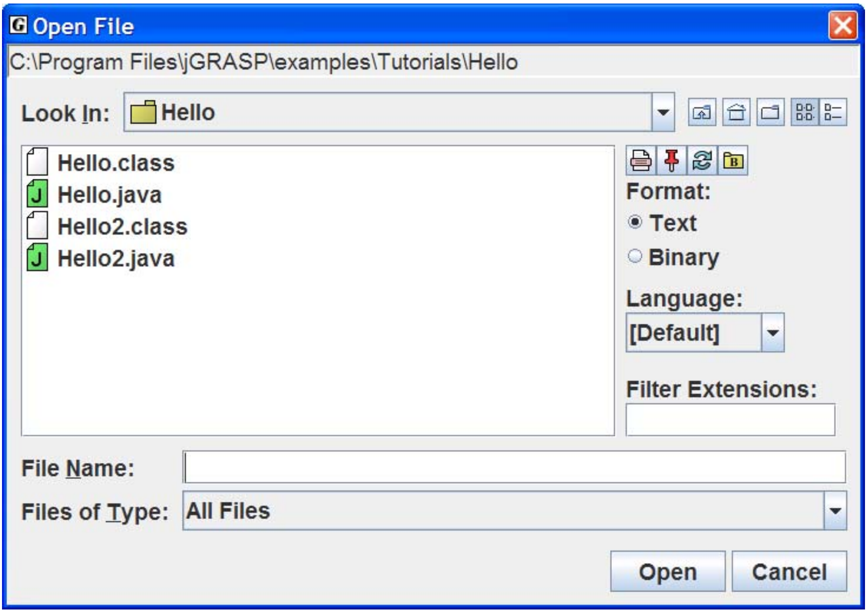
In the example below, the program has stopped at the first output statement. When the step button is clicked, this statement will be executed and “Hello world!” will be output to the Run I/O tab pane. Clicking the step button again will output “Welcome to jGRASP!” on the next line. The third click on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

2.13 Opening a File – Additional Options

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

(1) **Browse Tab** - If the file is listed in jGRASP Browse tab, you can simply double click on the file name, and the file will be opened in a new CSD window. We did this back in section **2.1 Quick Start**. You can also drag a file from the Browse tab and drop it in the CSD window area.

(2) **Menu or Toolbar** - On the menu, click **File > Open** or Click the Open File button on the toolbar. Either of these will bring up the Open File dialog shown in Figure 2-21.



Open File

C:\Program Files\jGRASP\examples\Tutorials\Hello

Look In: Hello

Hello.class
Hello.java
Hello2.class
Hello2.java

Format:
● Text
○ Binary

Language: [Default]

Filter Extensions:

File Name:

Files of Type: All Files

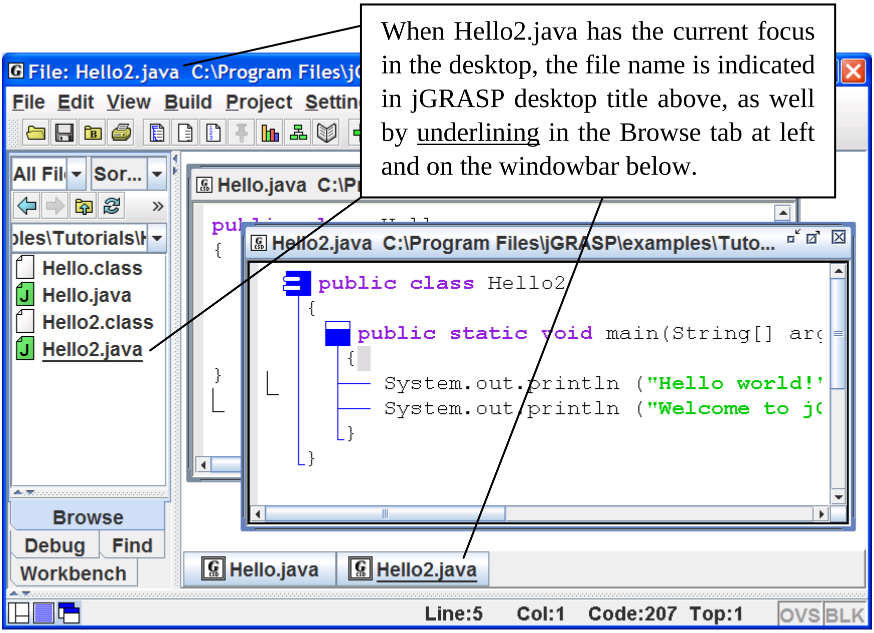
Figure 2-21. Open File dialog

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

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

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

Multiple CSD Windows – When you have multiple files open, each is in a separate CSD window. Each program can be compiled and run from its respective CSD window. When multiple windows are open, the single menu and toolbar go with the top window only, which is said to have "focus" in the desktop. In Figure 2-22, two CSD windows have been opened. One contains Hello.java and the other contains Hello2.java. If the window in which you want to work is visible, simply click the mouse on it to bring it to the top. If you have many windows open, you may need to click the **Window** menu, then click the file name in the list of the open files. However, the easiest way to give focus to a window is to click the window's button on the *windowbar* below the CSD window. As described earlier, these buttons can be reordered by dragging/dropping them on the windowbar. In the figure below, the windowbar has buttons for Hello and Hello2. Notice that Hello2.java is underlined both on the windowbar and in the Browse tab to indicate that it has the current focus. Hello2.java is also displayed in the desktop's blue title bar.



When Hello2.java has the current focus in the desktop, the file name is indicated in jGRASP desktop title above, as well by underlining in the Browse tab at left and on the windowbar below.

```
public class Hello2
{
   public static void main(String[] arg
   {
      System.out.println ("Hello world!'
      System.out.println ("Welcome to j(
   }
}
```

Figure 2-22. Multiple files open

2.14 Closing a File

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

(1) ☒ If the CSD window is maximized, you can close window and file by clicking the Close button at the right end of the top level Menu.

(2) ☒ If the CSD window is not maximized, click the Close button in the upper right corner of the CSD window itself.

(3) **File Menu** – Click **File > Close** or **Close All Files**.

(4) **Window Menu** – Click **Window > Close All Windows**.

In each of the scenarios above, if the file has been modified and not saved, you will be prompted to *Save and Exit*, *Discard Edits*, or *Cancel* before continuing. After the files are closed, your Desktop should look like the one shown in Figure 2-23, which is essentially how we began this tutorial.

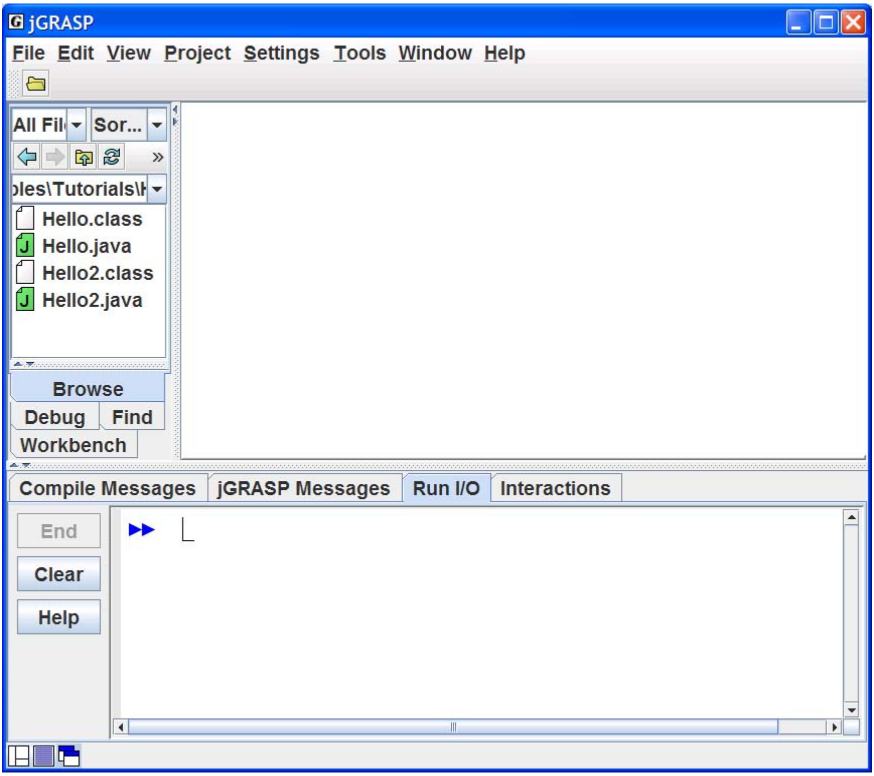


Figure 2-23. Desktop with all CSD Windows closed

2.15 Exiting jGRASP

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

Close jGRASP in either of the following ways:

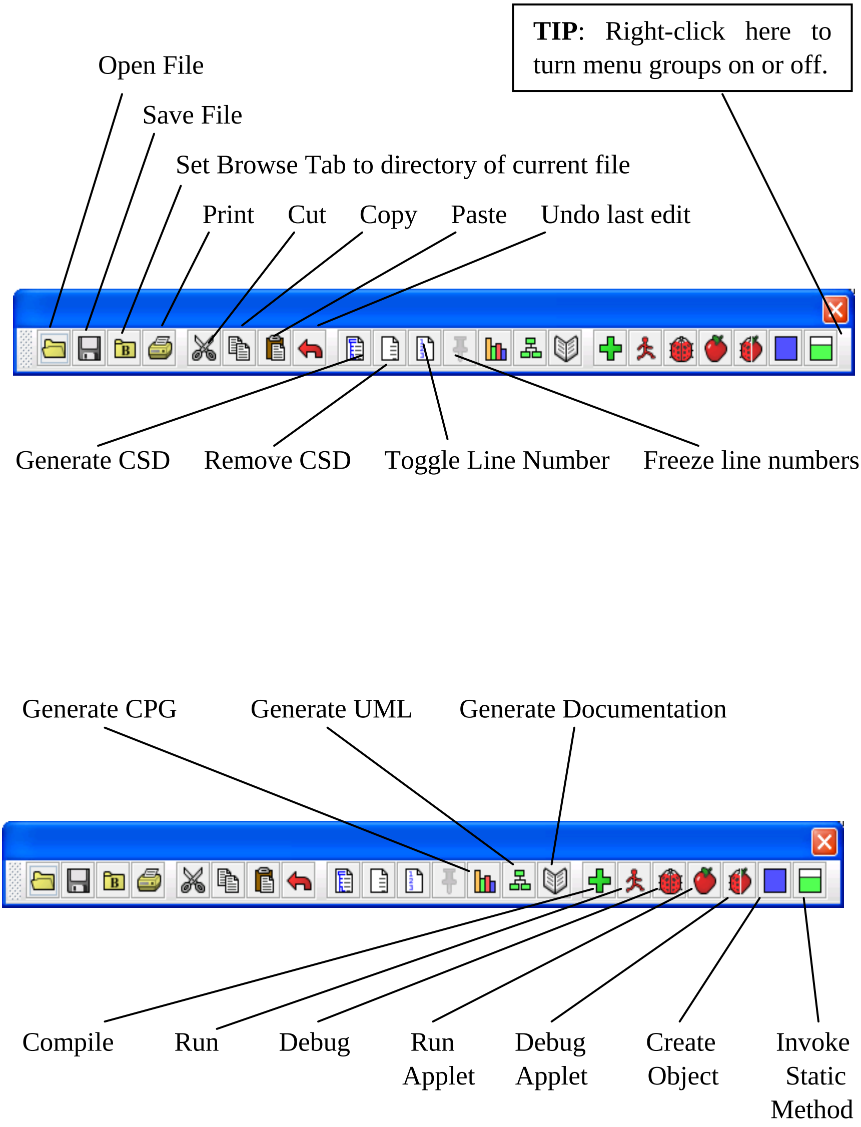
(1) Click the Close button ☒ in the upper right corner of the desktop; or

(2) On the File menu, click **File > Exit jGRASP**.

2.16 Review and Preview of What’s Ahead

As a way of review and also to look ahead, let’s take a look at the jGRASP *toolbar*. Hovering the mouse over a button on the toolbar provides a “tool hint” to help identify its function. Also, **View > Toolbar Buttons** allows you to display *text* labels on the buttons. Figure 2-24 provides a brief description of the each button.

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



TIP: Right-click here to turn menu groups on or off.

Figure 2-24. Toolbar

2.17 Exercises

(1) Create your own program then save, compile, and run it.

(2) Enter several statements and expressions in **Interactions** to immediately see the results of their execution and/or evaluation.

(3) Generate the CSD for your program. On the View menu, turn on Auto Generate CSD (**Settings** > **CSD Window Settings** – then (checkbox) **Auto Generate CSD**).

(4) Display the line numbers for your program.

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

(6) On the Build menu, make sure Debug Mode is ON (indicated by a check box). [Note that Debug Mode should be ON by default, and we recommend that this be left ON.] Recompile your program.

(7) Set a breakpoint on the first executable line of your program then run it with the debugger. Step through each statement, checking the Run I/O window for output.

(8) If you have other Java programs available, open one or more of them, then repeat steps (1) through (5) above for each program.

Notes

3 Getting Started with Objects

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

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

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

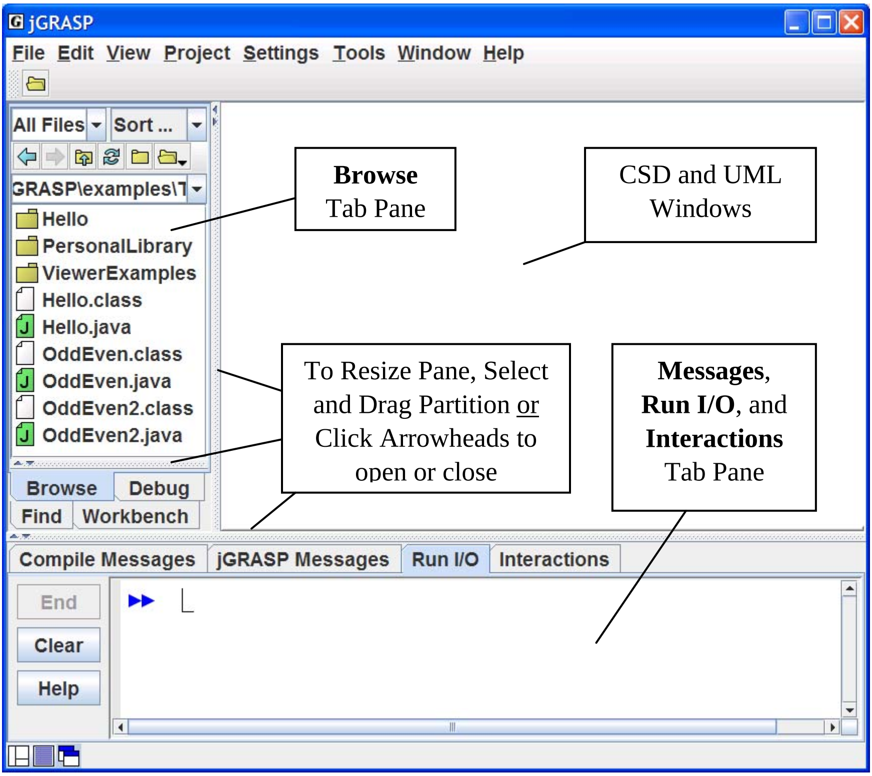
3.1 Starting jGRASP
3.2 Navigating to Our First Example Project
3.3 Opening a Project and UML Window
3.4 Compiling and Running the Program from UML Window
3.5 Exploring the UML Window
3.6 Viewing the Source Code in the CSD Window
3.7 Exploring the Features of the UML and CSD Windows
3.8 Generating Documentation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

3.1 Starting jGRASP

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

jGRASP

If you are working in a Microsoft Windows environment, you can start jGRASP by double clicking its icon on your Windows desktop. If you are working on a PC in a computer lab and you do not see the jGRASP icon on the desktop, try the following: click ***Start*** > ***All Programs*** > ***jGRASP*** (folder) > ***jGRASP***. Depending on the speed of your computer, jGRASP may take between 10 and 30 seconds to start up. The jGRASP virtual **Desktop**, shown below, is composed of a Control Panel with a menu across the top and three panes. The *left pane* has tabs for **Browse**, **Find**,



Browse Tab Pane

CSD and UML Windows

To Resize Pane, Select and Drag Partition or Click Arrowheads to open or close

Messages, **Run I/O**, and **Interactions** Tab Pane

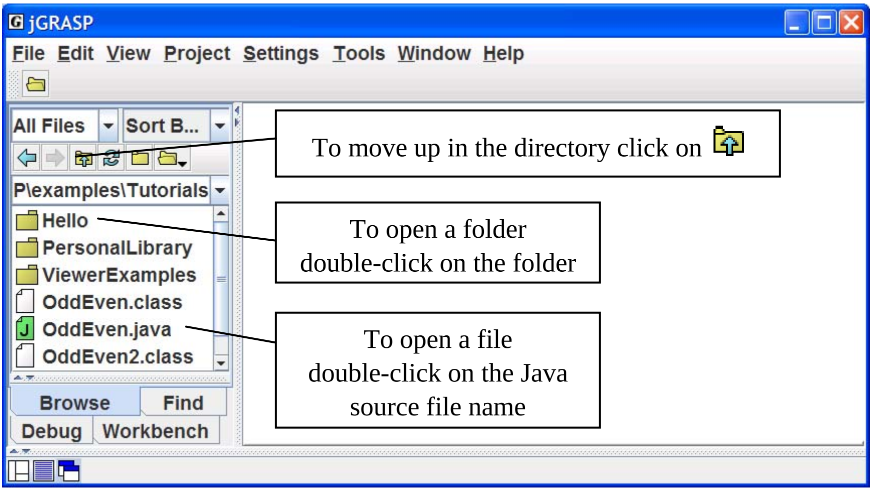
Figure 3-3. The jGRASP Virtual Desktop

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

3.2 Navigating to Our First Example Project

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

The files shown initially in the **Browse tab** will most likely be in your home directory. You can navigate to the appropriate directory by double-clicking on a folder in the Browse tab or by clicking on the up-arrow as indicated in the figure below. The left-arrow and right-arrow allow you to navigate *back* and *forward* to directories that have already been visited during the session. The refresh button 🔄 updates the Browse pane. In the example below, the Browse tab is displaying the contents of the Tutorials folder.



To move up in the directory click on

To open a folder double-click on the folder

To open a file double-click on the Java source file name

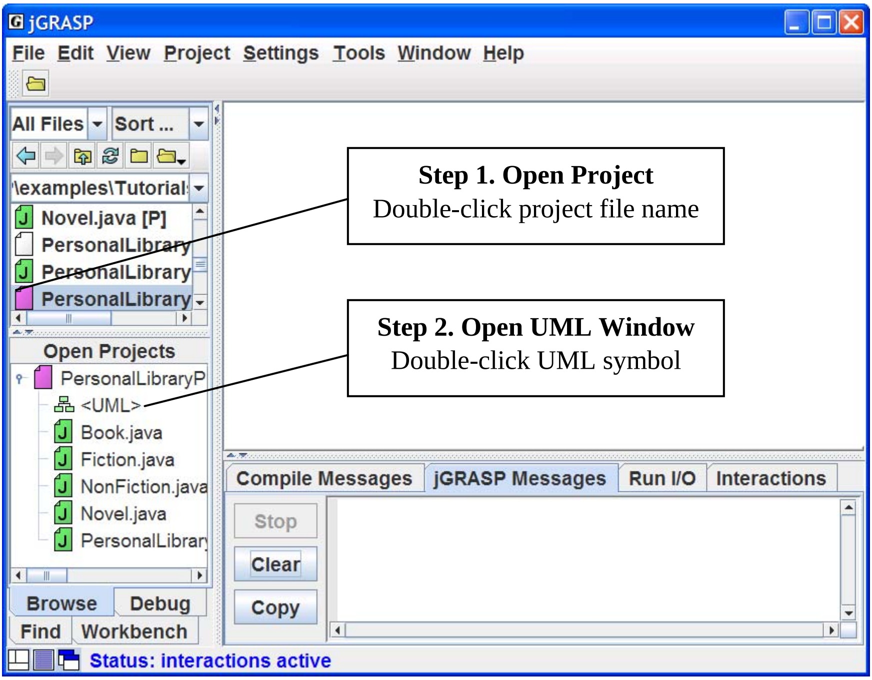
Figure 3-2. The jGRASP Virtual Desktop

3.3 Opening a Project and UML Window

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

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

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



Step 1. Open Project
Double-click project file name

Step 2. Open UML Window
Double-click UML symbol

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

3.4 Compiling and Running the Program from UML Window

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

TIP: Usually the reason for compiling a program is because you have modified or "added" something, hence the green plus ✚.

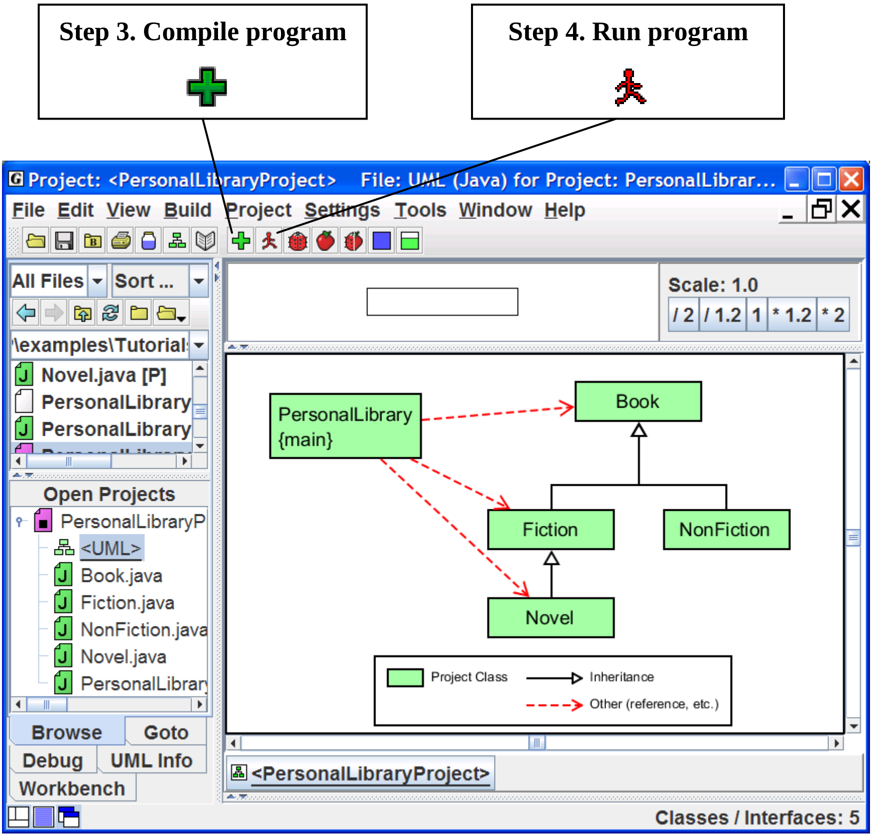


Figure 3-4. After loading file into CSD Window

3.5 Exploring the UML Window

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

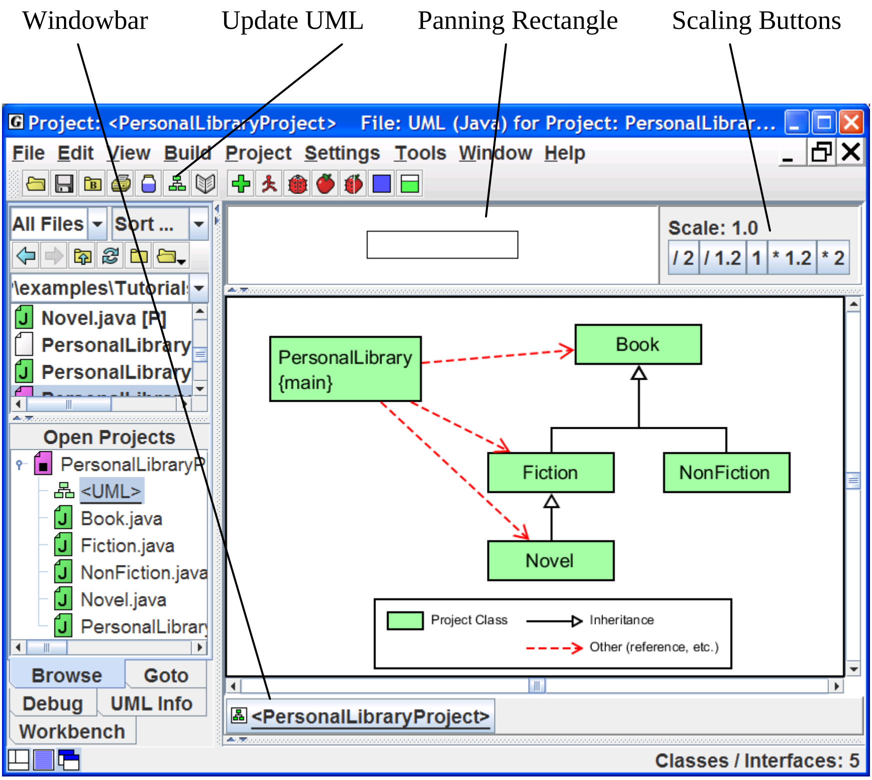
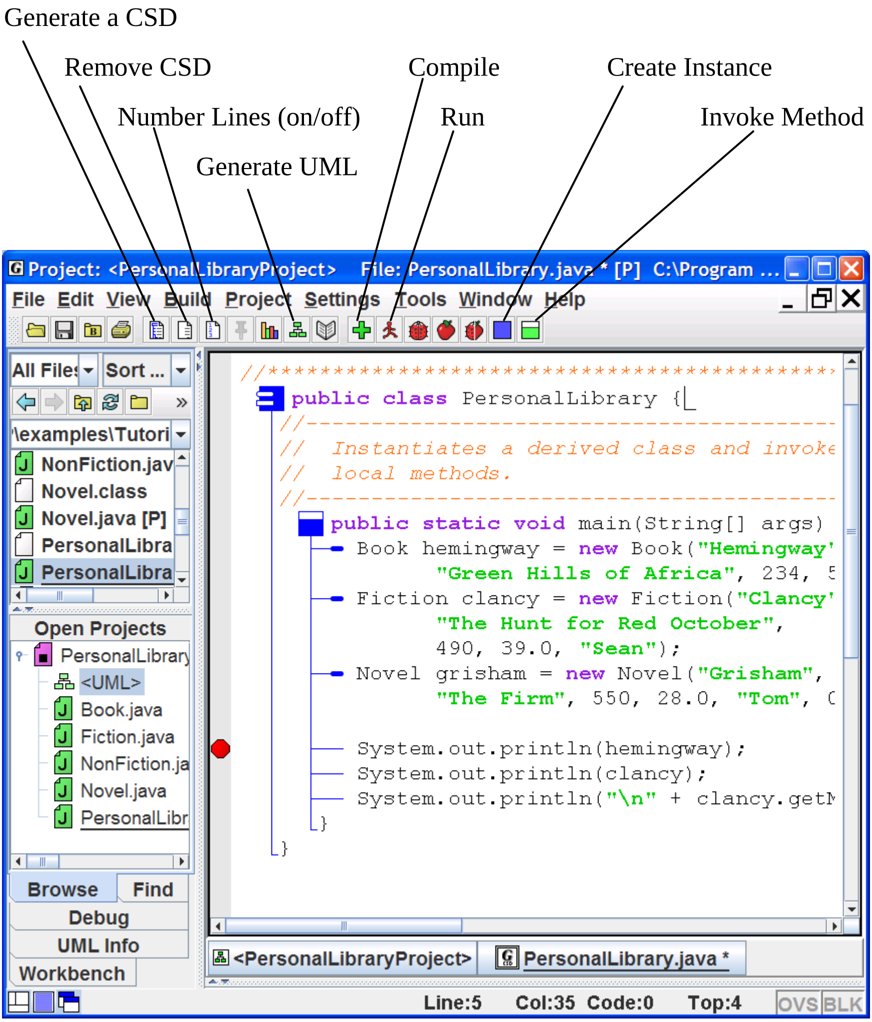


Figure 3-5. UML window with PersonalLibraryPorject

3.6 Viewing the Source Code in the CSD Window

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



```
//*********************************************
public class PersonalLibrary {
   //-------------------------------------------
   //   Instantiates a derived class and invoke
   //   local methods.
   //-------------------------------------------
      public static void main(String[] args)
         Book hemingway = new Book("Hemingway'
               "Green Hills of Africa", 234, 5
         Fiction clancy = new Fiction("Clancy'
               "The Hunt for Red October",
               490, 39.0, "Sean");
         Novel grisham = new Novel("Grisham",
               "The Firm", 550, 28.0, "Tom", (

         System.out.println(hemingway);
         System.out.println(clancy);
         System.out.println("\n" + clancy.get
      }
}
```

Figure 3-6. After the CSD is generated

3.7 Exploring the Features of the UML and CSD Windows

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

3.7.1 Viewing the source code for a class

(1) In the UML diagram, double-click on the PersonalLibrary class. This should open the source file in a CSD window. Notice a button for this CSD window is added to the windowbar. You should also see a button for the UML window.

(2) Review the source code in the CSD window; generate the CSD; fold and unfold the CSD; turn line numbers on and off. [See Sections 2.7 - 2.9 in *Getting Started* for details.]

(3) On the windowbar, click the button for the UML window to pop it to the top. *Remember to do this anytime you need to view the UML window.*

(4) View the source code for the other classes by: (1) double-clicking on the class in the UML diagram, (2) double-clicking on the class in the Open Projects section of the Browse tab, or (3) double-clicking on the file name in the upper section of the Browse tab.

(5) Close one or more of the CSD windows by clicking the **X** in the upper right corner of the CSD window.

3.7.2 Displaying class information

(1) In the UML window, select the Fiction class by left-clicking on it.

(2) Right-click on it and select Show Class Info. This should pop the **UML Info** tab to the top in the left pane of the Desktop, and you should be able to see the **fields**, **constructors**, and **methods** of the Fiction class.

(3) In the UML Info tab, double-click on the getMainCharacter() method. This should open a CSD window with the first executable line in the method highlighted.

(4) Close the CSD window by clicking the X in the upper right corner.

3.7.3 Displaying Dependency Information

(1) In the UML window, select the arrow between PersonalLibrary and Fiction by left-clicking on it.

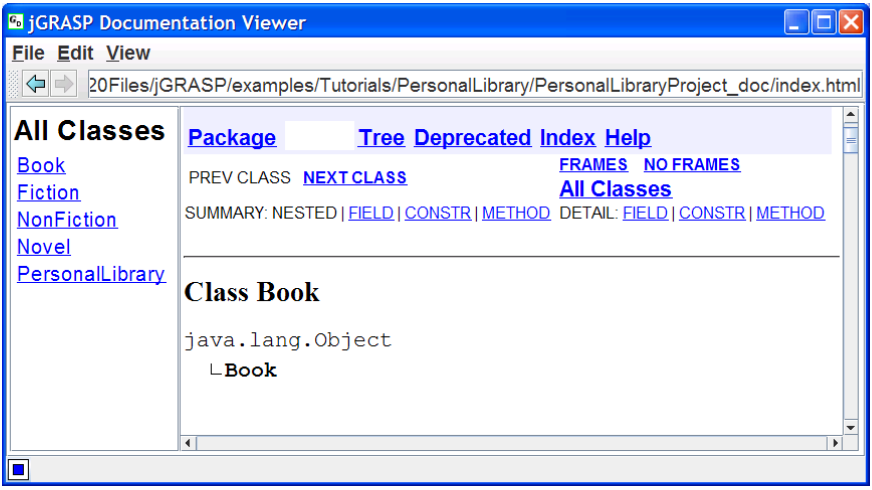
(2) If the UML Info tab is not showing in the left pane of the desktop, right-click on the arrow and select Show Dependency Info. Alternatively, you can click the UML Info tab near the bottom of the left pane.

(3) Review the information listed in the UML tab. As the arrow in the diagram indicates, PersonalLibrary uses a constructor from Fiction as well as the getMainCharacter() method.

(4) Double-click on the getMainCharacter method. This should open a CSD window for PersonalLibrary with the line highlighted where the method is invoked.

3.8 Generating Documentation for the Project

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


jGRASP Documentation Viewer
File Edit View
All Classes
Book
Fiction
NonFiction
Novel
PersonalLibrary
Package Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

20Files/jGRASP/examples/Tutorials/PersonalLibrary/PersonalLibraryProject_doc/index.html

Class Book

```
java.lang.Object
  └Book
```

Figure 3-7. 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 UML window opened for the PersonalLibraryProject. Earlier, we learned how to run the program as an application using the Run button 🏃. Since *main* is a static method, we can also invoke it directly from the class diagram by right-clicking on PersonalLibary and selecting **Invoke Method**. Alternatively, you can select the PersonalLibrary class, and then click the Invoke Method button 🟩 on the toolbar. When the Invoke Method dialog pops up, select and invoke *main* (without parameters). Try this now.

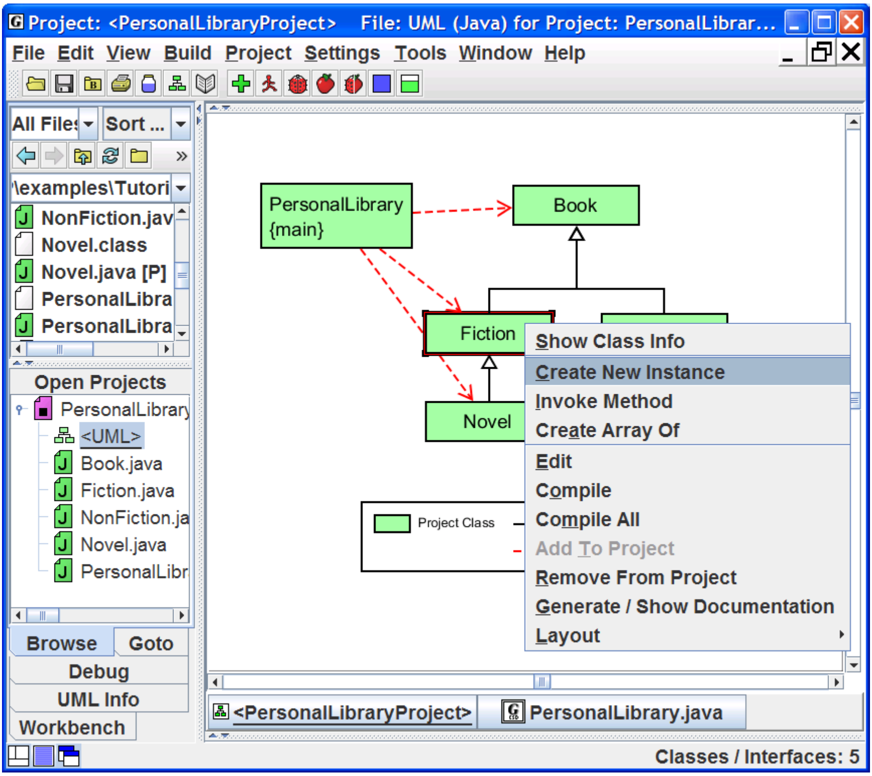
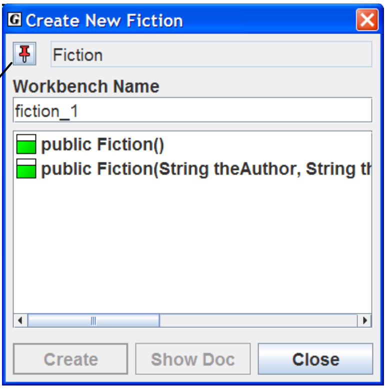


Figure 3-8. Creating an Object for the Workbench

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

Click on "stick-pin" 📌 to keep dialog open.



Fiction

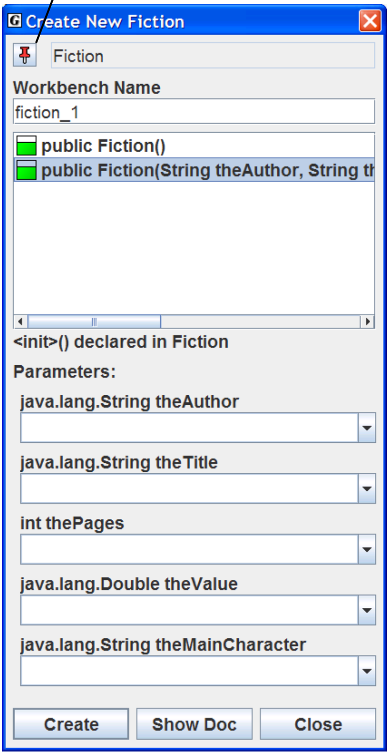
Workbench Name

fiction_1

🟩 public Fiction()

🟩 public Fiction(String theAuthor, String tl

Figure 3-9. Selecting a constructor



Fiction

Workbench Name

fiction_1

🟩 public Fiction()

🟩 public Fiction(String theAuthor, String tl

<init>() declared in Fiction

Parameters:

java.lang.String theAuthor

java.lang.String theTitle

int thePages

java.lang.Double theValue

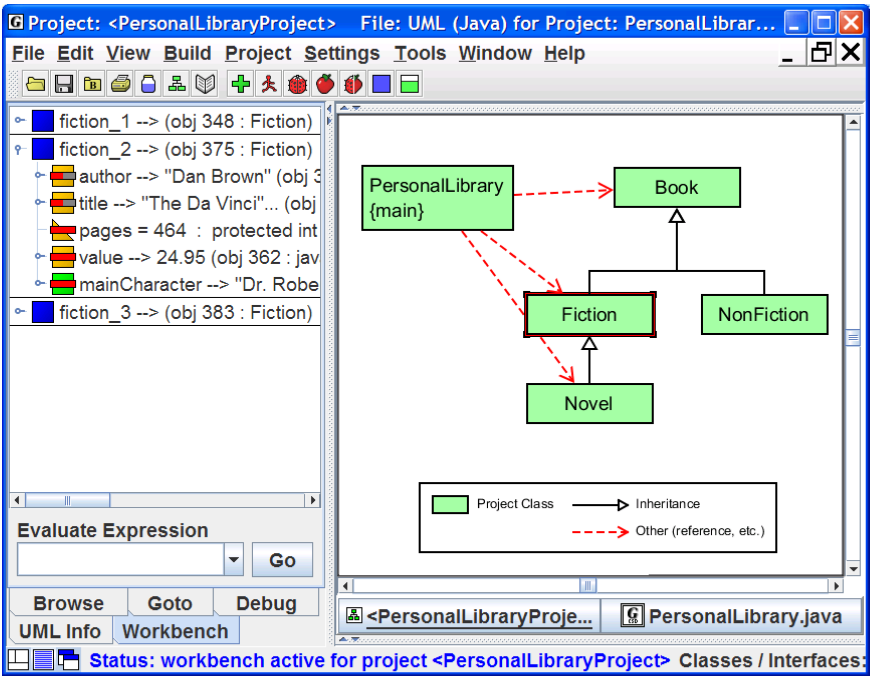
java.lang.String theMainCharacter

Figure 3-10. Constructor with parameters

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

In Figure 3-11, the Workbench tab is shown after two instances of Fiction and one of Novel have been created. The second object, fiction_2, has been expanded so that the fields (mainCharacter, author, title, and pages) can be viewed. An object can be expanded or contracted by clicking on its name. Notice that three fields in fiction_2 are also objects (i.e., instances of the String class); they too can be expanded.

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



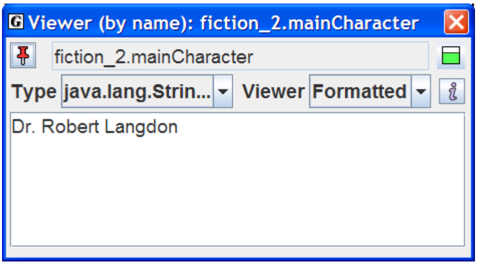
```
fiction_1 --> (obj 348 : Fiction)
fiction_2 --> (obj 375 : Fiction)
  author --> "Dan Brown" (obj 3
  title --> "The Da Vinci"... (obj
  pages = 464  :  protected int
  value --> 24.95 (obj 362 : jav
  mainCharacter --> "Dr. Robe
fiction_3 --> (obj 383 : Fiction)
```

Status: workbench active for project <PersonalLibraryProject> Classes / Interfaces:

Figure 3-11. Workbench with three Fiction objects

3.10 Opening a Viewer Window

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



Viewer (by name): fiction_2.mainCharacter

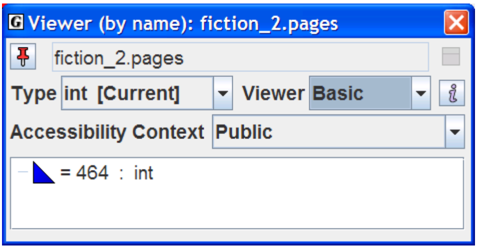
fiction_2.mainCharacter

Type java.lang.Strin... Viewer Formatted

Dr. Robert Langdon

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

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



Viewer (by name): fiction_2.pages

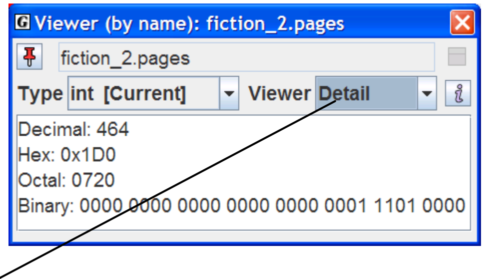
fiction_2.pages

Type int [Current] Viewer Basic

Accessibility Context Public

= 464 : int

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



Viewer (by name): fiction_2.pages

fiction_2.pages

Type int [Current] Viewer Detail

Decimal: 464
Hex: 0x1D0
Octal: 0720
Binary: 0000 0000 0000 0000 0000 0001 1101 0000

Select *view* from drop-down list.

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

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

3.11 Invoking a Method

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

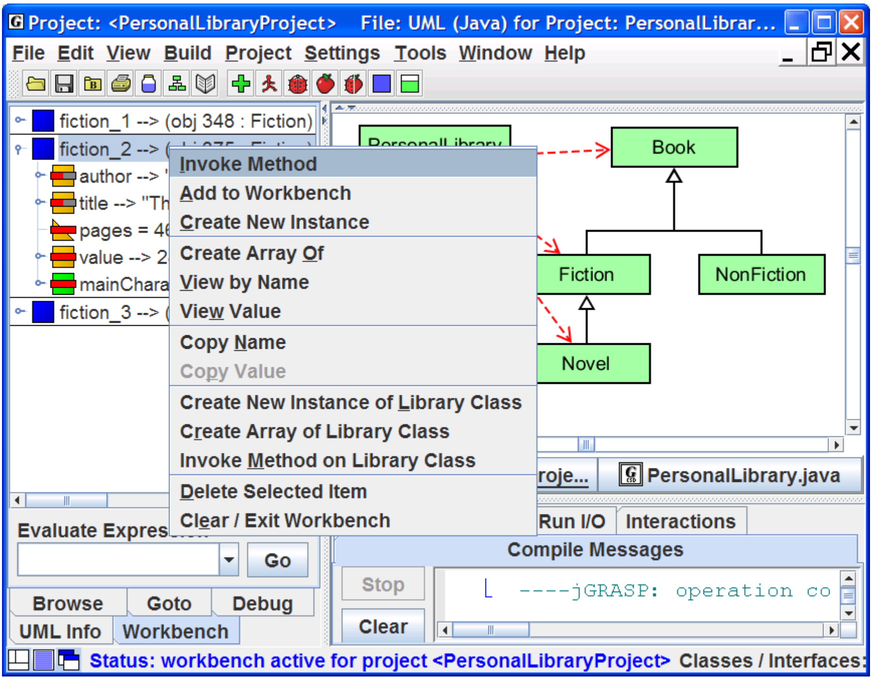
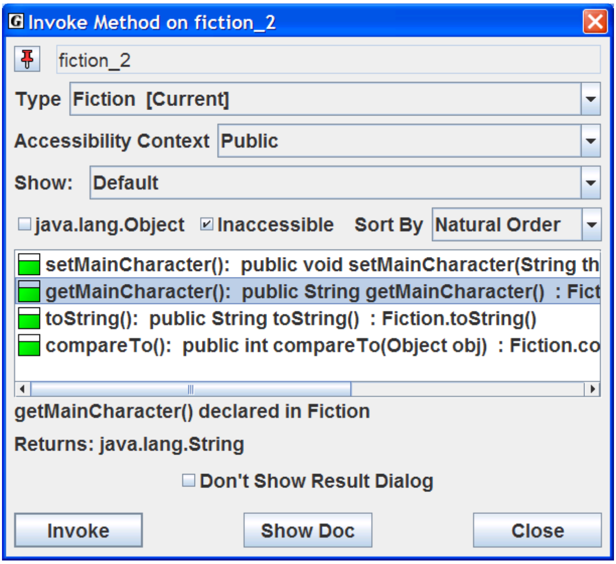


Figure 3-15. Workbench with two instances of Fiction

been generated for the project.



Invoke Method on fiction_2

fiction_2

Type Fiction [Current]

Accessibility Context Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

🟩 setMainCharacter(): public void setMainCharacter(String th
🟩 getMainCharacter(): public String getMainCharacter() : Fict
🟩 toString(): public String toString() : Fiction.toString()
🟩 compareTo(): public int compareTo(Object obj) : Fiction.co

getMainCharacter() declared in Fiction

Returns: java.lang.String

□ Don't Show Result Dialog

Figure 3-16. Selecting a method

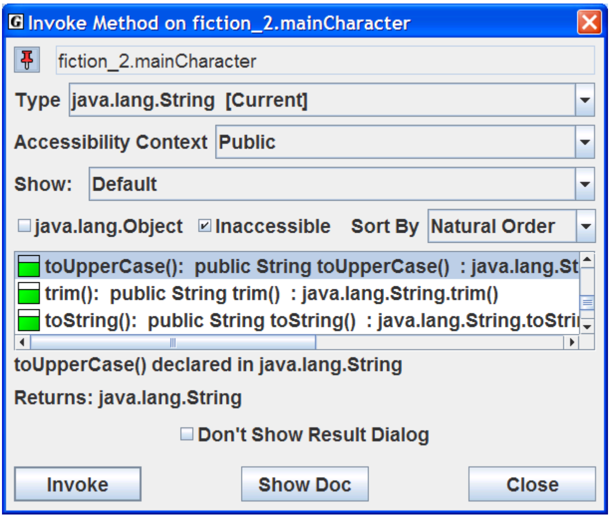
As indicated above, perhaps one of the most compelling reasons for using the workbench approach is that it allows the user to create an object and invoke each of its methods in isolation. Thus, with an instance of Fiction on the workbench, each of its four methods: getMainCharacter(), setMainCharacter(),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

3.12 Invoking Methods with Parameters That Are Objects

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

3.13 Invoking Methods on Object Fields

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



Invoke Method on fiction_2.mainCharacter

fiction_2.mainCharacter

Type java.lang.String [Current]

Accessibility Context Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

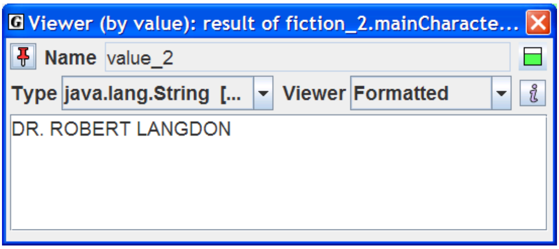
🟩 toUpperCase(): public String toUpperCase() : java.lang.St
🟩 trim(): public String trim() : java.lang.String.trim()
🟩 toString(): public String toString() : java.lang.String.toStri

toUpperCase() declared in java.lang.String

Returns: java.lang.String

□ Don't Show Result Dialog

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



Viewer (by value): result of fiction_2.mainCharacte...

Name value_2

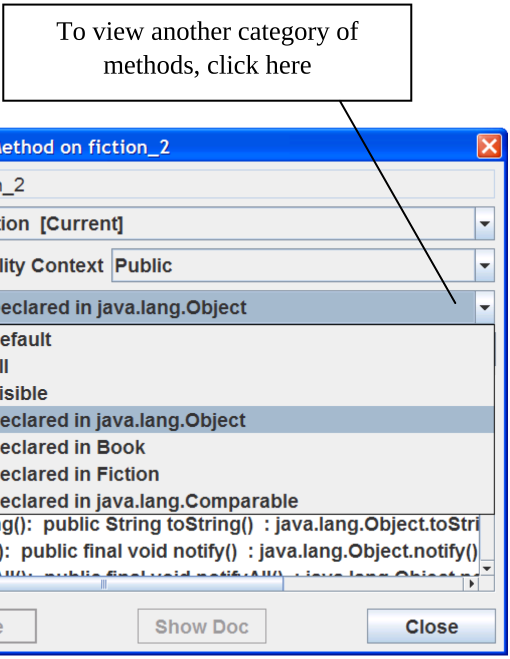
Type java.lang.String [... Viewer Formatted

DR. ROBERT LANGDON

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

3.14 Showing Categories of Methods

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



To view another category of methods, click here

Invoke Method on fiction_2

fiction_2

Type Fiction [Current]

Accessibility Context Public

Show: Declared in java.lang.Object

Default
All
Visible
Declared in java.lang.Object
Declared in Book
Declared in Fiction
Declared in java.lang.Comparable

□java.la

regis

getC

hash

equa

clon

toString(): public String toString() : java.lang.Object.toStri

notify(): public final void notify() : java.lang.Object.notify()

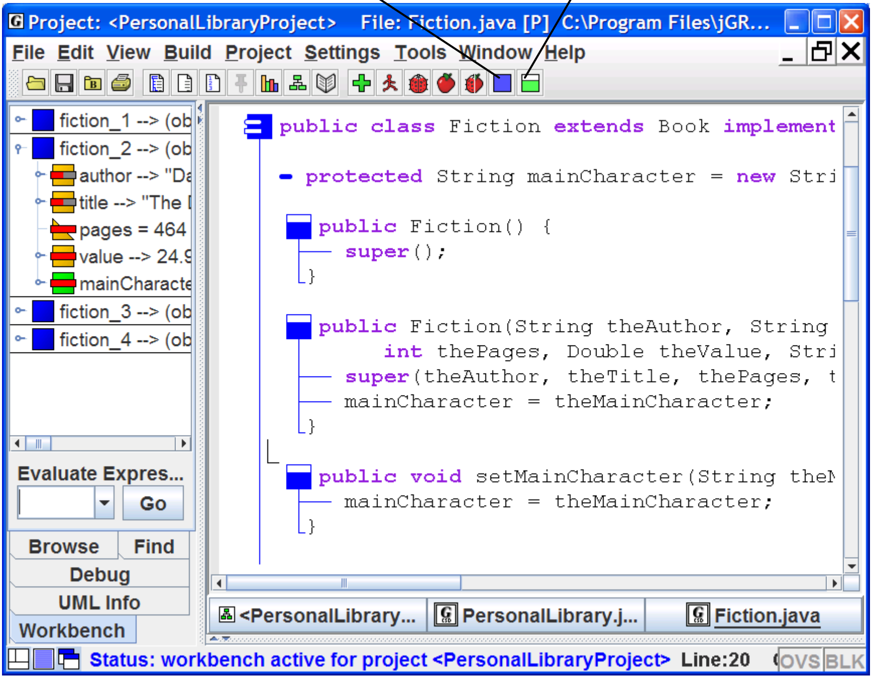
Figure 3-19. Showing methods declared in java.lang.Object

3.15 Creating Objects from the CSD Window

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

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

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



```
public class Fiction extends Book implement

   protected String mainCharacter = new Stri

   public Fiction() {
      super();
   }

   public Fiction(String theAuthor, String
         int thePages, Double theValue, Stri
      super(theAuthor, theTitle, thePages, t
      mainCharacter = theMainCharacter;
   }

   public void setMainCharacter(String theM
      mainCharacter = theMainCharacter;
   }
```

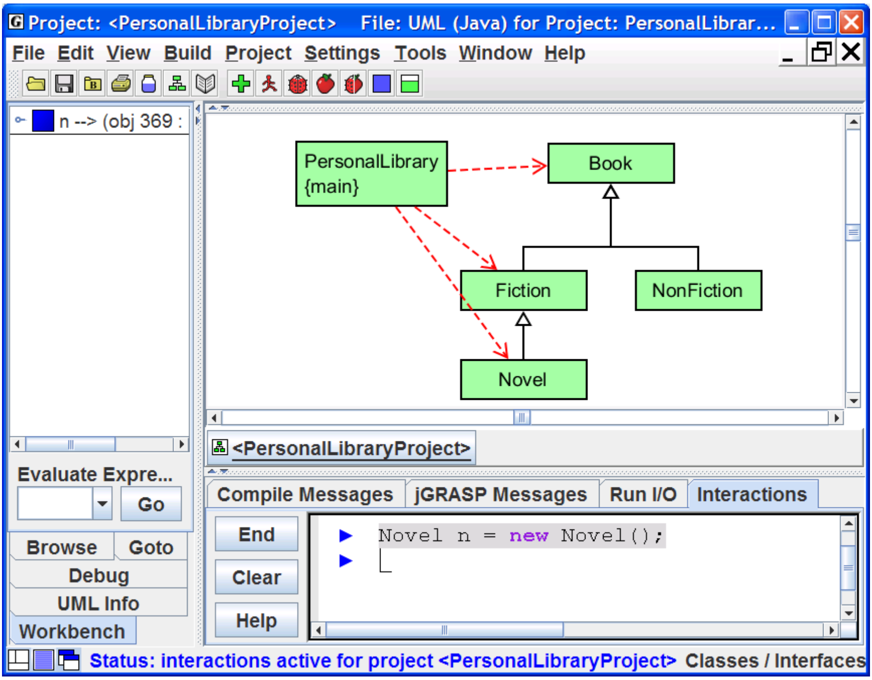
Figure 3-20. Creating an Instance from the CSD Window

3.16 Using Interactions

The **Interactions** tab, located next to the **Run I/O** tab in the lower window of the desktop, allows you to enter most Java statements and expressions and then execute or evaluate them immediately when you press ENTER. Interactions provide a convenient interface for working with items in the workbench or debug tabs. In fact, when you enter code that creates an object or primitive, the item is placed on the workbench where it can be inspected by unfolding and/or opening a viewer on it. Interactions can be especially helpful when learning and experimenting with objects and other elements in the Java language.

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

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

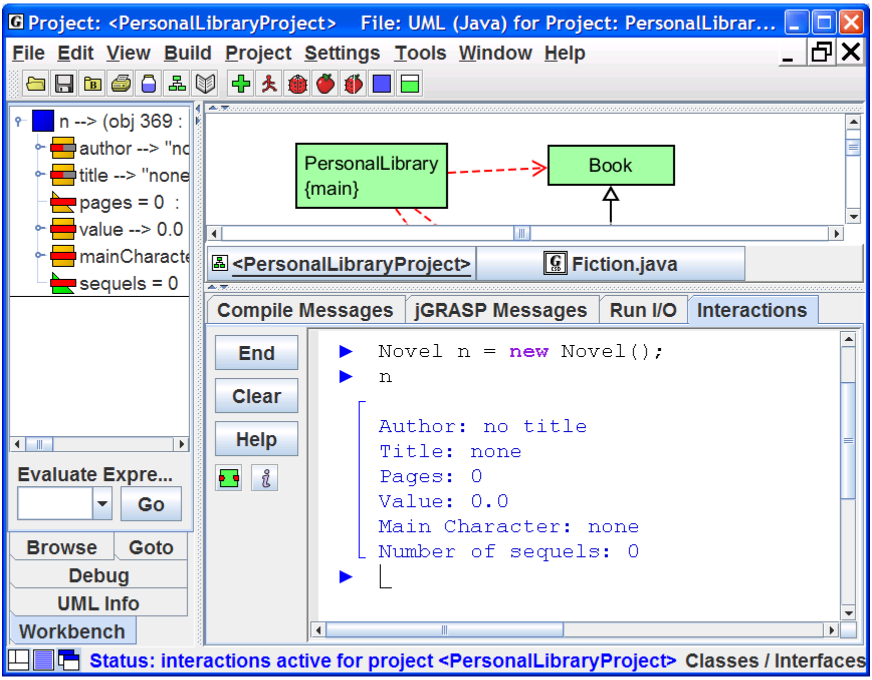


```
▶ Novel n = new Novel();
▶ 
```

Status: interactions active for project <PersonalLibraryProject>

Figure 3-21. Using Iteractions

With n on the workbench, we can now type statements or expressions that reference n and have them executed or evaluated immediately when ENTER is pressed. For example, typing n (followed by ENTER) is an expression that evaluates to the value of n, which is the Novel that was just created. For object values, the result of invoking toString() on the object is displayed, as shown in Figure 3-22.



```
▶ Novel n = new Novel();
▶ n
  Author: no title
  Title: none
  Pages: 0
  Value: 0.0
  Main Character: none
  Number of sequels: 0
▶
```

Status: interactions active for project <PersonalLibraryProject> Classes / Interfaces

Figure 3-22. Entering and evaluating the expression n

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

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

```
System.out.println                   Shift-ENTER
   ("The current value of n:" + n);     ENTER
```

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

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

3.17 Running the Debugger on Invoked Methods

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

3.18 Creating an Instance from the Java Class Libraries

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

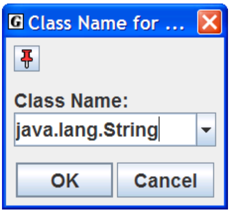
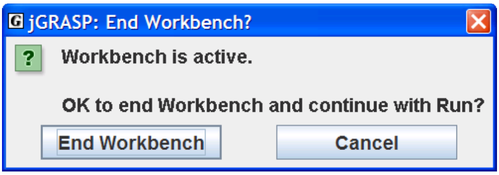


Figure 3-23. Creating an instance of String

3.19 Exiting the Workbench

The workbench is *running* whenever you have objects on it or if you have invoked main() directly from the class diagram. If you attempt to do an operation that conflicts with workbench, such as compiling a class, jGRASP will prompt you with a message indicating that the workbench is active and ask you if it is OK to end the Workbench (see Figure 3-24). The prompt is to let you know that the operation you are about to perform will clear the workbench. You can also clear or exit the workbench by right-clicking in the Workbench tab pane and selecting **Clear/Exit Workbench**.



Workbench is active.

OK to end Workbench and continue with Run?

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

3.20 Closing a Project

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

Here are two ways to close a project:

(1) From the Desktop menu – Click **Project > Close** or **Close All Projects**.

(2) In the Open Projects section of the **Browse** tab – Right-click on the project name and select **Close** or **Close All Projects**.

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

3.21 Exiting jGRASP

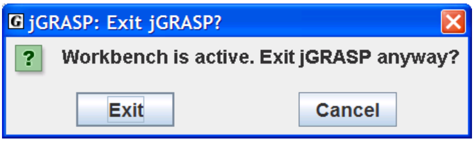
When you have completed your session with jGRASP, you should “exit” (or close) jGRASP rather than leaving it open for Windows to close when you log out or shut down your computer. When you exit jGRASP, it saves its current state and closes all open files. If a file was edited during the session, it prompts you to save or discard the changes. The next time you start jGRASP, it will open your files, and you will be ready to begin where you left off.

Close jGRASP in either of the following ways:

(1) Click the Close button ☒ in the upper right corner of the desktop; or

(2) On the File menu, click **File > Exit jGRASP**.

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



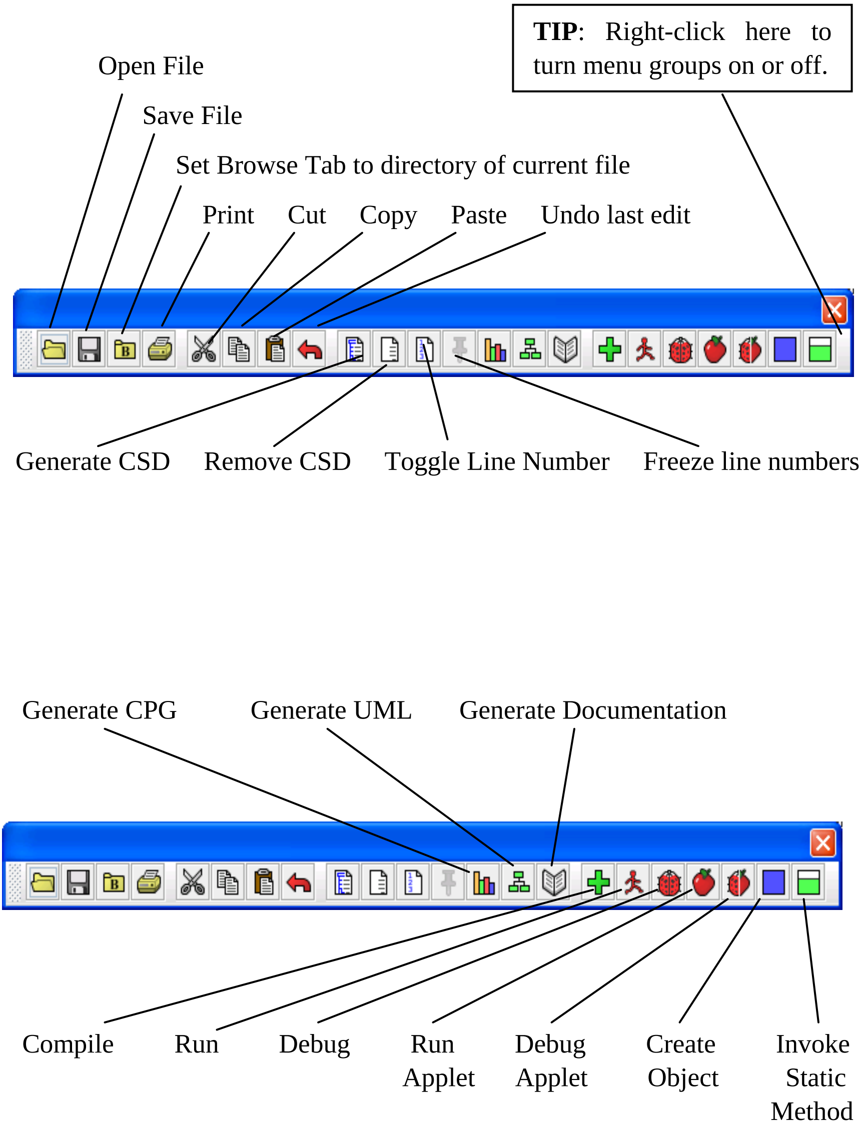
jGRASP: Exit jGRASP?

Workbench is active. Exit jGRASP anyway?

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

3.22 Review of Toolbar Buttons

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



TIP: Right-click here to turn menu groups on or off.

Figure 3-26. Toolbar

3.23 Exercises

(1) Create a new project (**Project** > **New**) named PersonalLibraryProject2 in the same directory folder as the original PersonalLibraryProject. During the create step, add the file Book.java to the new project.

  a. After the new project is created, add the other Java files in the directory to the project. Do this by dragging each file from the Files section of the Browse tab and dropping it in PersonalLibraryProject2 in the open projects section.

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

(2) Generate the documentation 📖 for PersonalLibraryProject2, using the default name for the documentation folder. After the Documentation Viewer pops up:

  a. Click the Fiction class link in the API (left side).

  b. Click the Methods link to view the methods for the Fiction class.

  c. Visit the other classes in the documentation for the project.

(3) Close the project.

(4) Open the project by double-clicking on the project file in the files section of the Browse tab.

(5) Generate the UML class diagram for the project.

  a. Display the class information for each class.

  b. Display the dependency information between two classes by selecting the appropriate arrow.

  c. Compile ✚ and run 🏃 the program using the buttons on the toolbar.

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

  e. Create three instances of Fiction from the class diagram. Open Novel in a CSD window, then create two instances of Novel from the CSD window

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

  g. Open an object viewer for one or more String fields of one of the instances.

(6) Use Interactions to enter statements and expressions that reference items on the workbench. Create new objects by entering statements such as:

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

(7) Open the CSD window for PersonalLibrary.java.

  a. Set a breakpoint on the first executable statement.

  b. From the UML window, start the debugger by clicking the Debug button.

  c. Step through the program, watching the objects appear in the Debug tab as they are created.

  d. Restart the debugger. This time click “step in” instead of “step”. This should take you into the constructors, etc.

(8) If you have other Java programs available, repeat the steps above for each program.

Notes

4 Interactions

The **Interactions** feature in jGRASP allows the user to enter Java statements and expressions and then execute/evaluate them immediately. This feature is not meant to be a replacement for the traditional edit-compile-run cycle, but rather a convenient way to experiment with Java statements and expressions. The Interactions feature is relevant for beginning as well as advanced users who are programming in Java. The feature was introduced briefly in *Getting Started* and *Getting Started with Objects*. In this tutorial, we provide a more complete description with detailed examples. If you are not familiar with the basic features of jGRASP (e.g., compiling, running, and debugging), you are encouraged to read *Getting Started*.

Objectives – When you have completed this tutorial, you should be able to use Interactions with the Object Workbench, Debugger, and Viewers in jGRASP. You should be able to declare primitive variables, assign values, and use them in expressions. You should be able declare reference variables, create instances of objects, invoke methods on the objects, and use the reference variables in expressions. You should be able use interactions containing variables from the workbench and debugger. You should be able to copy interactions and paste them to a CSD window as source code.

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

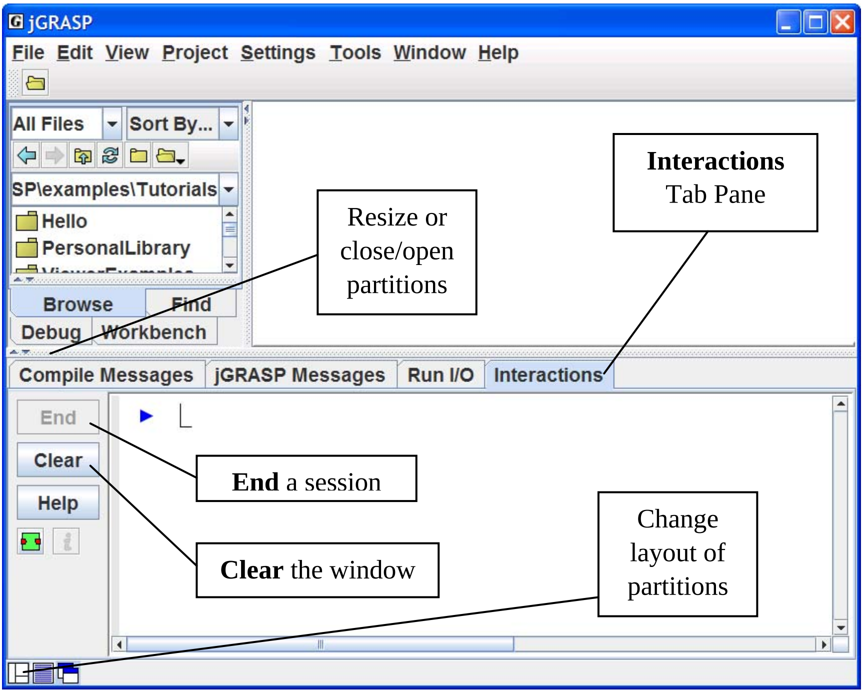
4.1 Starting Interactions
4.2 Interactions with Primitives
4.3 Interactions with Reference Types
4.4 Interactions with Your Own Classes
4.5 Working with Reference Types – Important Details
4.6 Interactions with the Debugger

4.1 Starting Interactions

jGRASP

Let’s begin by starting jGRASP and then closing any files that had been left open from the previous session. We need to select the Interactions tab in the lower window of the jGRASP desktop as shown in Figure 4-1. As indicated in the figure, an Interactions session can be terminated by clicking the **End** button, the window can be cleared by clicking he **Clear** button, and the window containing the Interactions tab can be resized by dragging on the partitions or by clicking on the up/down arrows at the top or left end of each of the partitions. Many users find it helpful to switch between a full-width message pane across the bottom of the desktop and a full-height tab pane on the left. The button (⊟ or ⊞) in the lower left corner of the desktop provides convenient way to change the layout of the partitions.

The blue triangle in the Interactions window indicates where we should enter our first interaction. Click in the window to gain focus, and we are ready to begin.



Interactions Tab Pane

Resize or close/open partitions

End a session

Clear the window

Change layout of partitions

Figure 4-1. The jGRASP Virtual Desktop

4.2 Interactions with Primitives

Our first interactions will explore Java primitive types. Let’s begin by declaring an integer variable `i` and assigning it an initial value of 10. After entering the following statement, press ENTER.

```
int i = 10;
```

As soon as you press ENTER, the Interactions session will be started, and the variable `i` should appear on the Workbench.

Now enter the code to declare a variable x of type double:

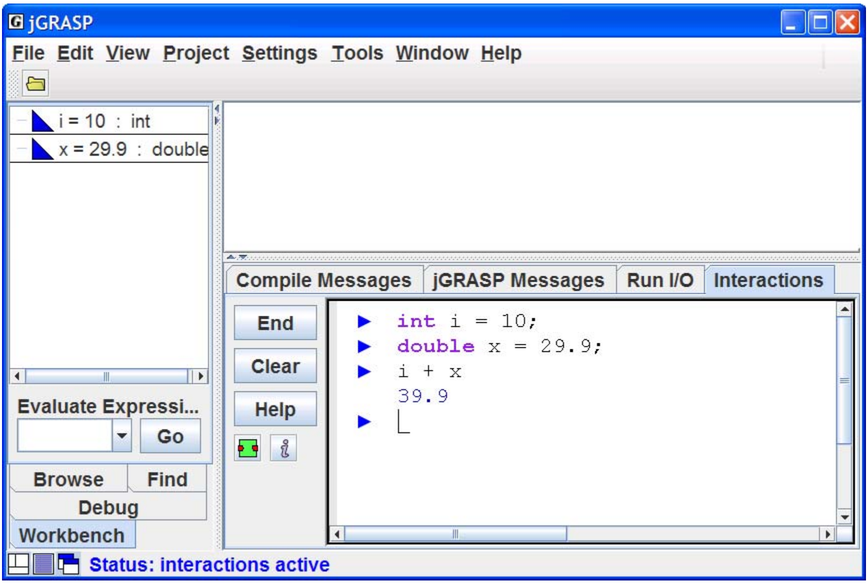
```
double x = 29.9;
```

After pressing ENTER, `x` should appear on the Workbench.

Now let’s enter an expression that uses the two variables `i` and `x`. Note that an expression does not end with a semi-colon (;)

```
i + x
```

As soon as ENTER is pressed the expression will be evaluated, and we should see 29.9 displayed below the expression. Figure 4-2 shows the desktop after the interactions above have been entered.



i = 10 : int

x = 29.9 : double

```
▶ int i = 10;
▶ double x = 29.9;
▶ i + x
  39.9
▶
```

Status: interactions active

Figure 4-2. Our first interactions

Now let's try a few more interactions that use `i` and `x`.

```
i = 10;

i = i + 10;

x = x + 3.5;
```

As you enter each of these, be sure to observe the changes to the variables on the Workbench.

Errors – If a statement contains an error, a message similar to a compiler error message will be displayed.

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

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

```
System.out.println   Shift-ENTER
 ("i = " + i + " and x = " + x);   ENTER
```

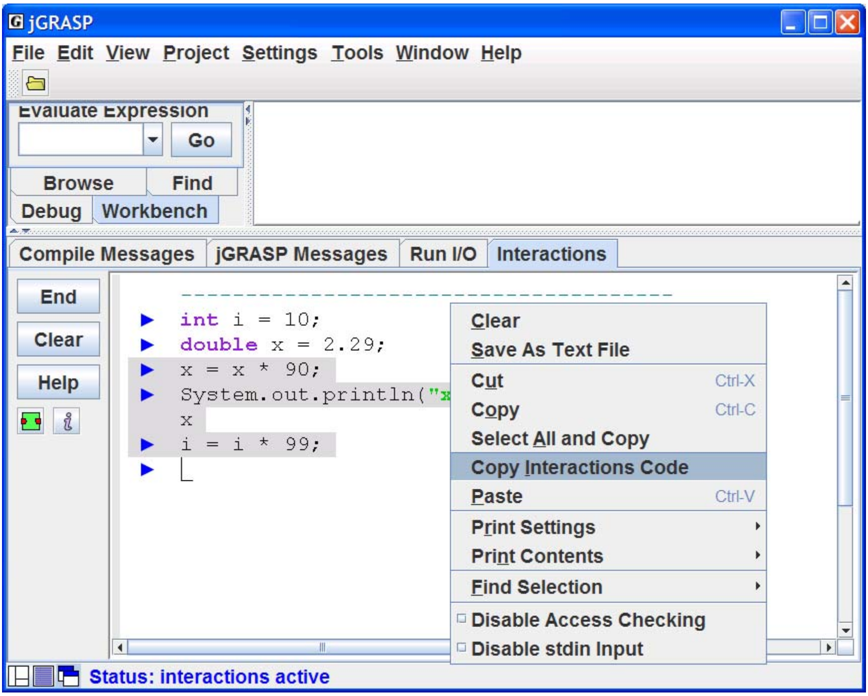
If you simply press ENTER at the end of the first line, Interactions will attempt to execute the incomplete statement and you will get an error message. Below is the result you should see after the statements above are entered with delayed execution.

```
System.out.println
("i = " + i + " and x = " + x);
i = 10 and x = 22.89
```

Compound statements – When entering statements such as `if`, `if-else`, `while`, `for`, and block statements `{}`, execution is delayed until the "normal" end of the statement is reached. To enter the following while statement on two lines, you can press ENTER at the end of the each line (i.e., there is no need to press SHIFT-ENTER after the first line).

```
while (i > 0)
i = i - 1;
```

Copying Interactions – After you have entered one or more statements in Interactions, you may find it useful to copy and then paste them back into Interactions in order to execute them again or perhaps paste them into a CSD window to make them part of a program. To copy statements, first use the mouse to select the range of statements. Next, right-click the mouse to bring up the context menu and then select "Copy Interactions Code" as shown in Figure 4-3. When you do the "paste", it will not include the "x" that was output when `System.out.println("x");` was executed.

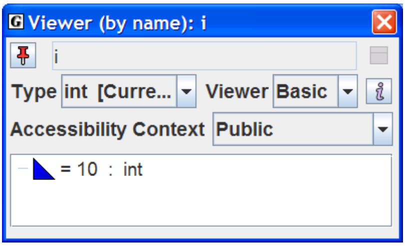


```
   ---------------------------------------
▶  int i = 10;
▶  double x = 2.29;
▶  x = x * 90;
▶  System.out.println("x
   x
▶  i = i * 99;
▶  
```

Figure 4-3. Selecting and copying interactions

Viewers – Now let's take a quick side trip and open viewers on `i` and `x` to explore their details. The easiest way to open a viewer on a variable is to simply drag it from the Workbench (i.e., left-click on the item and while holding down on the button, "drag" the item and release the mouse anywhere). Alternatively, you can open a viewer by right-clicking on the item and then selecting "View by Name."

Figures 4-4 and 4-5 show viewers for each of `i` and `x`. Note that **Viewer** is set to **Basic**. This is similar to the view in the Workbench.



Viewer (by name): i

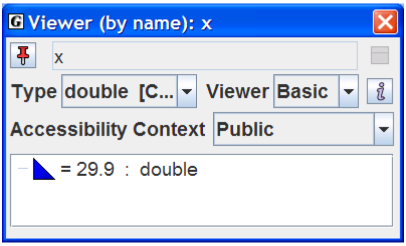
i

Type int [Curre... Viewer Basic

Accessibility Context Public

= 10 : int

Figure 4-4. Viewer (Basic) of `i`



Viewer (by name): x

x

Type double [C... Viewer Basic

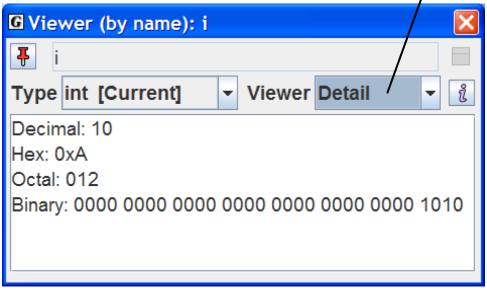
Accessibility Context Public

= 29.9 : double

Figure 4-5. Viewer (Basic) of `x`

Using the drop-down menu on the viewer, we can change the setting for **Viewer** from **Basic** to **Detail**. Figure 4-6 shows the **Detail** view for `i` with its value in decimal, hexadecimal, octal, and binary. If you change **Viewer** to **Detail** in the viewer for x, you will see the IEEE floating point representation (sign, exponent, and mantissa) for its value as well as the details for how the computation was done. See Figure 4-7.

Change **Viewer** from **Basic** to **Detail**



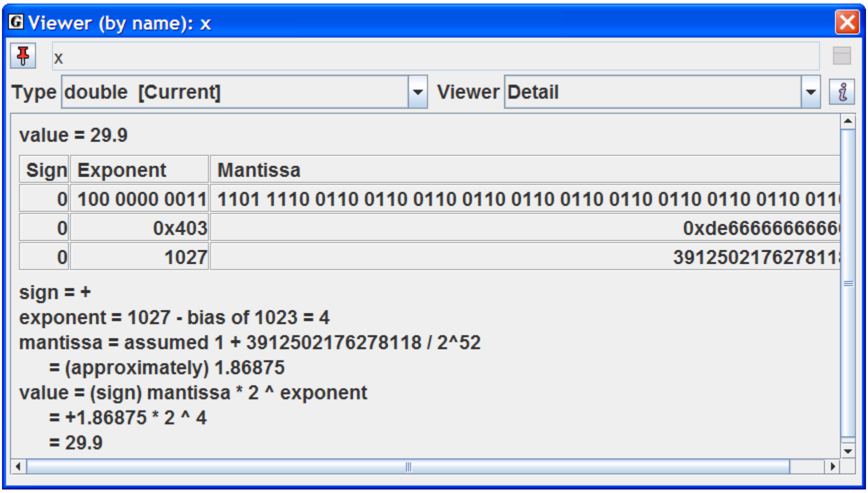
Viewer (by name): i

i

Type int [Current] Viewer Detail

Decimal: 10
Hex: 0xA
Octal: 012
Binary: 0000 0000 0000 0000 0000 0000 0000 1010

Figure 4-6. Viewer (Detail) of `i`



Type double [Current] Viewer Detail

value = 29.9

Sign	Exponent	Mantissa
0	100 0000 0011	1101 1110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 011
0	0x403	0xde666666666
0	1027	391250217627811

sign = +
exponent = 1027 - bias of 1023 = 4
mantissa = assumed 1 + 3912502176278118 / 2^52
    = (approximately) 1.86875
value = (sign) mantissa * 2 ^ exponent
    = +1.86875 * 2 ^ 4
    = 29.9

Figure 4-7. Viewer (Detail) of x

Exploring the increment operator – Many beginning programmers find the increment and decrement operators confusing. We finish up this section by taking a look at the two forms of the increment operator. Let's enter the following expressions and observe the result returned in Interactions versus the result shown on the Workbench or in the viewer.

```
++i

i++
```

The difference between these two expressions is significant. If you do not see a difference at first, enter each expression again (use UP arrow) and carefully observe the result in Interactions and the result on the Workbench. If you still do not see the difference, see the explanation below.

`++i` and `i++`
`++i` : the ++ <u>before</u> the `i` causes `i` to be incremented by 1 and the new value to be used in the expression. Thus, the value in Interactions will match the value on the Workbench. **`i++`** : the ++ <u>after</u> the `i` causes the current value of `i` to be used in the expression and then `i` is incremented by 1. Thus, the value in Interactions will be the old value, and the value on the Workbench will be the new incremented value.

4.3 Interactions with Reference Types

Now let's enter statements in Interactions that involve reference types and instances of objects and primitive types. We begin by entering a statement that declares a reference `s1` of type String and assigns a String literal to it.

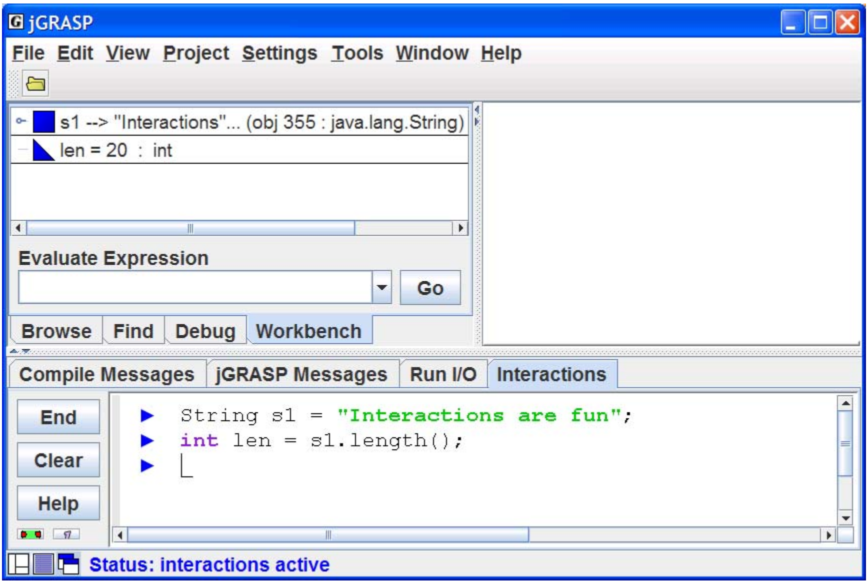
```
        String s1 = "Interactions are fun";
```

After ENTER is pressed, you should see an instance of String called s1 on the Workbench.

Now let's enter a statement that declares an integer variable `len` and sets its value by invoking the length() method on `s1`.

```
        int len = s1.length();
```

After ENTER is pressed, you should see `len` on the Workbench with a value of 20 as shown in Figure 4-9. Notice the difference in the notation used for the reference variable ■ `s1` versus the primitive variable ◣ `len`. We see that `s1` is "pointing to" an object of type String whereas `len` is an int whose value is simply "equal to" 20. This notation is intended to visually remind us that the underlying representations of primitive and reference variables are quite different.



```
■ s1 --> "Interactions"... (obj 355 : java.lang.String)
◣ len = 20  :  int
```

```
▶ String s1 = "Interactions are fun";
▶ int len = s1.length();
▶
```

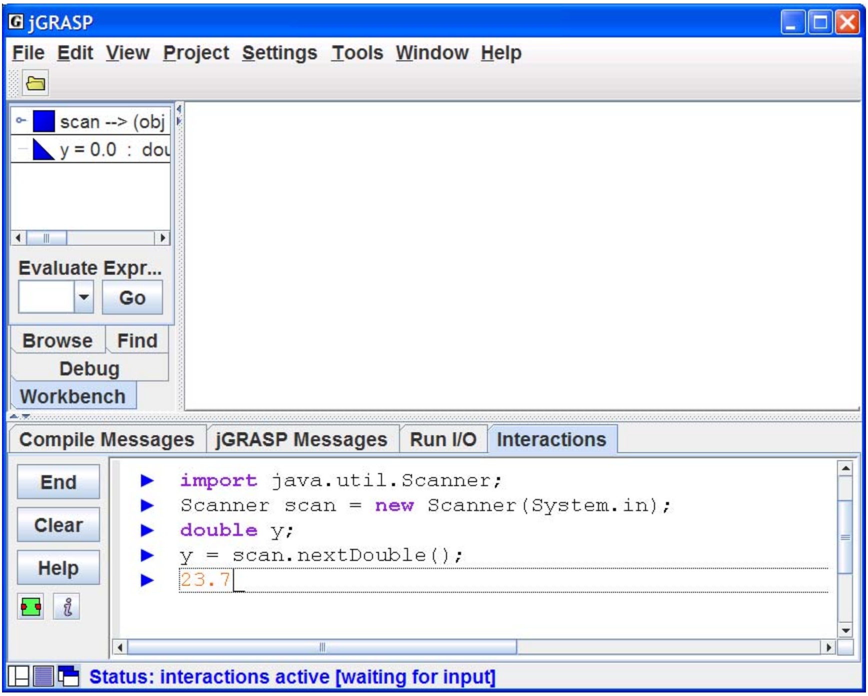
Status: interactions active

Figure 4-9. Interactions with results on the Workbench

Import statements in interactions work in the same way they do in a Java program. For example, to create an instance of the Scanner class, we could enter the following import statement at some point during the Interactions session prior to entering a statement that references the Scanner. Suppose we want to use the Scanner class to input a double and assign it to the variable y. Let’s enter the four statements below.

```
import java.util.Scanner;

Scanner scan = new Scanner(System.in);

double y;

y = scan.nextDouble()
```

When the last statement is entered and executed to read in a double, an input box is opened in Interactions to allow you to enter the value. Figure 4-10 shows the desktop after 23.7 has been entered in the input box but before ENTER has been pressed. When ENTER is pressed, the input box will disappear, and `y` will be updated on the Workbench.



```
▶ import java.util.Scanner;
▶ Scanner scan = new Scanner(System.in);
▶ double y;
▶ y = scan.nextDouble();
▶ 23.7
```

Status: interactions active [waiting for input]

Figure 4-10. Interactions to input and assign a double

4.4 Interactions with Your Own Classes

If you want to reference one or more of your own classes in Interactions, the classes need to be visible from Interactions. The easiest way to accomplish this is to open the file containing the class. If you start Interactions while the file has focus (assuming it has been compiled), this class as well as others in the same directory will be available to Interactions. Your file has focus if it is underlined in the Browse tab and/or on the window bar. The name of the file in focus will also be displayed in the title of the jGRASP desktop. If your class is not recognized in Interactions, click the END button and try it again, making sure your file has focus.

4.5 Working with Reference Types – Important Details

Performing interactions with reference types and instances of objects is similar to working with primitives. That is, after you enter a statement or expression, it is executed/evaluated when you press ENTER. The only significant difference is that while primitives are always available, you must ensure that any class which you are referencing is available to Interactions.

If you want to reference one of your own classes that you have opened in a CSD window, you should start Interactions after the file as been opened and while it has focus. Your file has focus if it is underlined in Browse tab and on the window bar, and it is displayed in the title of the jGRASP desktop. If Interactions does not recognize your class, click the END button and try it again, making sure your file has focus. When you start Interactions, all classes in the same directory as the file with focus will also be available to Interactions.

4.6 Interactions with the Debugger

When variables are declared in Interactions they are placed on the Workbench as seen in the examples above. You can also interact with variables in the Debug tab. When you run your program in debug mode and the program stops at a breakpoint, the Debug tab will contain the variables that have been declared and initialized. You can enter statements and expressions in Interactions that use these variables. That is, the variables in the Debug tab are available to Interactions. You may find this useful in debugging. For example, to find the length of the 10,000th element in an array of Strings named *stringArray*, you could simply enter stringArray[10000].length() in Interactions.

5 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 is easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and class diagrams.

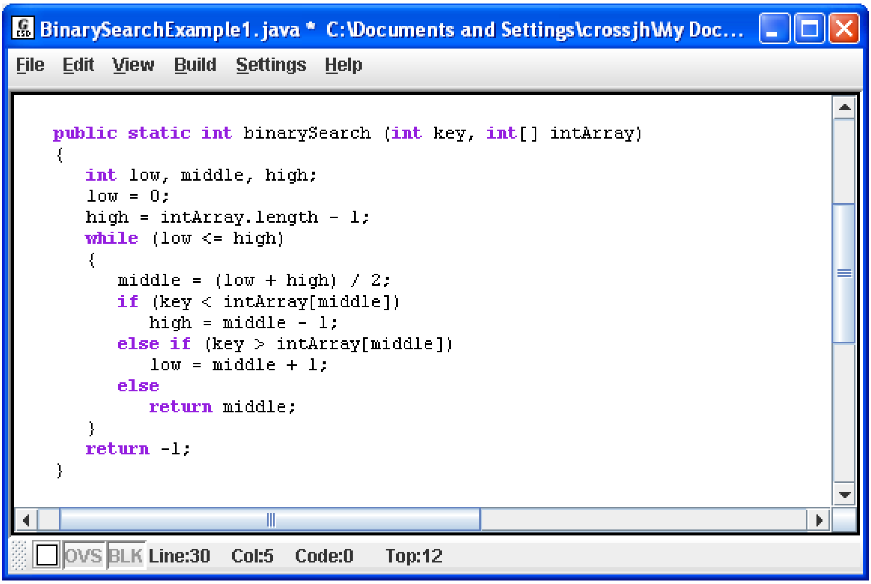
Objectives – When you have completed this tutorial, you should be able to use and understand the graphical notations used in the **CSD** for basic control constructs of modern programming languages, including sequence, selection, iteration, exits, and exception handling.

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

5.1 An Example to Illustrate the CSD
5.2 CSD Program Components/Units
5.3 CSD Control Constructs
5.4 CSD Templates
5.5 Hints on Working with the CSD
5.6 Reading Source Code with the CSD
5.7 References

5.1 An Example to Illustrate the CSD

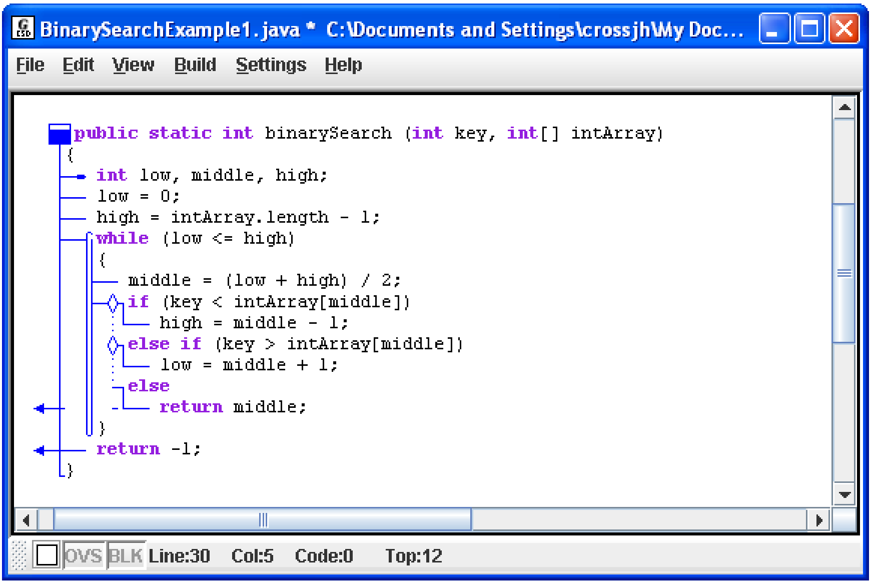
Figure 5-1 shows the source code for a Java method called binarySearch. The method implements a binary search algorithm by using a *while* loop with an *if..else..if* statement nested within the loop. Even though this is a simple method, displayed with colored keywords and traditional indentation, its readability can be improved by adding the CSD. In addition to the *while* and *if* statements, we see that the method includes the declaration of primitive data (int) and two points of exit. The CSD provides visual cues for each of these constructs.


BinarySearchExample1.java * C:\Documents and Settings\crossjh\My Doc...
File
Edit
View
Build
Settings
Help
OVS
BLK
Line:30
Col:5
Code:0
Top:12

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

Figure 5-1. binarySearch method without CSD

Figure 5-2 shows the binarySearch method after the CSD has been generated. Although all necessary control information is in the source text, the CSD provides additional visual cues 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 branching off the stem as it extends downward. The declaration of primitive data is highlighted with the symbol appended to the statement stem. The CSD construct for the *while* statement is represented by the double line "loop" (with break at the top), and the *if* statement uses the familiar diamond symbol from traditional flowcharts. Finally, the two ways to exit from this method are shown explicitly with an arrow drawn from inside the method through the method stem to the outside.


BinarySearchExample1.java * C:\Documents and Settings\crossjh\My Doc...
File
Edit
View
Build
Settings
Help
OVS
BLK
Line:30
Col:5
Code:0
Top:12

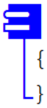
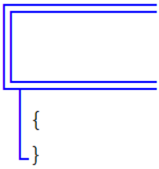
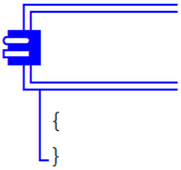
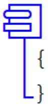
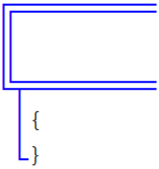
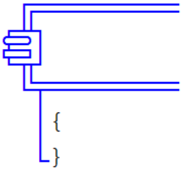
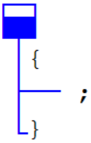
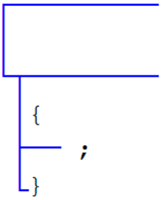
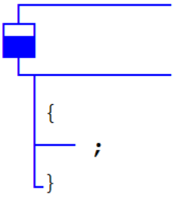
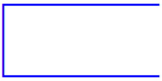
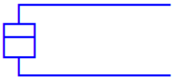
```
public static int binarySearch (int key, int[] intArray)
{
   int low, middle, high;
   low = 0;
   high = intArray.length - 1;
   while (low <= high)
   {
      middle = (low + high) / 2;
      if (key < intArray[middle])
         high = middle - 1;
      else if (key > intArray[middle])
      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

Figure 5-2. binarySearch with CSD

While this is a small piece of code, it does illustrate the basic CSD constructs. However, the true utility of the CSD can be realized best when reading or writing larger, more complex programs, especially when control constructs become deeply nested. A number of studies involving the CSD have been done and others are in progress. In one of these, the 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 to provide significant advantages in numerous code reading activities [Hendrix 2002]. In the following sections, the CSD notation is described in more detail.

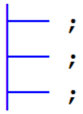
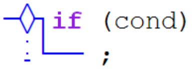
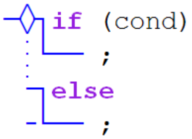
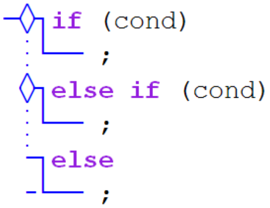
5.2 CSD Program Components/Units

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			

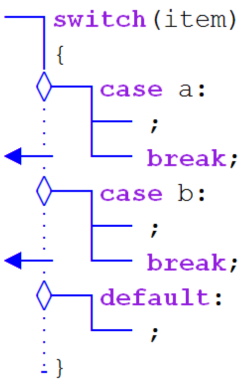
5.3 CSD Control Constructs

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. The semi-colons in the examples are placeholders for statements in the language.

Sequence	 ; ; ;	Sequential flow is represented in the CSD by a vertical stem with a small horizontal stem for each individual statement on a particular level of control.
Selection **if**	 if (cond) ;	For selection statements, the True/False condition itself is marked with a small diamond, just as in a flow chart. The statements to be executed if the condition is **true** are marked by a solid line leading from the right of the decision diamond.
if..else	 if (cond) ; else ;	The control path for a **false** condition is marked with a dotted line leading from the bottom of the diamond to another decision diamond, an else clause, a default clause, or the end of the decision statement.
if..else..if	 if (cond) ; else if (cond) ; else ;	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.

Selection (cont'd)

switch

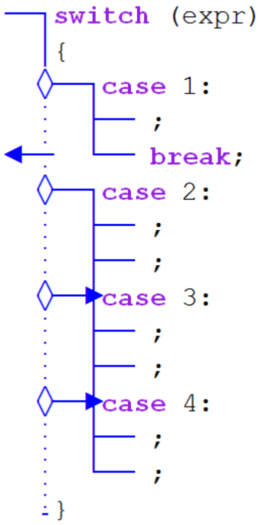


```
switch(item)
{
   case a:
      ;
      break;
   case b:
      ;
      break;
   default:
      ;
}
```

The semantics of the ***switch*** statement are different from those of ***if*** statements. The *expression* (of integral or enum type) 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.

switch

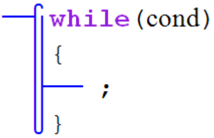
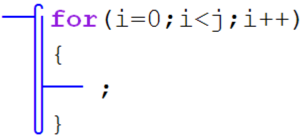
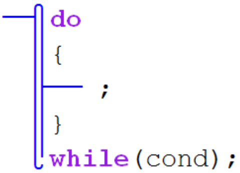
when break is omitted

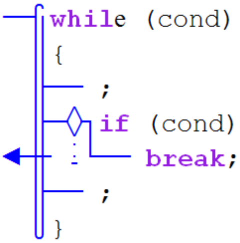
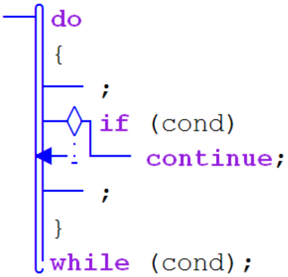


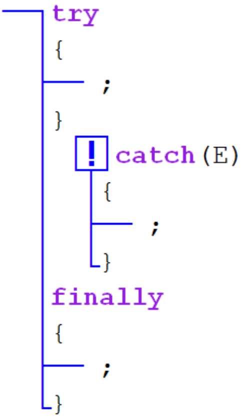
```
switch (expr)
{
   case 1:
      ;
      break;
   case 2:
      ;
      ;
   case 3:
      ;
      ;
   case 4:
      ;
      ;
}
```

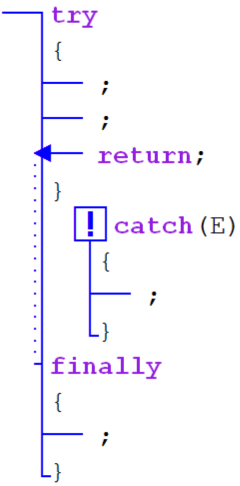
When the break statement is omitted from end of the sequence within a case, control falls through to the next case. In the example at left, case 1 has a ***break*** statement at the end of its sequence, which will pass control to the end of the switch (as indicated by the arrow). However, case 2, case 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.

Iteration
while loop
(pre-test)
for loop
(discrete)
do loop
(post-test)
	
	while (cond)
	{
	;
	}
	
	for (i=0;i<j;i++)
	{
	;
	}
	
	do
	{
	;
	}
	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 the boolean condition. The sequence within the ***while*** will be executed zero or more times.
		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 ***for*** will be executed zero or more times.
		The ***do*** statement is similar to the while except that the loop condition is at the end of the loop instead of the beginning. Thus, the body of the loop is guaranteed to execute at least once.

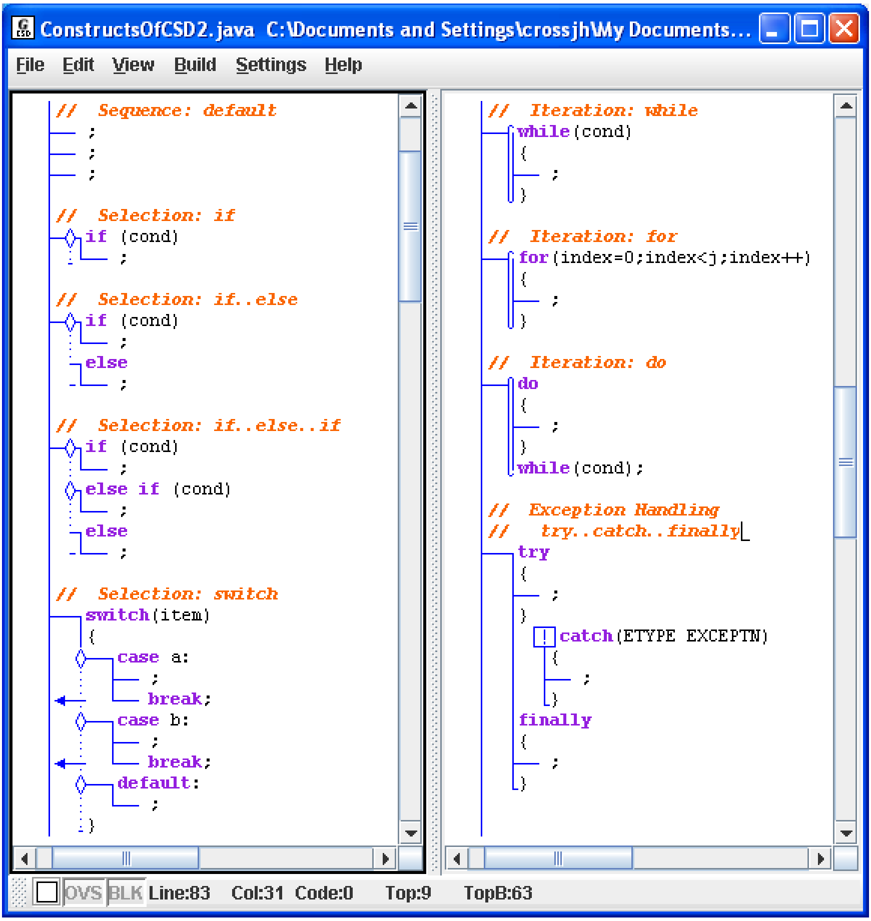
break in loop
 	
Iteration (cont'd)
 	
continue
	
	while (cond)
	{
	;
	if (cond)
	break;
	;
	}
	
	
	do
	{
	;
	if (cond)
	continue;
	;
	}
	while (cond);
		The ***break*** statement can be used to transfer control flow out of any loop (***while***, ***for***, ***do***) body, as indicated by the arrow, 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.
		
		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.

With **a return**	
	try
	{
	;
	;
	return;
	}
	catch (E)
	{
	;
	}
	finally
	{
	;
	}
		The ***try..catch*** statement can have multiple ***catch*** clauses, one for each exception to be handled.
		By definition, the ***finally*** clause is always executed no 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 be 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.

5.4 CSD Templates

In Figure 5-3, the basic CSD control constructs, described above, are shown in the CSD window. These are generated automatically based on the text in the window. In addition to being typed or read from a file, the text can be inserted from a list of templates by selecting **Templates** on the CSD window tool bar.


ConstructsOfCSD2.java C:\Documents and Settings\crossjh\My Documents...
File
Edit
View
Build
Settings
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 5-3. CSD Control Constructs generated in CSD Window

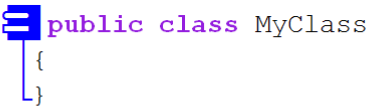
5.5 Hints on Working with the CSD

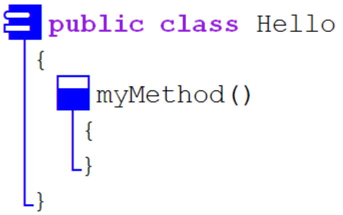
The CSD is generated based on the source code text in the CSD window. When you click **View** > **Generate CSD** (or press **F2**), jGRASP parses the source code based on a grammar or syntax that is slightly more forgiving than the Java compiler. If your program will compile successfully, the CSD should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

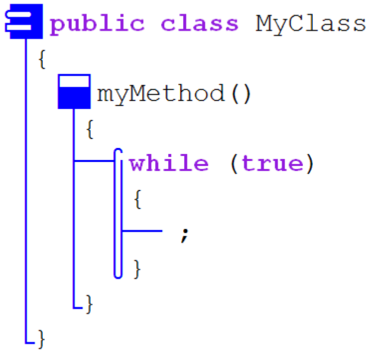
Enter code in syntactically correct chunks - To reap the most benefit from using the CSD when entering a program, you should take care to enter code in syntactically correct chunks, and then regenerate the CSD often. If an error is reported, it should be fixed before you move on. If the error message 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. After the code is entered in each step, click the Generate CSD button or press F2 to generate the CSD.

Step	Code to Enter	After CSD is generated

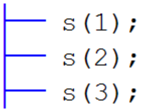
1	public class MyClass
	{
	}
		
		public class MyClass
		{
		}

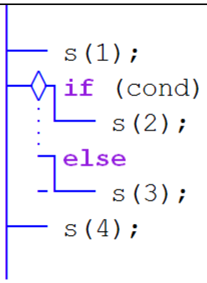
2	public class MyClass
	{
	   myMethod()
	   {
	   }
	}
		
		public class Hello
		{
		myMethod()
		{
		}
		}

3	`public class MyClass`
	`{`
	`    myMethod()`
	`    {`
	`       while (true)`
	`       {`
	`             ;`
	`       }`
	`    }`
	`}`
		
		`public class MyClass`
		`{`
		`   myMethod()`
		`   {`
		`      while (true)`
		`      {`
		`          ;`
		`      }`
		`   }`
		`}`

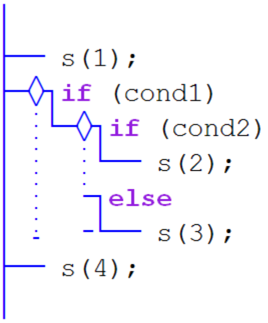
5.6 Reading Source Code with the CSD

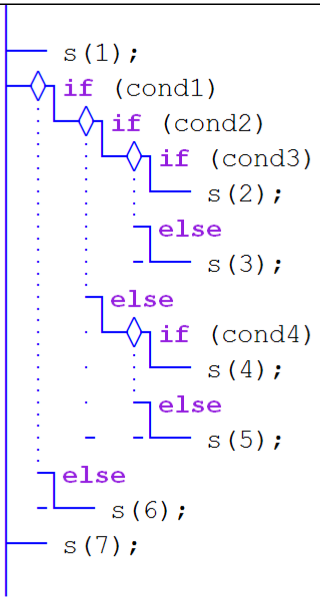
The CSD notation for each of the control constructs has been carefully designed to aid in reading and scanning source code. While the notation is meant to be intuitive, there are several reading strategies worth pointing out, especially for deeply nested code.

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.
	
	`s(1);`
	`s(2);`
	`s(3);`

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
	
	`s(1);`
	`if (cond)`
	`   s(2);`
	`else`
	`   s(3);`
	`s(4);`

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).	

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 the next statement on the stem which is s(4). If 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 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.

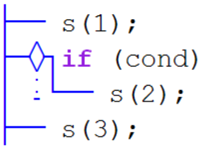
```
s(1);
if (cond1)
   if (cond2)
      if (cond3)
         s(2);
      else
         s(3);
   else
      if (cond4)
         s(4);
      else
         s(5);
else
   s(6);
s(7);
```

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.

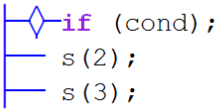


```
s(1);
if (cond)
   s(2);
s(3);
```

Here is another common mistake made obvious by the CSD.

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

The semi-colon after the condition was almost certainly unintended. However, the CSD shows what is there rather than what was intended.



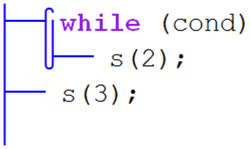
```
if (cond);
s(2);
s(3);
```

Similarly, the CSD provides the correct interpretation of the *while* statement.

```
while (cond)
   s(2);
   s(3);
```

Missing braces . . .

.

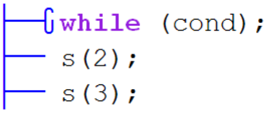


```
while (cond)
   s(2);
s(3);
```

Similarly, the CSD provides the correct interpretation of the *while* statement.

```
while (cond);
   s(2);
   s(3);
```

Unintended semi-colon . . .



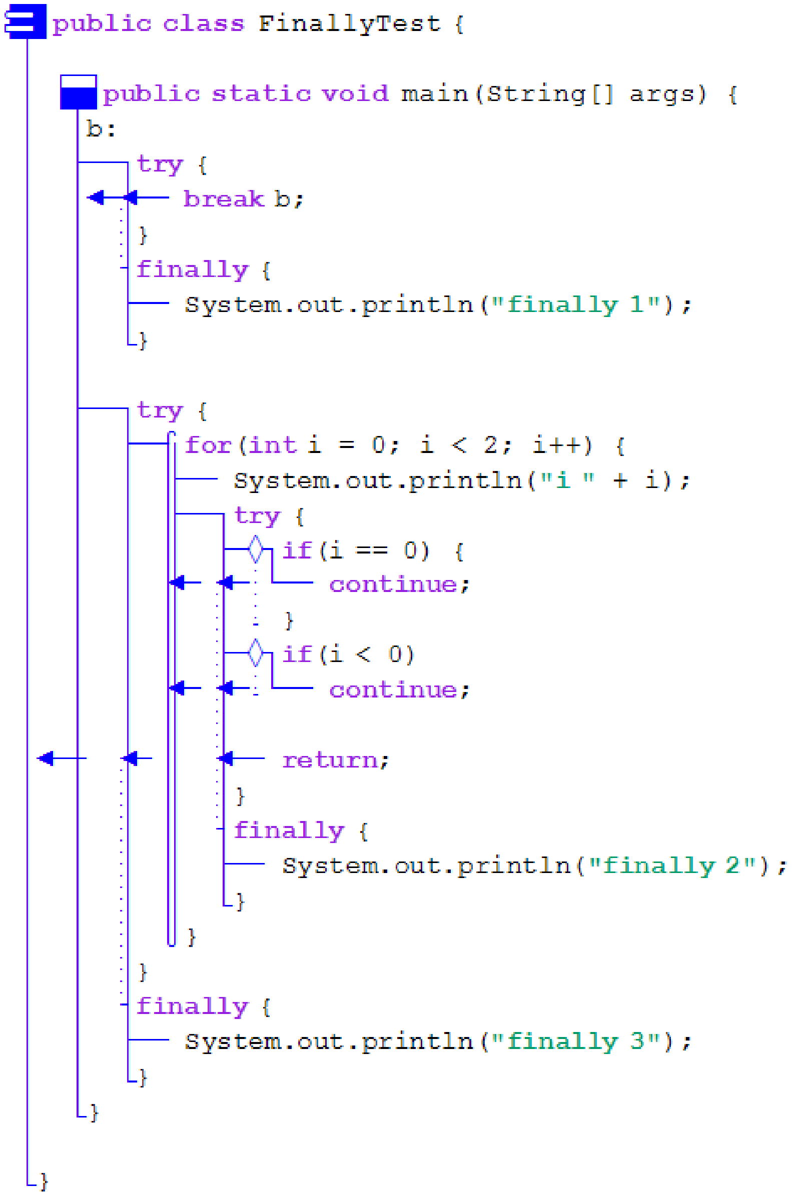
```
while (cond);
s(2);
s(3);
```

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.

Try-Finally **with break, continue, and return statements with no CSD**

```
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");
      }
   }

};
```

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



```
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. 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
```

5.7 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.

6 The Integrated Debugger

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

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

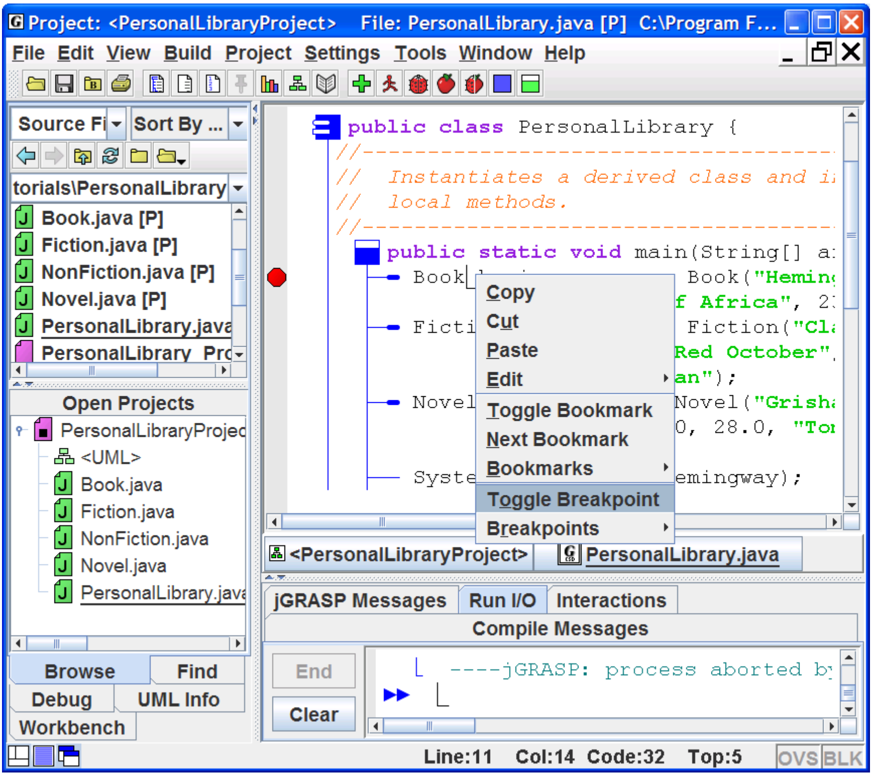
Objectives – When you have completed this tutorial, you should be able to set breakpoints and step through the program, either by single stepping or auto stepping. You should also be able to display the dynamic state of objects created by the program using the appropriate Object Viewer.

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

6.1 Preparing to Run the Debugger
6.2 Setting a Breakpoint
6.3 Running a Program in Debug Mode
6.4 Stepping Through a Program – the Debug Buttons
6.5 Stepping Through a Program – without *Stepping In*
6.6 Stepping Through a Program – and *Stepping In*
6.7 Opening Object Viewers
6.8 Debugging a Program

6.1 Preparing to Run the Debugger

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



```
public class PersonalLibrary {
//-----------------------------------
//  Instantiates a derived class and i
//  local methods.
//-----------------------------------
   public static void main(String[] a
      Book                      Book("Heming
                               f Africa", 2
      Ficti                     Fiction("Cla
                               Red October",
                               an");
      Novel                     Novel("Grisha
                               0, 28.0,  "Tor
      Syste                     emingway);
```

```
----jGRASP: process aborted by
```

Figure 6-1. Setting a breakpoint

6.2 Setting a Breakpoint

In order to examine the state of your program at a particular statement, you need to set a breakpoint. The statement you select must be “executable” rather than a simple declaration. To set a breakpoint in a program, move the mouse to the line of code and left-click the mouse to move the cursor there. Now right-click on the line to display a set of options that includes **Toggle Breakpoint**. For example, in Figure 6-1 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. You can remove a breakpoint by clicking on the red octagon. This second approach is the one most commonly used for setting and removing breakpoints.

6.3 Running a Program in Debug Mode

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

(1) Click **Build** – **Debug** on the CSD window menu, or

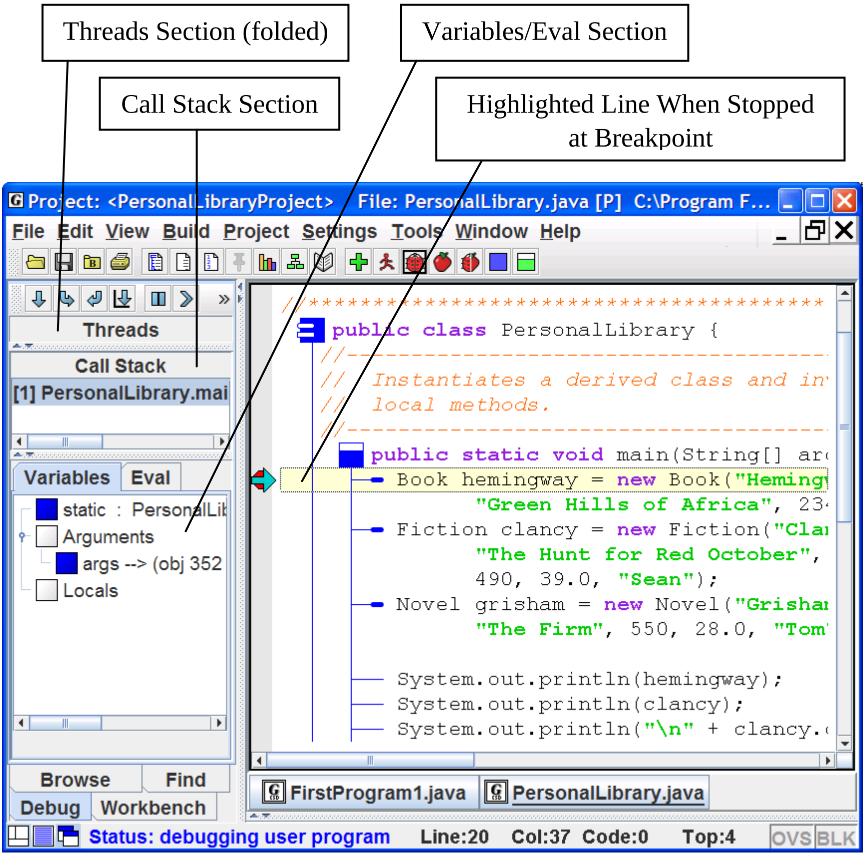
(2) Click the Debug button 🐞 on the toolbar.

After you start the debug session, several things happen. In the Run window near the bottom of the Desktop, you should see a message indicating that the debugger has been launched. In the CSD window, the line with the breakpoint set is eventually highlighted, indicating that the program will execute this statement next. On the left side of the jGRASP desktop, the Debug tab is popped to the top. Each of these can be seen in Figure 6-2. Notice the Debug tab pane is further divided into three sub-panes or sections labeled **Threads**, **Call Stack**, and **Variables/Eval**. Each of these sections can be resized by selecting and dragging one of the horizontal partitions.

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

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

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



Threads Section (folded)

Call Stack Section

Variables/Eval Section

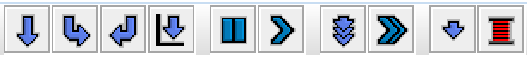
Highlighted Line When Stopped at Breakpoint

```
//*******************************************************
public class PersonalLibrary {
   //---------------------------------------------------
   //  Instantiates a derived class and in
   //  local methods.
   //---------------------------------------------------
   public static void main(String[] ar
      Book hemingway = new Book("Heming
            "Green Hills of Africa", 23
      Fiction clancy = new Fiction("Cla
            "The Hunt for Red October",
            490, 39.0, "Sean");
      Novel grisham = new Novel("Grisha
            "The Firm", 550, 28.0, "Tom'

      System.out.println(hemingway);
      System.out.println(clancy);
      System.out.println("\n" + clancy.
```

Status: debugging user program   Line:20   Col:37   Code:0   Top:4

Figure 6-2. Desktop after debugger is started

6.4 Stepping Through a Program – the Debug Buttons



After the program stops at the breakpoint (Figure 6-2), you can use the buttons at the top of the Debug tab to *step*, *step into* a method call, *step out* of a method, *run to the cursor*, *pause* the current thread, *resume*, turn on/off *auto step* mode, turn on/off *auto resume* mode, and *suspend new threads*. The sequence of statements that is executed when you run your program is called the *control path* (or simply *path*). If your program includes statements with conditions (e.g., *if* or *while* statements), the control path will reflect the *true* or *false* state of the conditions in these statements.

Clicking the *Step* button will single step to the next statement. The highlighted line in the CSD window indicates the statement that's about to be executed. When the Step button is clicked, that statement is executed and the "highlighting" is moved to the next statement along the control path.

Clicking the *Step in* button for a statement with a method call that's part of the user's source code will open the new file, if it's not already open, and pop its CSD window to the top with the current statement highlighted. The top entry in the *Call Stack* indicates where you are in the program. Note that clicking the *Step in* button for a statement without a method call is equivalent to clicking *Step*.

Clicking the *Step out* button will take the debugger back to the point where the current method was called (i.e., it will step out of the current method). The *Call Stack* will be updated accordingly.

Clicking the *Run to Cursor* button will cause your program to step automatically until the statement with the cursor L is reached. If the cursor is not on a statement along the control path, the program will stop at the next breakpoint it encounters or at the end of the program. The *Run to Cursor* button is convenient since placing the cursor on a statement is like setting "temporary" breakpoint.

Clicking the *Pause* button will suspend the program running in debug mode. Note that if you didn't have a breakpoint set in your code, you may have to select the main thread in the *Threads* section before the *Pause* button is available. After the program has halted, refer to the *Call Stack* and select the last method in your source code that was invoked. This should open the CSD window containing the method with the current line highlighted. Click the *step* button to advance through the code.

Clicking the Resume button advances the program along the control path to the next breakpoint or to the end of the program. If you have set a breakpoint in a CSD window containing another file and this breakpoint is on the control path (i.e., in a method that gets called), then this CSD window will pop to the top when the breakpoint is reached.

The *Auto Step* button is used to toggle off and on a mode which allows you to step repeatedly after clicking the *step* button only once. This is an extremely useful feature in that it essentially let's you watch your program run. Notice that with this feature turned on, a *Delay* slider bar appears beneath the Debug controls. This allows you to set the delay between steps from 0 to 26 seconds (default is .5 seconds). While the program is auto stepping, you can stop the program by clicking the *Pause* button. Clicking the *Step* button again continues the auto stepping. Remember after turning on Auto Step, you always have to click the *step* button once to get things rolling.

The *Auto Resume* button is used to toggle off and on a mode which allows you to resume repeatedly after clicking the *Resume* button only once. The effect is that your program moves from breakpoint to breakpoint using the delay indicated on the *delay* slider bar. As with auto step above, you can click the *Pause* button to interrupt the auto resume; then click the *Resume* button again to continue the auto resume.

The *Use Byte Code Size Steps* button toggles on and off the mode that allows you to step through a program in the smallest increments possible. With this feature off, the step size is approximately one source code statement, which is what most users want to see. This feature is seldom needed by beginning and intermediate programmers.

The *Suspend New Threads* button toggles on and off the mode that will immediately suspend any new threads that start. With this feature on when the debugging process is started, all startup threads are suspended as soon as is possible. Unless you are writing programs with multiple threads, you should leave the feature turned off.

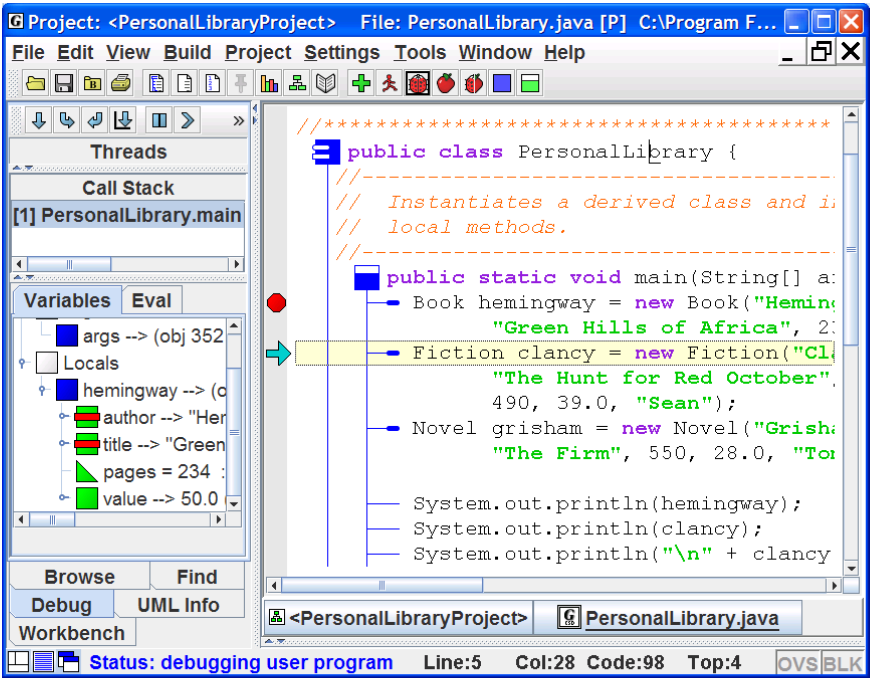
As you move through the program, you can watch the call stack and contents of variables change dynamically with each step. The integrated debugger is especially useful for watching the creation of objects as the user steps through various levels of constructors. The jGRASP debugger can be used very effectively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

6.5 Stepping Through a Program – without *Stepping In*

After initially arriving at the breakpoint in Figure 6-2, the **Variables/ Settings** section indicates that no local variables exist. Figure 6-3 shows the results of clicking the *Step* ⇩ button to move to the next statement. Notice that under Locals in the **Variables/Eval** section, we now have an instance of Book called *hemingway*. Objects, represented by a colored square, can be opened and closed by clicking the "handle" in front of the square object. Primitives, like the integer *pages*, are represented by colored triangles. In Figure 6-3, *hemingway* has been opened to show the author, title, and pages fields. Each of the String instances (e.g., author) can be opened to show the details of a String object, including the character array that holds the actual value of the string.

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

After executing the statement indicated in Figure 6-3, an instance of the Fiction class called clancy is created as shown in Figure 6-4. In the figure, clancy has been opened to reveal its fields. The field "mainCharacter" is green, indicating that it is defined in Fiction. The other fields (author, title, and pages) are orange, which indicates that these fields were inherited from Book.


Project: <PersonalLibraryProject>   File: PersonalLibrary.java [P] C:\Program F...
File Edit View Build Project Settings Tools Window Help
Threads
Call Stack
[1] PersonalLibrary.main
Variables
Eval
Browse
Find
Debug
UML Info
Workbench
<PersonalLibraryProject>
PersonalLibrary.java

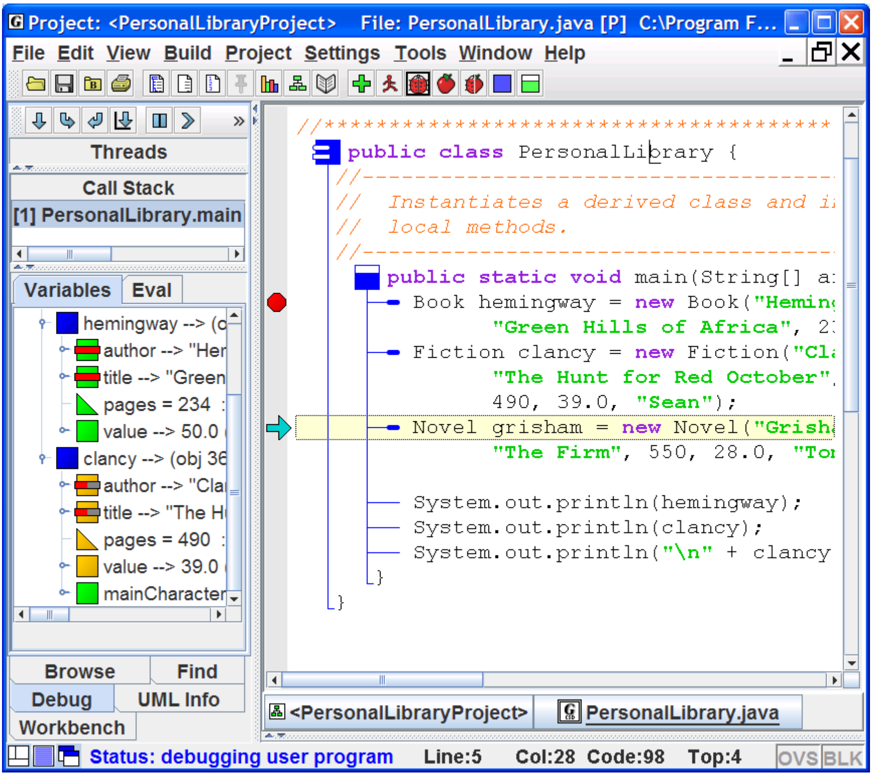
```
args --> (obj 352
Locals
   hemingway --> (o
      author --> "Her
      title --> "Green
      pages = 234
      value --> 50.0
```

```
//**************************************
public class PersonalLibrary {
   //----------------------------------
   //   Instantiates a derived class and i
   //   local methods.
   //----------------------------------
      public static void main(String[] a
         Book hemingway = new Book("Heming
               "Green Hills of Africa", 2
         Fiction clancy = new Fiction("Cl
               "The Hunt for Red October",
               490, 39.0, "Sean");
         Novel grisham = new Novel("Grisha
               "The Firm", 550, 28.0, "To

         System.out.println(hemingway);
         System.out.println(clancy);
         System.out.println("\n" + clancy
```

```
Status: debugging user program     Line:5     Col:28  Code:98     Top:4     OVS BLK
```

Figure 6-3. Desktop after hemingway (book) is 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 tab should become blank, indicating that the program is no longer running.

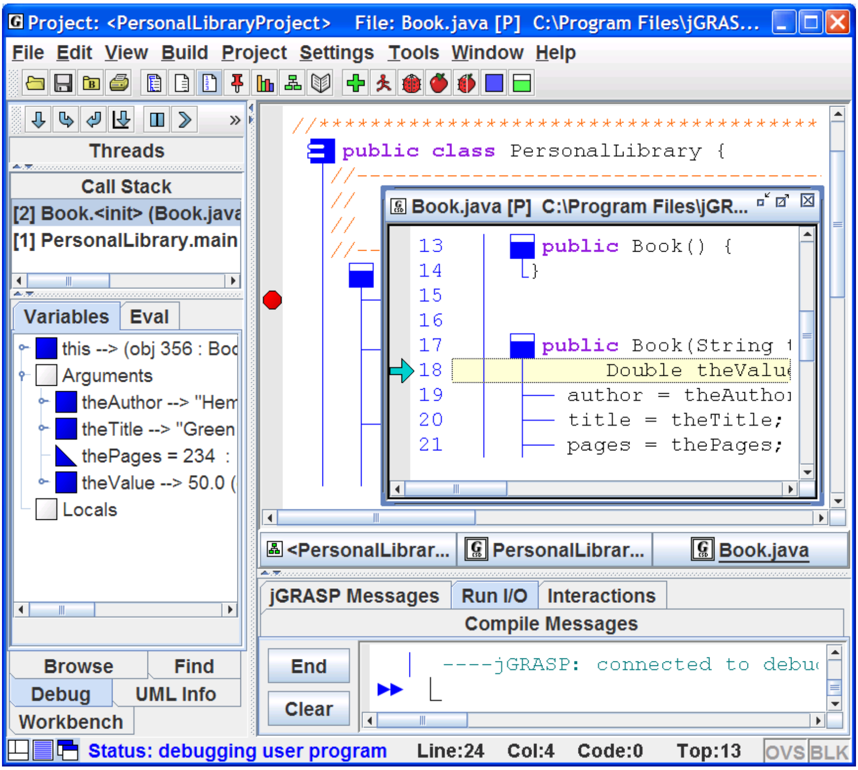


```
//**************************************
public class PersonalLibrary {
   //------------------------------------
   //  Instantiates a derived class and i
   //  local methods.
   //------------------------------------
   public static void main(String[] a
      Book hemingway = new Book("Heming
         "Green Hills of Africa", 2
      Fiction clancy = new Fiction("Cla
         "The Hunt for Red October",
         490, 39.0, "Sean");
      Novel grisham = new Novel("Grisha
         "The Firm", 550, 28.0, "To

      System.out.println(hemingway);
      System.out.println(clancy);
      System.out.println("\n" + clancy
   }
}
```

Figure 6-4. After next step and "clancy" created

6.6 Stepping Through a Program – and *Stepping In*

Now we are ready to make a second pass and "step in" to the methods called. Tracing through a program by following the calls to methods can be quite instructive in the obvious way. In the object-oriented paradigm, it is quite useful for illustrating the concept of constructors. As before, we need to run the example program in the debugger by clicking **Build – Debug** on the CSD window menu or by clicking the debug button on the toolbar. After arriving at the breakpoint, we click the *Step in* button and the constructor for class Book pops up in the CSD window (Figure 6-5). Notice that the C*all Stack* in the Debug tab indicates that you have moved into Book from PersonalLibrary (i.e., the entry for Book is listed above PersonalLibrary in the *call stack*). If you click on the PersonalLibrary entry in the call stack, the associated CSD window will pop to the top and you will see the variables associated with it. If you then click the Book entry, its CSD window will pop to the top and you will see the variables associated with the call to Book’s constructor. In Figure 6-5, the entry for *this* has been expanded in the *Variables* section. The *this* object represents the object that is being constructed. Notice that none of the fields have a red border since we are inside the Book class. As you step through the constructor, you should see the fields in *this* get initialized to the values passed in as arguments. Also, note the *id* for *this* (it is 356 in our example debug session; it may be a different number in your session). You can then step through the constructor in the usual way, eventually returning to the statement in the main program that called the constructor. One more step should finally get you to the next statement, and you should see *hemingway* in the *Variables* section with the same *id* as you saw in the constructor as it was being built. If you expand *hemingway*, you should see that the red borders are back on *author* and *title* since we’re no longer in the Book class.



Project: <PersonalLibraryProject>   File: Book.java [P] C:\Program Files\jGRAS...

[2] Book.<init> (Book.java
[1] PersonalLibrary.main

this --> (obj 356 : Boo
Arguments
theAuthor --> "Hem
theTitle --> "Green
thePages = 234 :
theValue --> 50.0 (
Locals

```
//**********************************
public class PersonalLibrary {
//-----------------------------------
//
//
//--
```

Book.java [P] C:\Program Files\jGR...

```
   public Book() {
   }


   public Book(String t
      Double theValue
      author = theAuthor
      title = theTitle;
      pages = thePages;
```

----jGRASP: connected to debu

Status: debugging user program   Line:24   Col:4   Code:0   Top:13   OVS BLK

Figure 6-5. After next stepping into the Book 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.

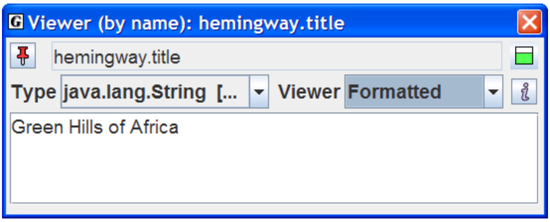
Another even more common example is when the *toString* method of an object is invoked indirectly in a print statement (System.out.println). The debugger actually takes the user to the object’s respective *toString* method.

6.7 Opening Object Viewers

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

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

Figure 6-6 shows an object viewer for the *title* field of *hemingway* in Figure 6-4, which is a String object in an instance of Book. *Formatted* is the default “view” for a String object which is especially useful when viewing a String object with a large value (e.g., a page of text). In Figure 6-7, the *Basic* view has been selected and expanded to show the details of the String object. Notice that the first field is value[21] which is a character array holding the actual value of the string. If we open a separate viewer on *value*, we have a *Presentation* view of the array as shown in Figure 6-8. Notice that the first element (‘G’) in the array has been selected and this opened a subview of type character. The subview displays the ‘G’ and its integral value of 71. If our example had been an array of strings (e.g., a list of words) then selecting an array element would have displayed the formatted view of a String object in the subview. Presentation view is the default for arrays. There is also a view called *Array Elements* which is quite useful for large arrays.



Viewer (by name): hemingway.title

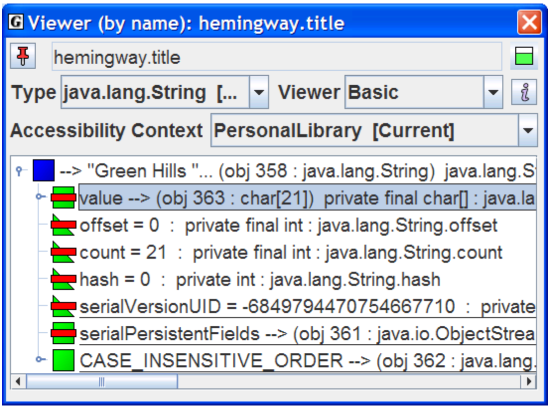
hemingway.title

Type java.lang.String [...

Viewer Formatted

Green Hills of Africa

Figure 6-6. Viewing a String Object



Viewer (by name): hemingway.title

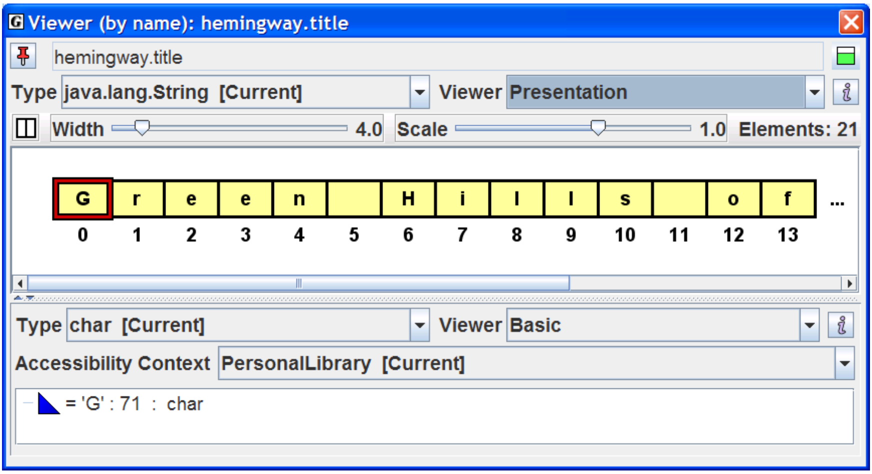
hemingway.title

Type java.lang.String [... Viewer Basic

Accessibility Context PersonalLibrary [Current]

```
--> "Green Hills "... (obj 358 : java.lang.String) java.lang.S
  value --> (obj 363 : char[21]) private final char[] : java.la
  offset = 0 : private final int : java.lang.String.offset
  count = 21 : private final int : java.lang.String.count
  hash = 0 : private int : java.lang.String.hash
  serialVersionUID = -6849794470754667710 : private
  serialPersistentFields --> (obj 361 : java.io.ObjectStrea
  CASE_INSENSITIVE_ORDER --> (obj 362 : java.lang.
```

Figure 6-7. Basic view of a string (expanded to see fields)



Viewer (by name): hemingway.title

hemingway.title

Type java.lang.String [Current] Viewer Presentation

Width 4.0 Scale 1.0 Elements: 21

G	r	e	e	n		H	i	l	l	s		o	f	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	

Type char [Current] Viewer Basic

Accessibility Context PersonalLibrary [Current]

= 'G' : 71 : char

Figure 6-8. Presentation View of hemingway.title.value

You are encouraged to open separate viewers for any of the primitives and objects in the Variables section of the Debug tab. In addition to providing multiple views of the object, each viewer includes an Invoke Method button 🟩 for the object being viewed.

In the tutorial *Viewers for Data Structures*, many other examples are presented along with a more detailed description of viewers in general. The "Structure Identifier" viewer is also introduced. This viewer attempts to automatically recognize and display linked lists, binary trees, and array wrappers (lists, stacks, queues, etc.) when opened on an object during debugging or workbench use.

6.8 Debugging a Program

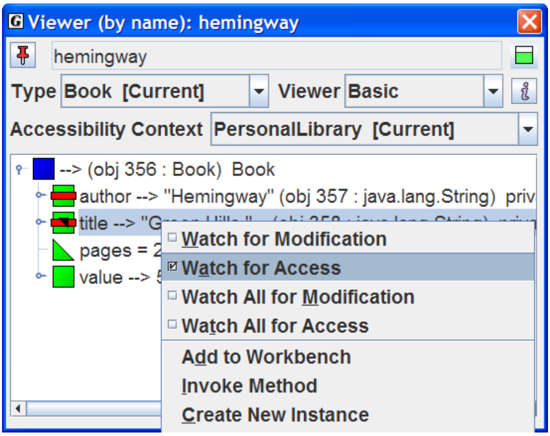
You have, no doubt, noticed that the previous discussion was only indirectly related to the activity of finding and removing *bugs* from your program. It was intended to show you how to set and unset breakpoints and how to step through your program. Typically, to find a bug in your program, you need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

You can also set several types of “watches” on a field of an object. In Figure 6-9, a *Watch for Access* has been set on the *title* in *hemingway* just after it was created. If you click the Resume button ❯ at this point, with no breakpoints set before the end of the program, the next place the program should stop is in the *toString* method of Book in conjunction with the *println* statement for *hemingway*. This is because the *title* field of *hemingway* is accessed in the statement:

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

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

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



Viewer (by name): hemingway

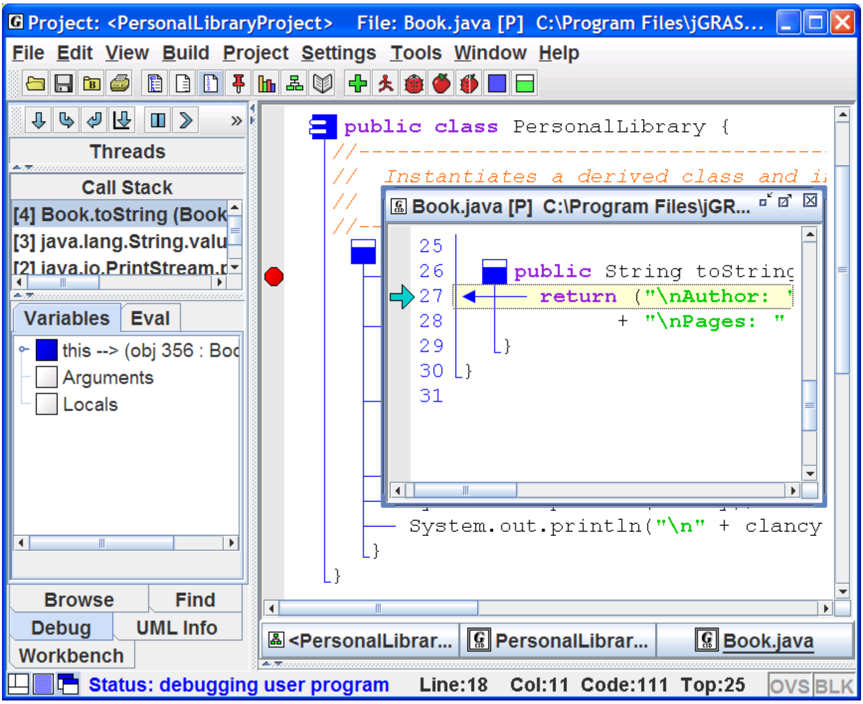
hemingway

Type Book [Current]   Viewer Basic

Accessibility Context PersonalLibrary [Current]

```
--> (obj 356 : Book) Book
  author --> "Hemingway" (obj 357 : java.lang.String) priv
  title --> "G
  pages = 2
  value --> 5
```

Figure 6-9. Setting a Watch for Access



Project: <PersonalLibraryProject> File: Book.java [P] C:\Program Files\jGRAS...

File Edit View Build Project Settings Tools Window Help
Threads
Call Stack

```
[4] Book.toString (Book
[3] java.lang.String.valu
[2] java.io.PrintStream.p
```

```
this --> (obj 356 : Boo
  Arguments
  Locals
```

```
public class PersonalLibrary {
//------------------------------------
//  Instantiates a derived class and i
//
//--
```

Book.java [P] C:\Program Files\jGR...

```
25
26      public String toString
27         return ("\nAuthor: "
28                + "\nPages: "
29      }
30   }
31
```

```
      System.out.println("\n" + clancy
   }
}
```

Browse Find
Debug UML Info
Workbench
<PersonalLibrar... PersonalLibrar... Book.java

Status: debugging user program   Line:18 Col:11 Code:111 Top:25 OVS BLK

Figure 6-10. Stopping at a Watch for Access to hemingway.title

7 Projects

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

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

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

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

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

7.1 Creating a Project
7.2 Adding files to the Project
7.3 Removing files from the Project
7.4 Generating Documentation for the Project (Java only)
7.5 Jar File Creation and Extraction
7.6 Closing a Project
7.7 Exercises

7.1 Creating a Project

On the Desktop menu, click **Project** > **New** > **New Standard Project** (Figure 7-1) to open the **New Project** dialog. *Note that the "New J2ME Project" option should only be selected if you have installed the Java Wireless Took Kit (WTK) and you plan to develop a project based on the Java 2 Micro Edition (J2ME).*

Within the New Project dialog (Figure 7-2), notice the two check boxes (*Add Files Now* and *Open UML Window*). Normally, you would want to have the *Add Files Now* checked ON so that as soon as you click the Create button, the Add Files dialog will pop up. If you are working in Java, you may also want to turn ON the *Open UML Window* option. This will generate the UML class diagram and open the UML window (see Section 5 for details).

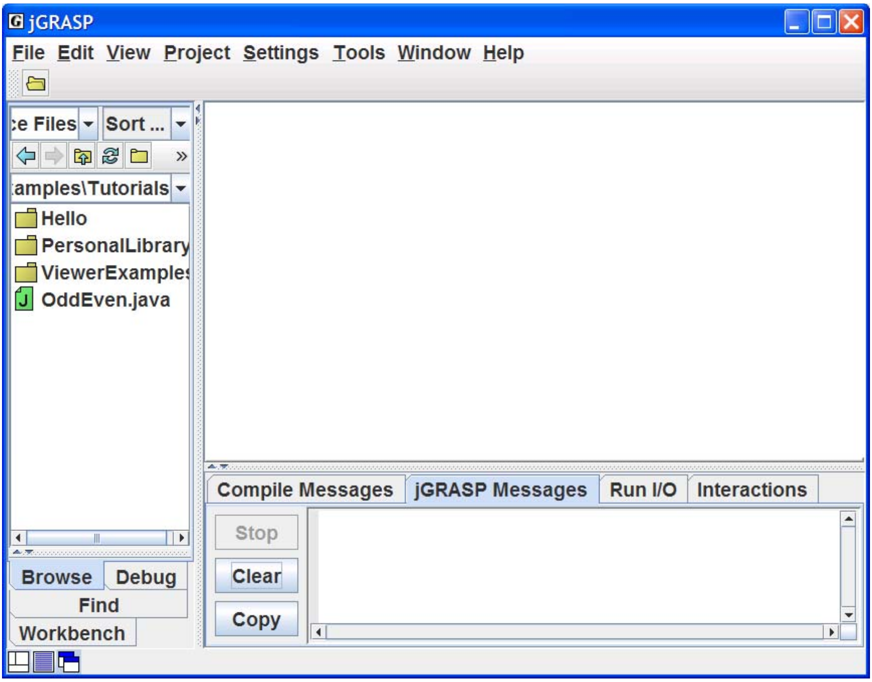
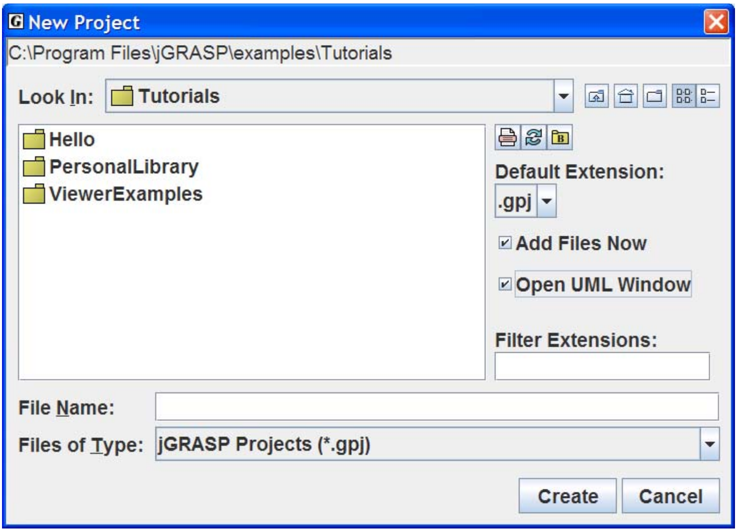


Figure 7-1. Creating a Project



New Project

C:\Program Files\jGRASP\examples\Tutorials

Look In: Tutorials

Hello
PersonalLibrary
ViewerExamples

Default Extension: .gpj

☑ Add Files Now

☑ Open UML Window

Filter Extensions:

File Name:

Files of Type: jGRASP Projects (*.gpj)

Figure 7-2. New Project dialog

Navigate to the directory where you want the project to reside and enter the project file name. It is recommended that the project file be stored in the same directory as the file containing *main*. A useful naming convention for a project is *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 new project file with .gpj extension is listed in the Files section of the Browse tab. The project is also listed in the Open Projects section of Browse tab. If *Add Files Now* was checked ON when you created the project, the Add Files dialog will pop up. As files are added to the project, they will appear under the project name in the Open Projects section of the Browse tab. When you have finished adding files, click the *Close button* on the dialog. You can always add more files to a project later.

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

7.2 Adding files to the Project

The Browse tab is split to show the current file directory in the top part and the open projects in the lower part as shown in Figure 7-3. After a project has been created and/or opened, there are several ways to add Java files to the project.

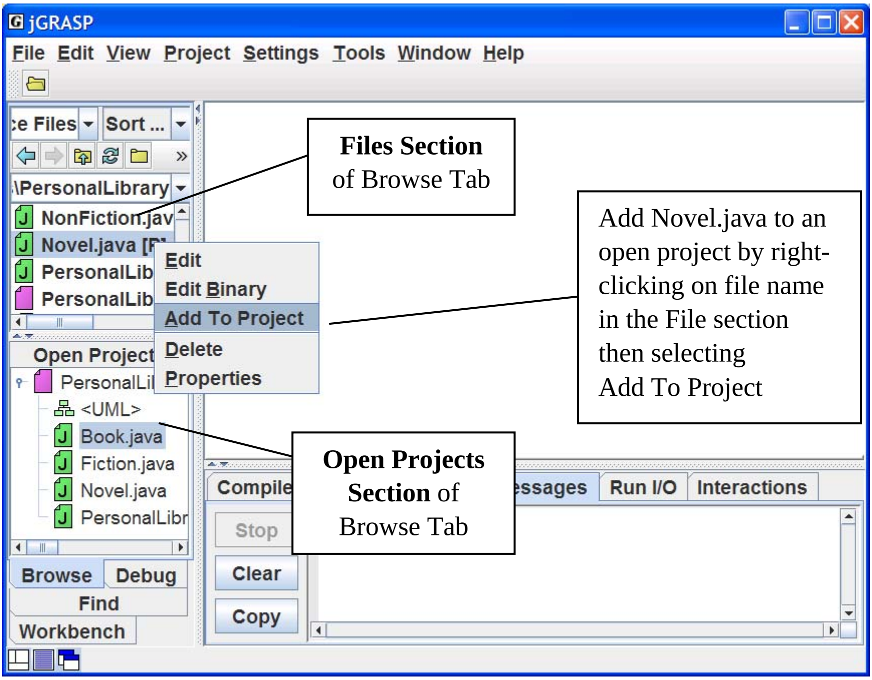
(1) **From Browse Tab** - Drag the file (left click and hold) from the Files section to the project in the Open Projects section below.

(2) **From Browse Tab** - Drag the file from the Files section to the UML Window.

(3) **In Browse Tab** - Right click on the file and select **Add to Project**. (Figure 7-3).

(4) **From CSD window** > Click **Project** > **Add files**.

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



Files Section of Browse Tab

Add Novel.java to an open project by right-clicking on file name in the File section then selecting Add To Project

Open Projects Section of Browse Tab

Figure 7-3. Adding a file to the Project

7.3 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 7-4. You can also remove the selected file(s) by pressing *Delete* on the keyboard. Note that *removing* a file from a project does not delete the file from its directory, only from the project. However, you can delete a file by selecting it in the Files section of the Browse tab, then right-clicking and selecting **Delete** from the pop-up menu or by pressing the *Delete* key.

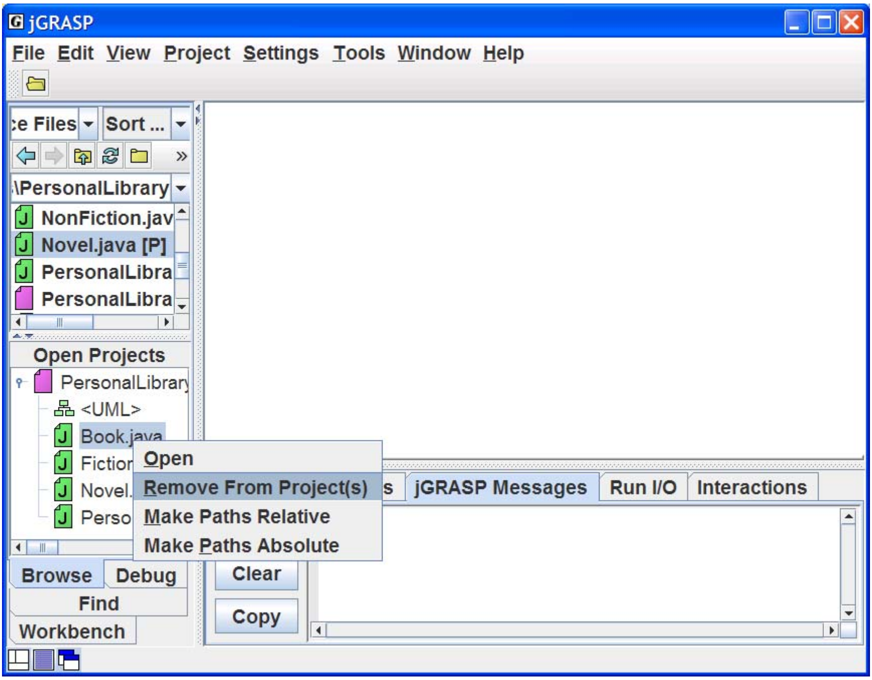


Figure 7-4. Removing a file from the Project

7.4 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., application programmer interface (API) documentation. To generate the documentation for our example project, PersonalLibaryProject, select **Project** > **Generate Documentation** > **<PersonalLibaryProject>** as shown in the Figure 7-5. This will bring up the "Generate Documentation for Project" dialog which asks for the directory in which the generated HTML files are to be stored. The default directory name is the project name with "_doc" appended to it (e.g., PersonalLibaryProject_doc). Using the default name is recommended so that your documentation directories will have a standard naming convention. If the default directory is not indicated, click the **Default** button in the dialog. However, you are free to use any directory as the target. Click the **Generate** button on the dialog to start the process. jGRASP calls the javadoc utility, which is included with the JDK, to create a complete hyper-linked document.

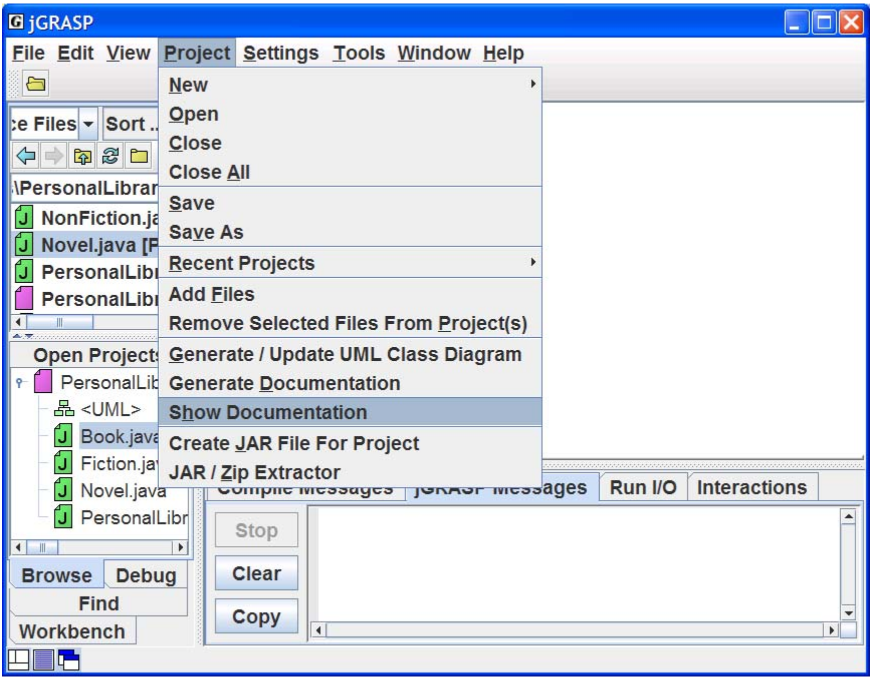
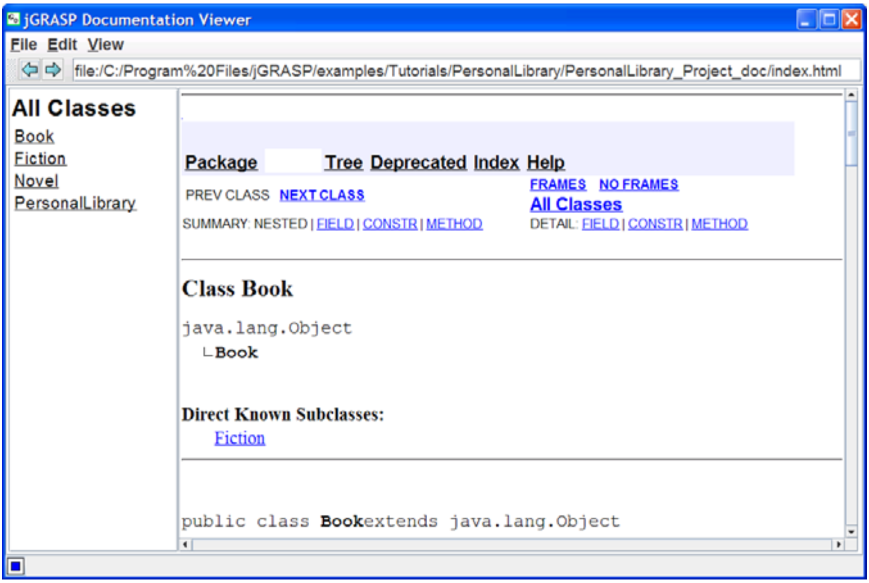


Figure 7-5. Generating Documentation for the Project

The documentation generated for PersonalLibaryProject is shown below in Figure 7-6. Note that in this example, even though no JavaDoc comments were included in the source file, the generated documentation is still quite useful. However, for even better documentation, JavaDoc formal comments should be included in the source code. When generated for a project, the documentation files are stored in a directory that becomes part of the project and, therefore, persists from one jGRASP session to the next. **Project** > **Show Documentation** can be used to display the documentation without regenerating it. However, if any changes have been made to a project source file and the file has been saved, jGRASP will indicate that the documentation needs to be regenerated. You may choose to view the documentation anyway or to regenerate the documentation.


jGRASP Documentation Viewer
File Edit View
file:/C:/Program%20Files/jGRASP/examples/Tutorials/PersonalLibrary/PersonalLibrary_Project_doc/index.html
All Classes
Book
Fiction
Novel
PersonalLibrary
Package Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class Book

```
java.lang.Object
  └Book
```

Direct Known Subclasses:
Fiction

```
public class Bookextends java.lang.Object
```

Figure 7-6. Project documentation

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

7.5 Jar File Creation and Extraction

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

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

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

7.6 Closing a Project

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

(1) From the Desktop toolbar - Click **Project** > **Close All Projects**.

(2) From the Desktop toolbar - Click **Project** > **Close** (if more than one project is open, select the one you want to close, e.g., ***<PersonalLibraryProject>.***

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

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

7.7 Exercises

(1) Create a new project called PersonalLibraryProject2 in the same directory folder as the original PersonalLibraryProject. During the create step, add the file Book.java to the new project. Close the Add Files dialog.

(2) Add the other Java files in the directory to the project by dragging each file from the Files section of the Browse tab and dropping the files in PersonalLibraryProject2 in the open projects section.

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

(4) Generate the documentation for PersonalLibraryProject2. After the Documentation Viewer pops up:

  a. Click the Fiction class link in the API documentation (left side).

  b. Click the Methods link to view the methods for the Fiction class.

  c. Visit the other classes in the documentation for the project.

(5) Close the project.

(6) Open the project by double-clicking on the project file in the files section of the Browse tab.

Notes

8 UML Class Diagrams

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

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

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

8.1 Opening the Project
8.2 Generating the UML
8.3 Compiling and Running from the UML Window
8.4 Determining the Contents of the UML Class Diagram
8.5 Laying Out the UML Class Diagram
8.6 Displaying the Members of a Class
8.7 Displaying Dependencies Between Two Classes
8.8 Navigating to Source Code via the Info Tab
8.9 Finding a Class in the UML Diagram
8.10 Opening Source Code from UML
8.11 Saving the UML Layout
8.12 Printing the UML Diagram

8.1 Opening the Project

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

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

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

(1) On the Desktop tool bar, click **Project** > **Open Project**, and then select the project from the list of project files displayed in the Open Project dialog and click the **Open** button.

(2) Alternatively, in the files section of the Browse tab, double-click the project file.

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

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

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

8.2 Generating the UML

In Figure 8-1, PersonalLibraryProject is shown in the Open Projects section of the Browse tab along with a **UML** symbol 🖧 and the list of files in the project. To generate the UML class diagram, double-click the UML symbol 🖧. Alternatively, on the Desktop menu, click on **Project** > **Generate/Update UML Class Diagram**.

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

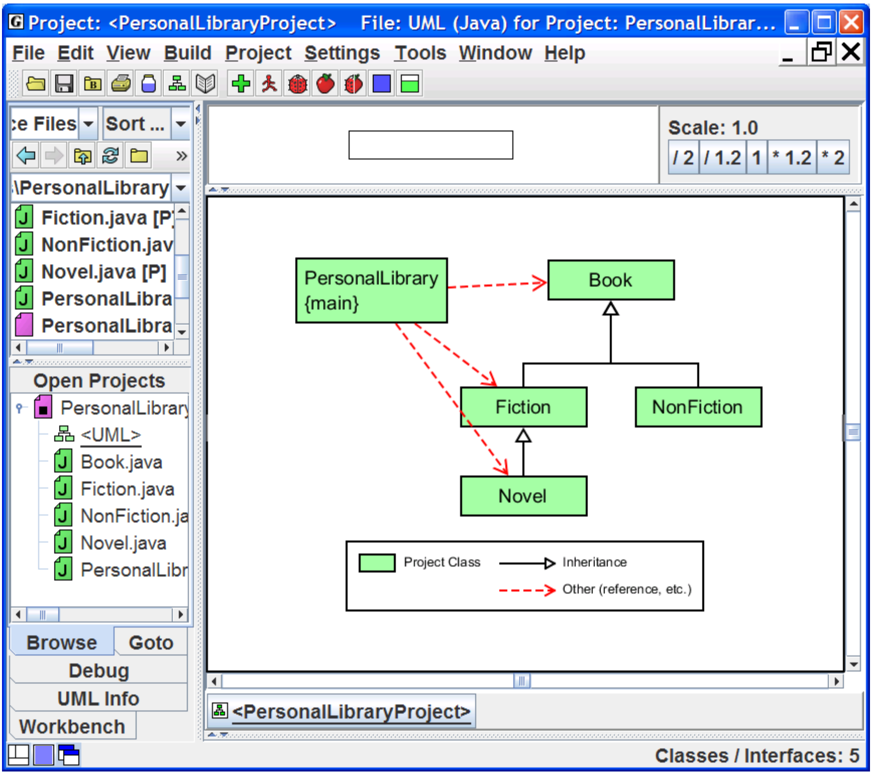


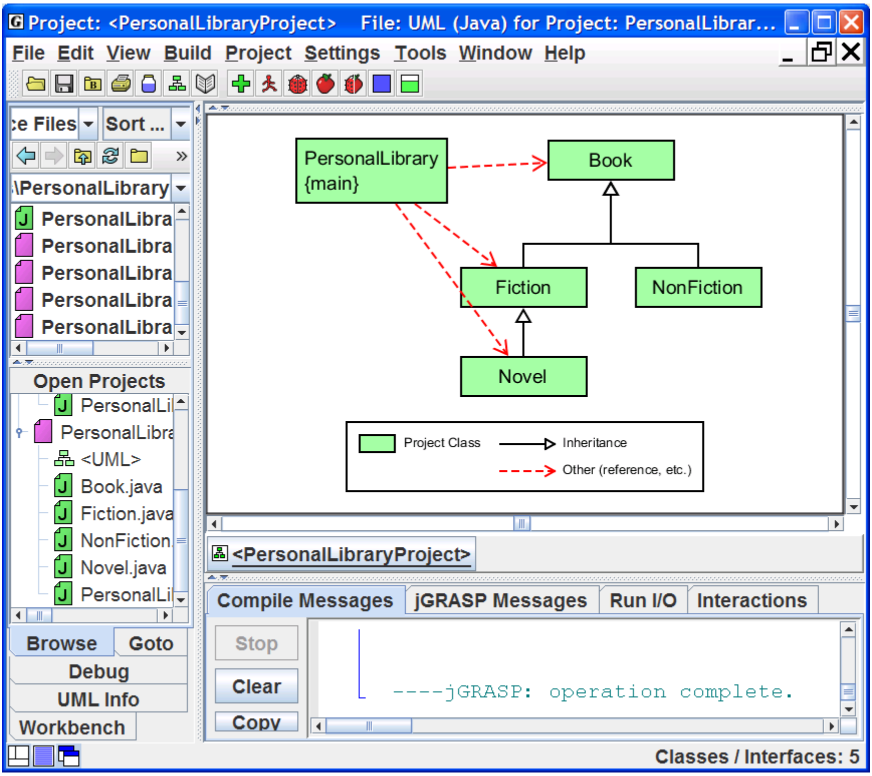
Figure 8-1. Generating the UML

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 buttons for scaling the UML: divide by 2 (/2), divide by 1.2 (/1.2), no scaling (1), multiply by 1.2 (*1.2), and multiply by 2 (*2). In general, the class diagram is automatically updated as required; however, the user can force an update by clicking the Update UML diagram button on the desktop menu.

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

8.3 Compiling and Running from the UML Window

The Build menu and buttons on the toolbar for the UML window are essentially the same as the ones for the CSD window. For example, clicking the Compile button compiles all classes in the project (Figure 8-2). When a class needs to be recompiled due to edits, the class symbol in the UML diagram is marked with red crosshatches (double diagonal lines). During compilation, the files are marked and then unmarked when done. Single red diagonal lines in a class symbol indicate that another class upon which the first class depends has been modified. Clicking the **Run** button on the toolbar will launch the program as an application if there is a *main()* method in one of the classes. Clicking on the **Run as Applet** button will launch the program as an applet if one of the classes is an applet. Similarly, clicking the **Debug** button or the **Debug Applet** button will launch the program in debug mode. Note that for running in debug mode, you should have a breakpoint set somewhere in the program so that the debugger will stop.



```
 ----jGRASP: operation complete.
```

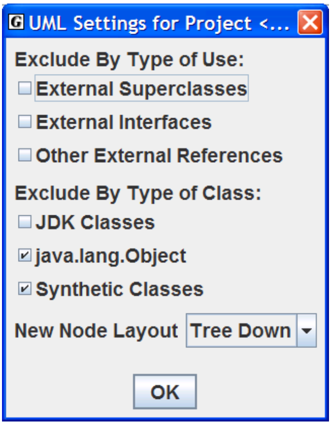
Figure 8-2. Compiling Your Program

8.4 Determining the Contents of the UML Class Diagram

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

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

Excluding (or not) items from the diagram - On the UML window menu, click on **Settings** > **UML Generation Settings**, which will bring up the **UML Settings** dialog. Generally you should leave the top three items unchecked so that they are not excluded from the UML diagram. Now for our example of not excluding the JDK classes, under **Exclude by Type of Class**, uncheck (turn OFF) the checkbox that excludes **JDK Classes,** as shown in Figure 8-3. Note that synthetic classes are created by the Java compiler and are usually not included in the UML diagram. After checking (or unchecking) the items so that your dialog looks like the one in the figure, click the OK button. This should close the dialog and update the UML diagram. All JDK classes used by the project classes should now be visible in the diagram as gray boxes. This is shown in Figure 8-4 after the JDK classes have been dragged around. To remove them from the diagram, you will need to turn on the exclude option. If you want to leave them in the diagram but not always display them, see the next paragraph. For more information see UML Settings in jGRASP **Help**.



Exclude By Type of Use:

☐ External Superclasses

☐ External Interfaces

☐ Other External References

Exclude By Type of Class:

☐ JDK Classes

☑ java.lang.Object

☑ Synthetic Classes

New Node Layout: Tree Down

Figure 8-3. Changing the UML Settings

Making objects in the diagram visible (or not) - On the UML window menu, click on **View** > **Visible Objects**, then check or uncheck the items on the list as appropriate. In general, you will want all of the items on the list in **View** >

Visible Objects checked ON as shown in Figure 8-4. For example, for the JDK classes and/or other classes outside the project to be visible, **External References** must be checked ON. Clicking (checking) ON or OFF any of the items on the Visible Objects list simply displays them or not, and their previous layout is retained when they are redisplayed. Note that if items have been *excluded* from the diagram via **Settings** > **UML Generation Settings**, as described above, then making them visible will have no effect since they are not part of the diagram. For more information see View Menu in jGRASP **Help**.

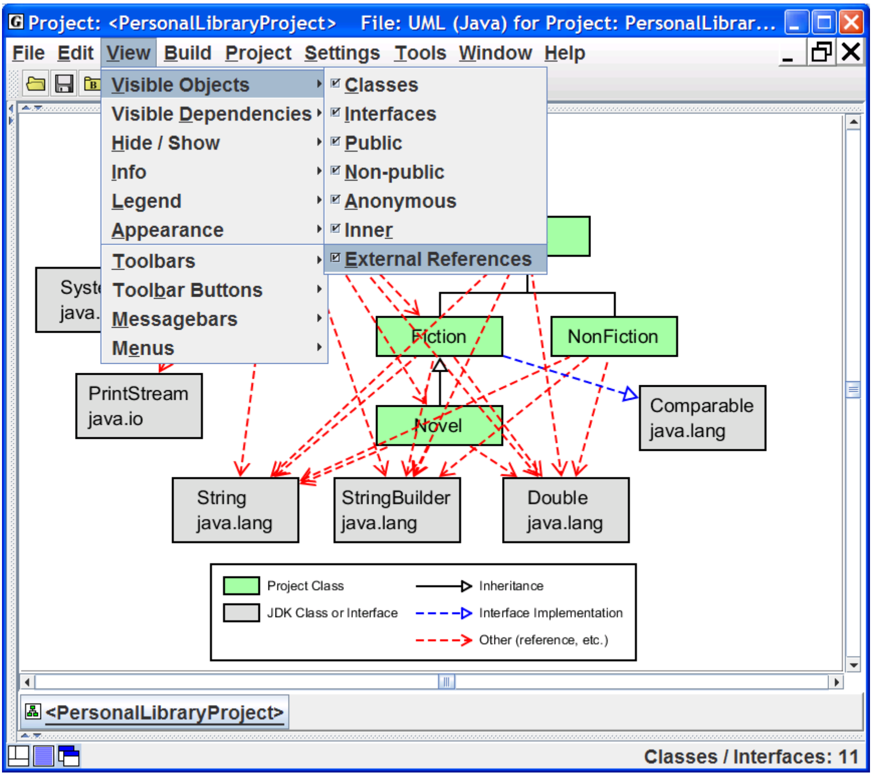


Figure 8-4. Making objects visible

Making dependencies visible - On the UML window menu, click on **View** > **Visible Dependencies**, then check or uncheck the items on the list as appropriate. The only two categories of dependencies in the example project are **Inheritance** and **Other**. **Inheritance dependencies** are indicated by black lines with closed arrowheads that point from a child to a parent to form an *is-a* relationship. Red dashed lines with open arrowheads indicate **other dependencies**. These include *has-a* relationships that indicate that a class uses fields, methods, or constructors of another class. The red dashed arrow is drawn from the class where an object is declared or referenced to the class where the item is actually defined. In general, you probably want to make all dependencies visible as indicated in Figure 8-5.

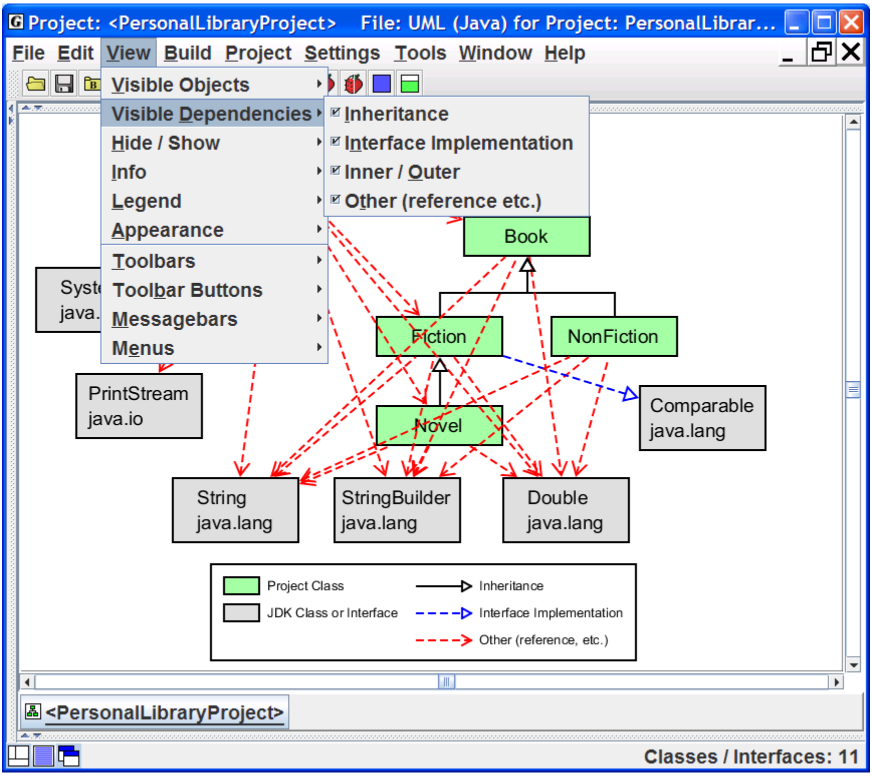


Figure 8-5. Making dependencies visible

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

8.5 Laying Out the UML Class Diagram

Currently, the jGRASP UML diagram 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 that 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 where you want it, and then release the mouse button. Now repeat this for the other class symbols until you have the diagram looking like you want it. Keep in mind that class–subclass relationships are indicated by the *inheritance arrow* and that these should be laid out in a tree-down fashion. You can do this automatically by selecting all classes for a particular class–subclass hierarchy (hold down SHIFT and left-click each class). Then click **Edit** > **Layout** > **Tree Down** to perform the operation; alternatively, you can right-click on a selected class or group of classes, then on the pop up menu select **Layout** > **Tree Down**. Finally, right-clicking in the background of the UML window with no classes selected will allow you to lay out the entire diagram.

With a two or more classes selected, you can move them as a group. Figure 8-5 shows the UML diagram after the PersonalLibrary class has been repositioned to the top left and the JDK classes have been dragged as a group to the lower part of the diagram. You can experiment with making these external classes visible by going to **View** > **Visible Objects** > then uncheck **External References**.

Here are several heuristics for laying out your UML diagrams:

(1) The class symbol that contains *main* should go near the top of the diagram.

(2) Classes in an inheritance hierarchy should be laid out *tree-down*, and then moved as group.

(3) Other dependencies should be laid out with the red dashed line pointing downward.

(4) JDK classes, when included, should be toward the bottom of the diagram.

(5) Line crossings should be minimized.

(6) The legend is usually below the diagram.

8.6 Displaying the Members of a Class

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

In Figure 8-6, external classes are not visible (**View** > **Visible Objects** > then uncheck **External References**). Class Fiction has been selected and its fields, constructors, and methods are displayed in the left pane. This information is only available when the source code for a class is in the project. In the previous example, the System class from package java.lang is an external class, so selecting it would result in a "no data" message. If the only field you are seeing is mainCharacter, click **View** > **Info** > **Show Inheritance within Project**. You should now see the fields that are inherited by Fiction (i.e., author, pages, and title).

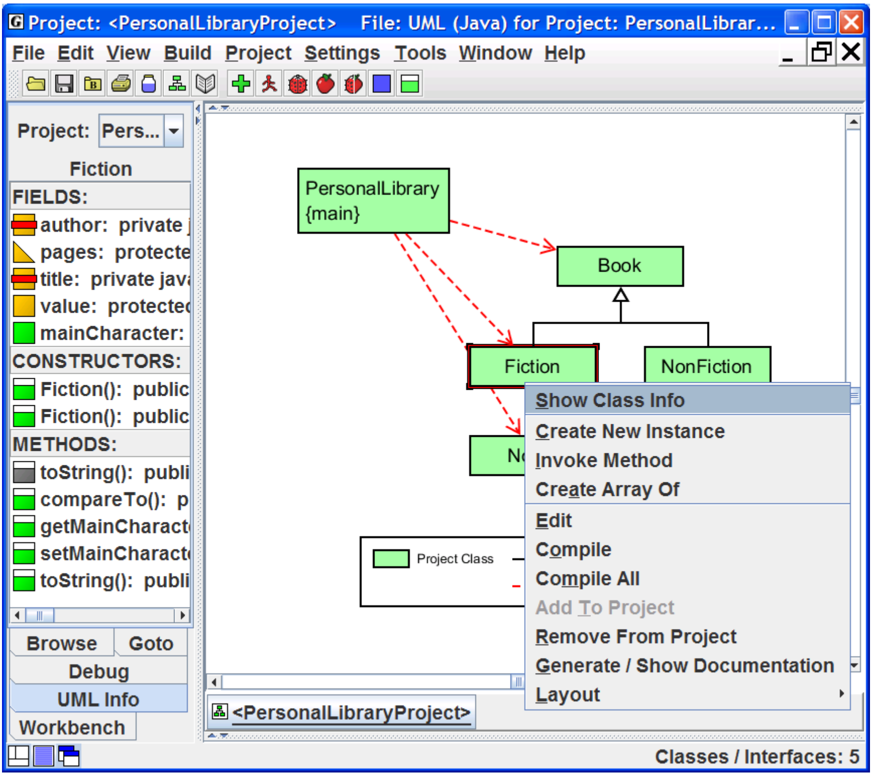


Figure 8-6. Displaying class members

8.7 Displaying Dependencies Between Two Classes

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

In Figure 8-7, the edge drawn from PersonalLibrary to Fiction has been selected as indicated by the large arrowhead. The list of dependencies in the Info tab includes one constructor (Fiction) and one method (getMainCharacter). These are the resources that PersonalLibrary uses from Fiction. Understanding the dependencies among the classes in your program should provide you with a more in-depth comprehension of the source code. Note that clicking on the arrow between PersonalLibary and the PrintStream class in Figure 8-6 would show that PersonalLibary is using two println() methods from the PrintStream class. Make the **External References** visible again and try this**.**

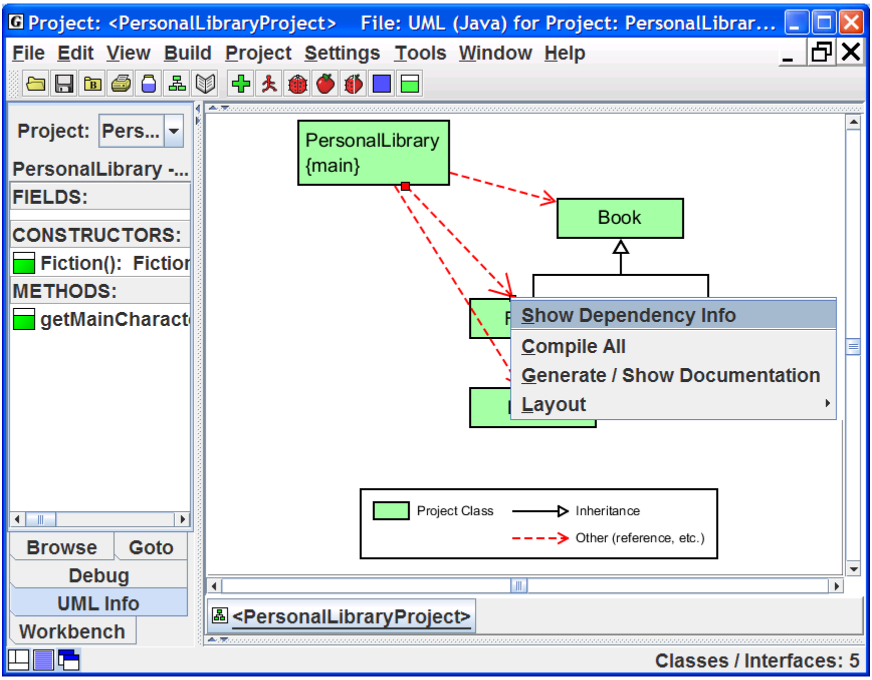


Figure 8-7. Displaying the dependencies between two classes

8.8 Navigating to Source Code via the Info Tab

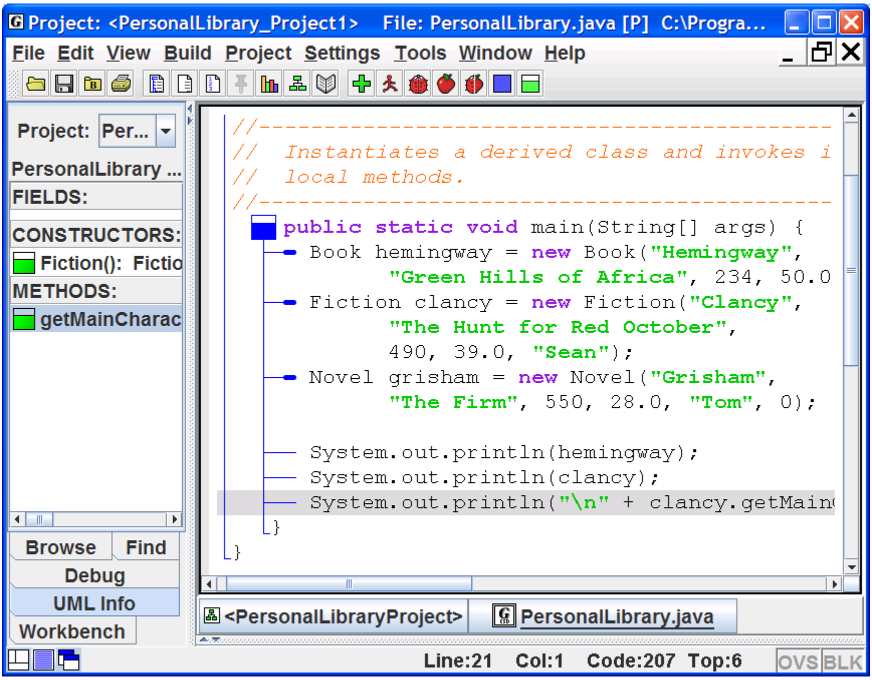
In the Info tab, a green symbol indicates that the item is defined or used in the class rather than inherited from a parent class. Double-clicking on a green item will take you to its definition or use in the source code. For example, clicking on getMainCharacer() in Figure 8-7 above will open PersonalLibrary in a CSD window with the line containing getMainCharacter() highlighted as shown in Figure 8-8 below.

8.9 Finding a Class in the UML Diagram

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

8.10 Opening Source Code from UML

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



```
//-----------------------------------------------
//  Instantiates a derived class and invokes i
//  local methods.
//-----------------------------------------------
   public static void main(String[] args) {
      Book hemingway = new Book("Hemingway",
            "Green Hills of Africa", 234, 50.0
      Fiction clancy = new Fiction("Clancy",
            "The Hunt for Red October",
            490, 39.0, "Sean");
      Novel grisham = new Novel("Grisham",
            "The Firm", 550, 28.0, "Tom", 0);

      System.out.println(hemingway);
      System.out.println(clancy);
      System.out.println("\n" + clancy.getMain(
   }
}
```

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

8.11 Saving the UML Layout

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

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

8.12 Printing the UML Diagram

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

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

9 The Workbench

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

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

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

9.1 Invoking Static Methods from the CSD Window
9.2 Invoking Static Methods from the UML Window
9.3 Creating an Object for the Workbench
9.4 Invoking a Method
9.5 Invoking Methods with Parameters Which Are Objects
9.6 Invoking Methods on Object Fields
9.7 Selecting Categories of Methods to Invoke
9.8 Opening Object Viewers
9.9 Running the Debugger on Invoked Methods
9.10 Exiting the Workbench

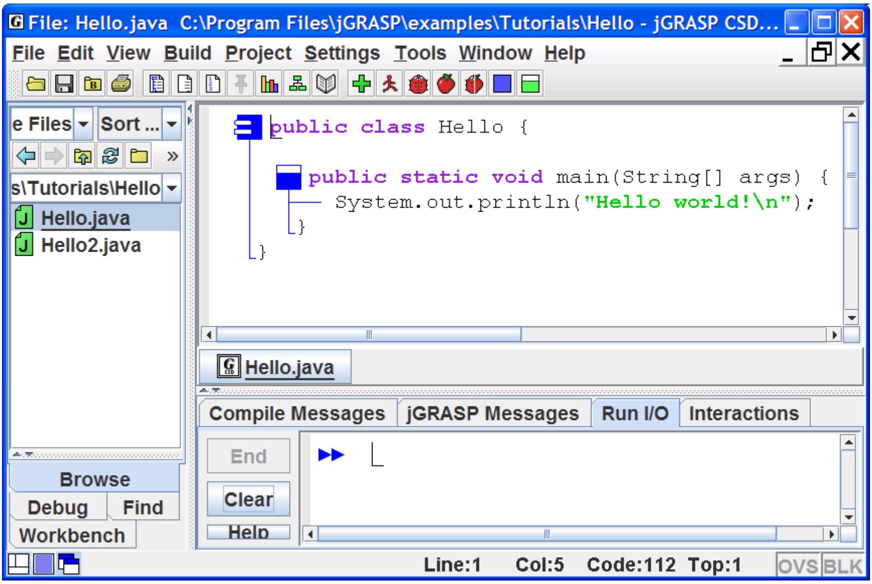
9.1 Invoking Static Methods from the CSD Window

In the tutorial *Getting Started*, we ran the Hello program in Figure 9-1 as an application by clicking the Run button . Now let's see how we can invoke its *main* method directly by using the workbench. Since *main* is a static method, it is associated with the Hello class rather than an instance of the Hello class; therefore, we don't have to create an instance for the workbench. There are two ways to invoke a static method from the CSD window:

a. Click **Build** > **Java Workbench** > **Invoke Static Method**.

b. Click the Invoke Static Method button 🟩 on the toolbar.

The latter is the easiest way, so click the Invoke Static Method 🟩 button now. This pops up the Invoke Method dialog which lists the static method *main*. After selecting main, the dialog expands to show the available parameters (Figure 9-2). We can leave the *java.lang.String[] args* blank since our *main* method is not expecting command line arguments to be passed into it.



File: Hello.java C:\Program Files\jGRASP\examples\Tutorials\Hello - jGRASP CSD...

```
public class Hello {

   public static void main(String[] args) {
      System.out.println("Hello world!\n");
   }
}
```

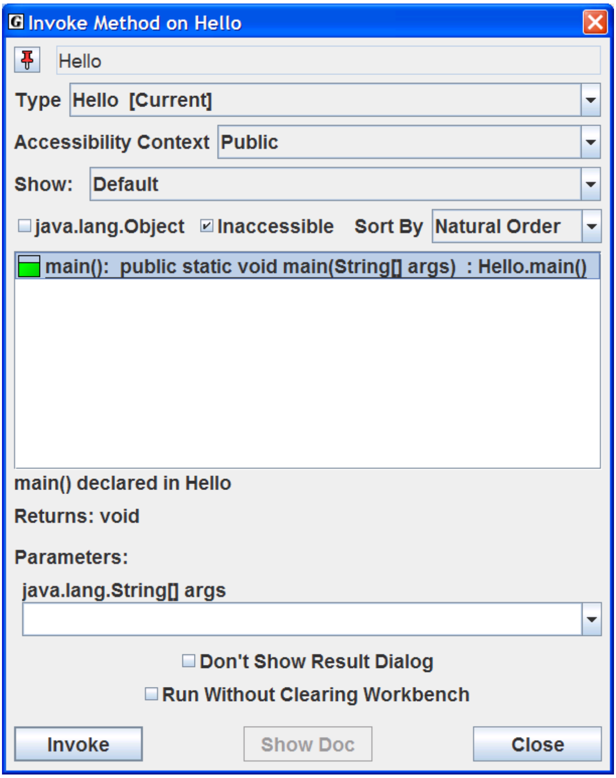
Line:1 Col:5 Code:112 Top:1

Figure 9-1. Invoking a static method from the Workbench

In Figure 9-2, notice the two check boxes below the String[] args field. The first, *Don't Show Result Dialog*, will be useful when you want to repeatedly invoke a method that has a void return type or one that you do not care about. When checked ON, all result dialogs (e.g., Figure 9-3) will be suppressed. The second check box, *Run Without Clearing the Workbench*, is a special case option for running a *main*. Normally it is okay to invoke a *main* method without clearing the workbench if you are sure this won't interfere with objects you previously placed on the workbench.

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

Now you are ready to invoke the main method by clicking the **Invoke** button in the lower left corner of the dialog. Figure 9-3 shows the desktop and the dialog that pops up with the result: "Method invocation successful (void return type)." Recall that *main* has a "void" return type. The standard output from the program, "Hello World!" appears in the Run I/O tab pane. When the return type for a method is not void, the dialog in Figure 9-3 will contain the value of the return type.



Invoke Method on Hello

Hello

Type: Hello [Current]

Accessibility Context: Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By: Natural Order

🟩 main(): public static void main(String[] args) : Hello.main()

main() declared in Hello

Returns: void

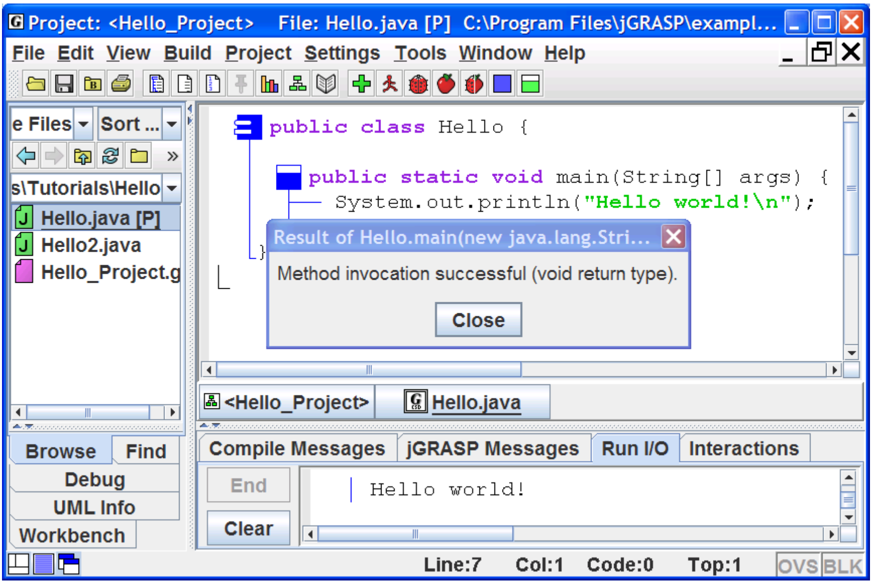
Parameters:

java.lang.String[] args

□ Don't Show Result Dialog

□ Run Without Clearing Workbench

Figure 9-2. Invoking *main*



Project: <Hello_Project>   File: Hello.java [P]   C:\Program Files\jGRASP\exampl...

```
public class Hello {

   public static void main(String[] args) {
      System.out.println("Hello world!\n");
}
```

Result of Hello.main(new java.lang.Stri...

Method invocation successful (void return type).

```
Hello world!
```

Line:7   Col:1   Code:0   Top:1

Figure 9-3. The Result dialog from invoking a method

9.2 Invoking Static Methods from the UML Window

Figure 9-4 shows that we have created a project file, Hello_Project, added Hello.java to the project, and then generated the UML class diagram. To make the class diagram more interesting, we have elected to display the Java library classes used by the Hello class. We did this by selecting **Settings** > **UML Generation Settings** – then in the dialog, we unchecked **JDK classes** under the **Exclude by Type** section. As always, feel free to substitute your own examples in the discussion below.

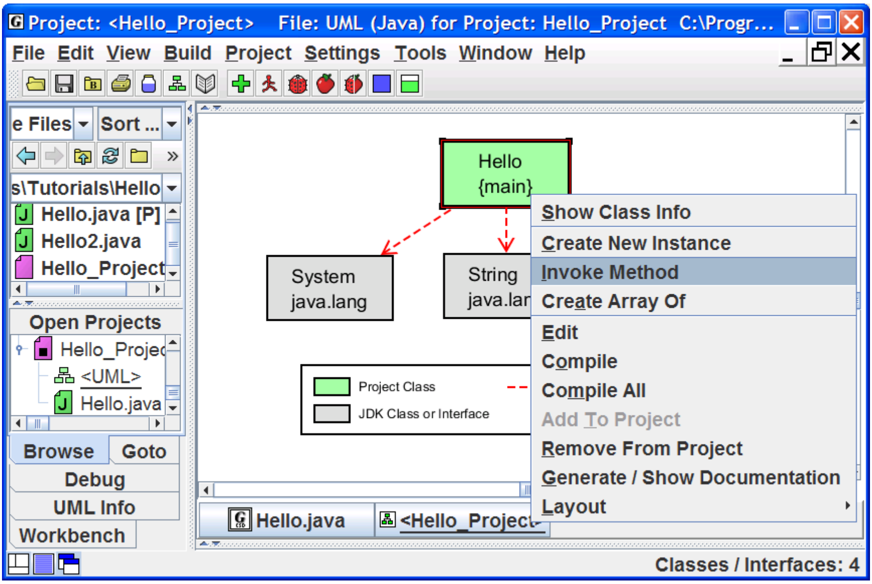
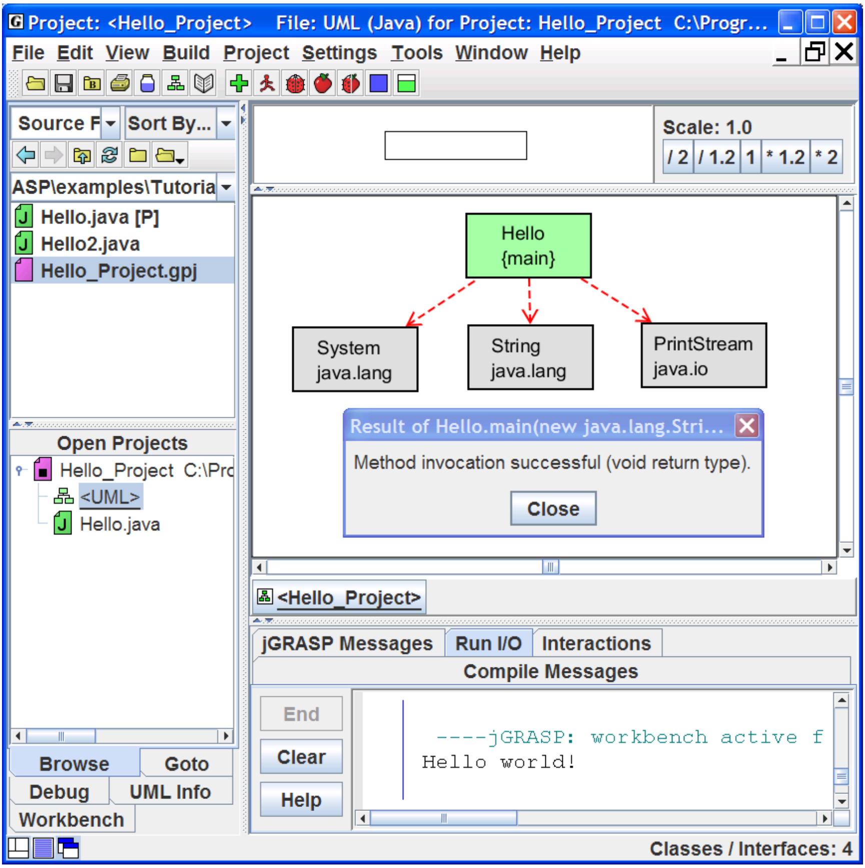


Figure 9-4. Invoking a static method from a class

Since *main* is a static method associated with the class rather than an instance of the class, it can be invoked by selecting the Hello class in the UML diagram, then right-clicking and selecting **Invoke Method**. This pops up the Invoke Method dialog which lists the static method *main* as described in the section above. After selecting *main*, leave the parameters blank, and then click the **Invoke** button. The “Result” dialog should pop up and you should see the output “Hello World!” in the Run I/O tab as shown in Figure 9-5.

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



Method invocation successful (void return type).

```
  ----jGRASP: workbench active f
Hello world!
```

Figure 9-5. Invoking a static method from a class

9.3 Creating an Object for the Workbench

Now we move to a more interesting example which contains multiple classes. Figure 9-5 shows the PersonalLibraryProject loaded in the UML window. In this section we want to create objects and place them on the workbench. In the next section, we'll see how to invoke the instance (or non-static) methods of the objects we've placed on the workbench.

We begin by right clicking on the Fiction class in the UML diagram, and then selecting **Create New Instance**, as shown in Figure 9-6. A list of constructors will be displayed in a dialog box.

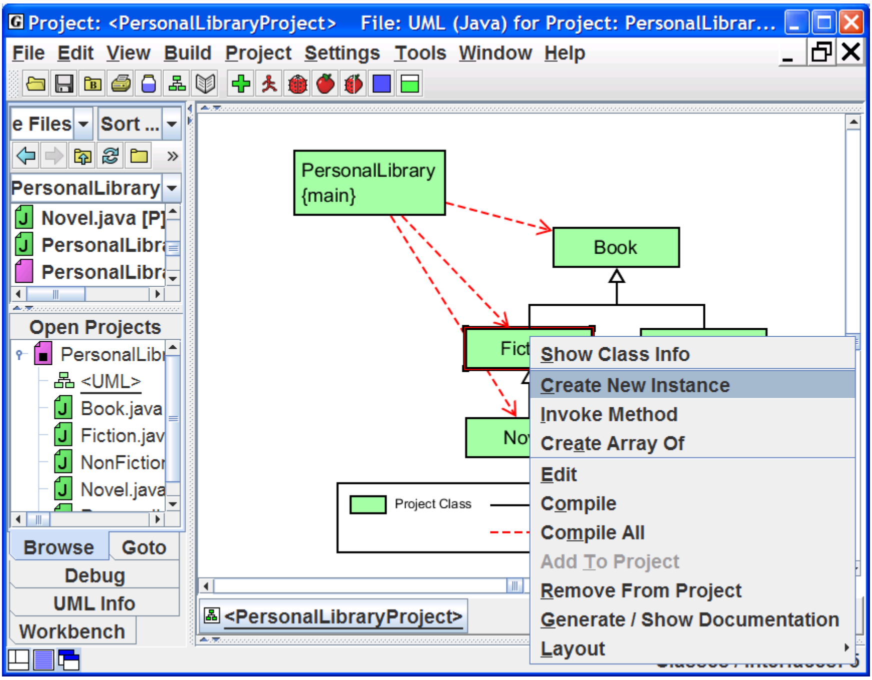


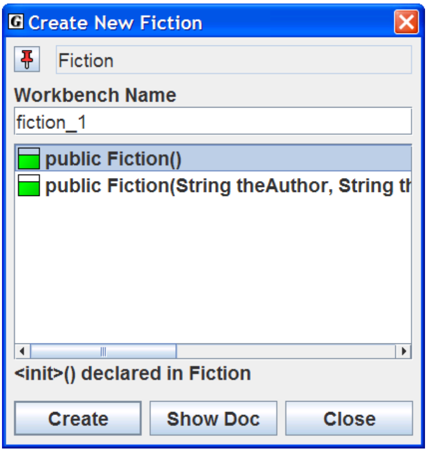
Figure 9-6. Creating an Object for the Workbench

Note: On any CSD or UML window, you can create an instance by clicking the Create Instance 🟦 *button on the tool bar (or* ***Build*** *>* ***Java Workbench*** *>* ***Create New Instance*** *on the main menu). Using the Create Instance* 🟦 *button from the CSD window is an easy way to create instances without the necessity of having a project or UML diagram.*

If a parameterless constructor is selected as shown in Figure 9-7, then clicking **Create** will immediately place the object on the workbench. However, if the constructor requires parameters, the dialog will expand to display the individual parameters as shown in Figure 9-8. The arguments (values of the parameters) should be filled in prior to clicking **Create**. Remember to enclose String arguments in double quotes.

In either case above, the user can set the name of the object being constructed or accept the default assigned by jGRASP. Also, the "stick-pin" located in the upper left of the dialog can be used to make the Create dialog "stay up" after you create an instance. This is handy for creating multiple instances of the same class. Click on the "stick-pin" (it should turn darker), then click the Create button three times and you should see three new instances appear on the workbench.

In Figure 9-9, the Workbench tab is shown after two instances (objects) of Fiction have been created. Notice that fiction_2 has been expanded so that its fields can be viewed: theAuthor, theTitle, thePages, theValue, theMainCharacter. Since the first three fields are instances of the String class, they too can be expanded. You should also note that mainCharacter is color coded green since it is the only field actually declared in Fiction. The other fields



Create New Fiction

Fiction

Workbench Name

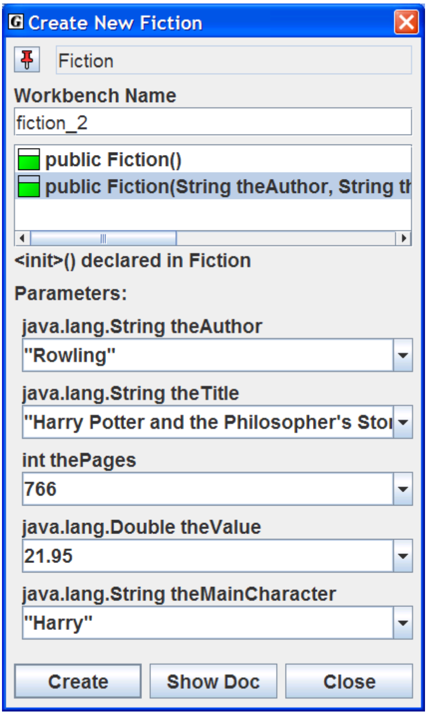
fiction_1

🟩 public Fiction()

🟩 public Fiction(String theAuthor, String th

<init>() declared in Fiction

Figure 9-7. Selecting a constructor



Create New Fiction

Fiction

Workbench Name

fiction_2

🟩 public Fiction()

🟩 public Fiction(String theAuthor, String th

<init>() declared in Fiction

Parameters:

java.lang.String theAuthor

"Rowling"

java.lang.String theTitle

"Harry Potter and the Philosopher's Stor

int thePages

766

java.lang.Double theValue

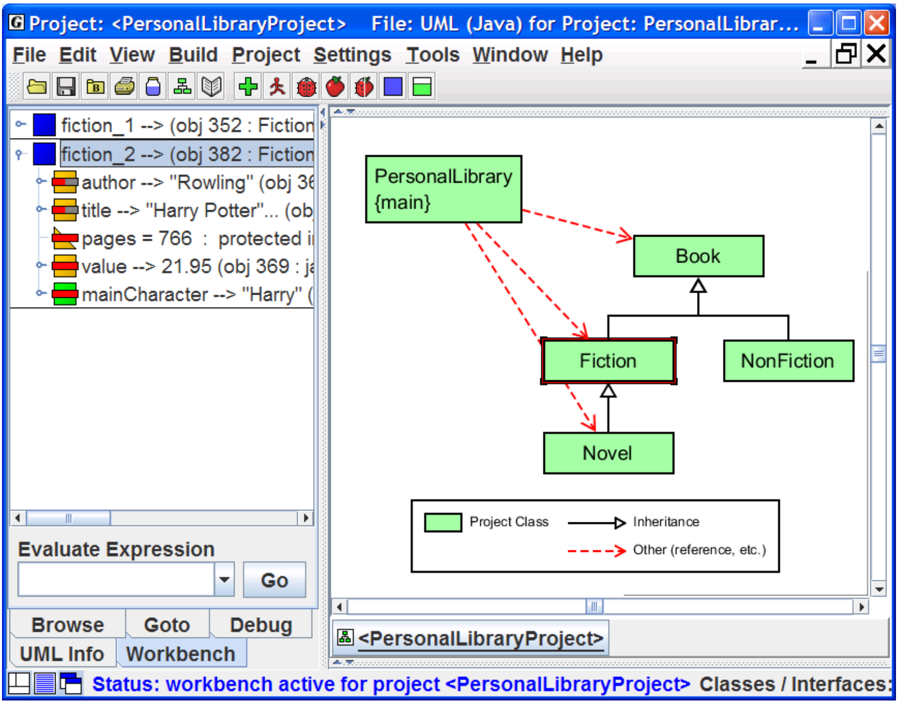
21.95

java.lang.String theMainCharacter

"Harry"

Figure 9-8. Constructor with parameters

are color coded orange to indicate they are inherited from a parent, which in this case is Book. The placement of these fields in Book vs. Fiction was a design decision. Since not all books have a mainCharacter (e.g., a math book) but works of fiction almost certainly do, mainCharacter was defined in Fiction. Notice that Novel, a subclass (or child) of Fiction, appropriately inherits mainCharacter.



Project: <PersonalLibraryProject> File: UML (Java) for Project: PersonalLibrar...

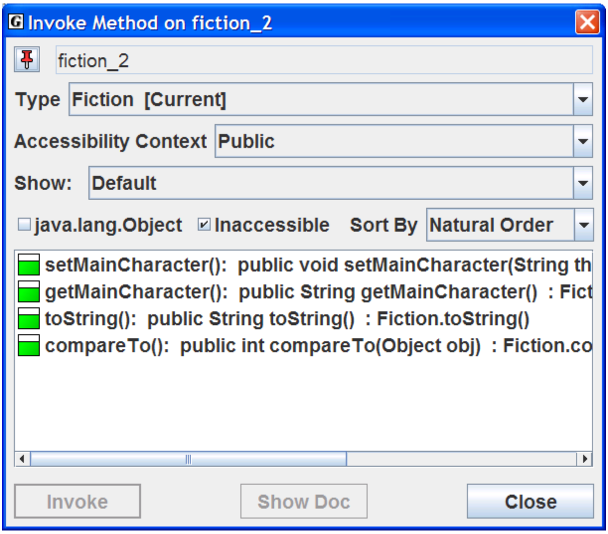
fiction_1 --> (obj 352 : Fiction
fiction_2 --> (obj 382 : Fiction
author --> "Rowling" (obj 36
title --> "Harry Potter"... (ob
pages = 766 : protected i
value --> 21.95 (obj 369 : ja
mainCharacter --> "Harry" (

Status: workbench active for project <PersonalLibraryProject> Classes / Interfaces:

Figure 9-9. Workbench with two instances of Fiction

9.4 Invoking a Method

To invoke a method for an object on the workbench, select the object, right click, and then select **Invoke Method**. In Figure 9-9, fiction_2 has been selected, followed by a right mouse click, and then Invoke Method has been selected. A list of user methods visible from Fiction will be displayed in a dialog box as shown in Figure 9-10. After one of the methods is selected and the parameters filled in as necessary, click **Invoke**. This will execute the method and display the return value (or void) in a dialog. Other output, if any, is handled in the usual way. If a method updates a field, as in the case of setMainCharacter(), the effect of the invocation is seen in the appropriate object field in the Workbench tab. The "stick-pin" located in the upper left of the dialog can be used to make the Invoke Method dialog stay up. This is useful for invoking multiple methods for the same object. For example, in a graphics program a "move" method could be clicked repeatedly to see an object move across the display.



Invoke Method on fiction_2

fiction_2

Type: Fiction [Current]

Accessibility Context: Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

```
🟩 setMainCharacter():  public void setMainCharacter(String th
🟩 getMainCharacter():  public String getMainCharacter()  : Fict
🟩 toString():  public String toString()  : Fiction.toString()
🟩 compareTo():  public int compareTo(Object obj)  : Fiction.co
```

Figure 9-10. Selecting a method

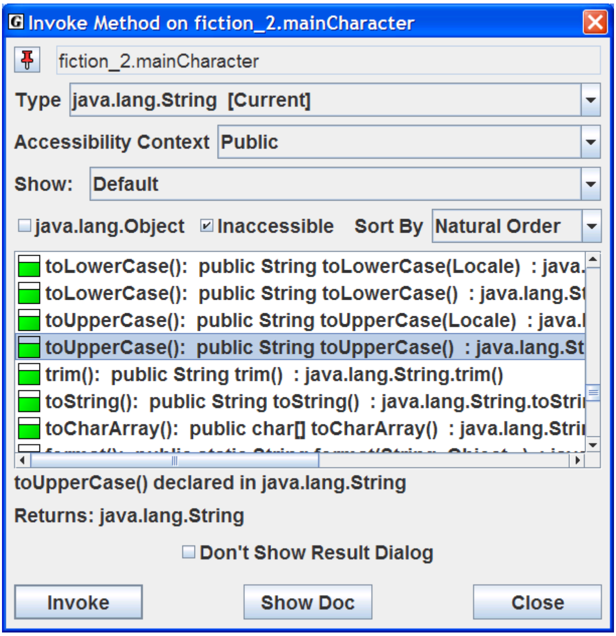
As indicated above, perhaps one of the most compelling reasons for using the workbench approach is that it allows the user to create an object and invoke each of its methods in isolation. Thus, with an instance of Fiction on the workbench, we can invoke each of its methods: setMainCharacter(), getMainCharacter(), toString(), and compareTo(). By reviewing the results of the method calls, we are essentially testing our class without a driver program.

9.5 Invoking Methods with Parameters Which Are Objects

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

9.6 Invoking Methods on Object Fields

If you have an object in the Workbench tab, you can expand it to reveal its fields. In Figure 9-9, fiction_2 is expanded to show its fields (author, title, pages, value, and mainCharacter). Since the field mainCharacter is itself an object of the class String, any of the String methods can be invoked on it. For example, right-click on mainCharacter in fiction_2, then select **Invoke Method**. When the dialog pops up (Figure 9-11), you'll see a rather lengthy list of all the methods visible to String objects. Scroll down the list and select the first *toUpperCase()* method, and then click **Invoke**. This should pop up the Result dialog with "HARRY" as the return value (Figure 9-12). 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.


Invoke Method on fiction_2.mainCharacter

fiction_2.mainCharacter

Type java.lang.String [Current]

Accessibility Context Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

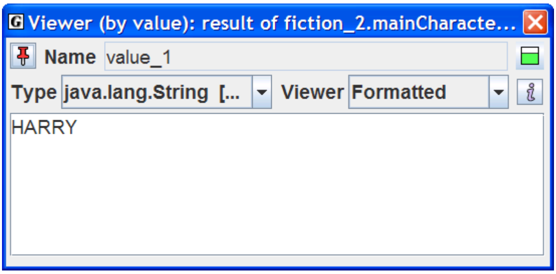
🟩 toLowerCase(): public String toLowerCase(Locale) : java.
🟩 toLowerCase(): public String toLowerCase() : java.lang.S
🟩 toUpperCase(): public String toUpperCase(Locale) : java.l
🟩 toUpperCase(): public String toUpperCase() : java.lang.St
🟩 trim(): public String trim() : java.lang.String.trim()
🟩 toString(): public String toString() : java.lang.String.toStri
🟩 toCharArray(): public char[] toCharArray() : java.lang.Stri

toUpperCase() declared in java.lang.String

Returns: java.lang.String

□ Don't Show Result Dialog

Figure 9-11. Invoking a String method


Viewer (by value): result of fiction_2.mainCharacte...

Name value_1

Type java.lang.String [... Viewer Formatted

HARRY

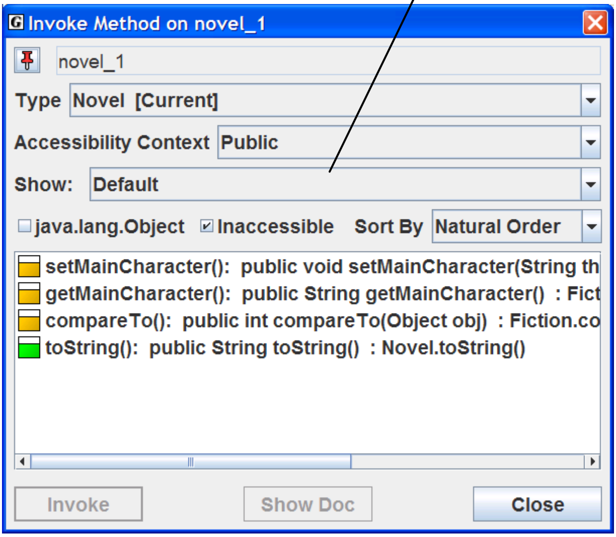
Figure 9-12. Result of invoking fiction_2.mainCharacter.toUpperCase()

9.7 Selecting Categories of Methods to Invoke

The Invoke Method dialog provides a list of categories of methods on a drop-down list. The “default” category list includes methods defined in the object’s class as well as those inherited from superclasses other than the Object class. This category was selected as the default so that *all* user defined methods could be conveniently viewed. In this section, we’ll explore the various categories of methods.

Let’s create an instance of Novel by right-clicking on Novel in the UML window and then selecting **Create New Instance**. On the Create dialog, choose the parameterless constructor and click **Create**. Now you should see novel_1 on the workbench. Right-clicking on novel_1 and then selecting the Invoke Method will open the Invoke Method dialog as shown in Figure 9-13. Notice that the first two methods are inherited (gold method symbols) and the third is defined in Novel (green method symbol). Now look back at the Invoke Method dialog for fiction_2 in Figure 9-10. The same methods are listed, but all are marked with green method symbols since those are defined in the Fiction class. One should surmise from this that both Fiction and Novel must have their own *toString* method.

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



Invoke Method on novel_1

novel_1

Type Novel [Current]

Accessibility Context Public

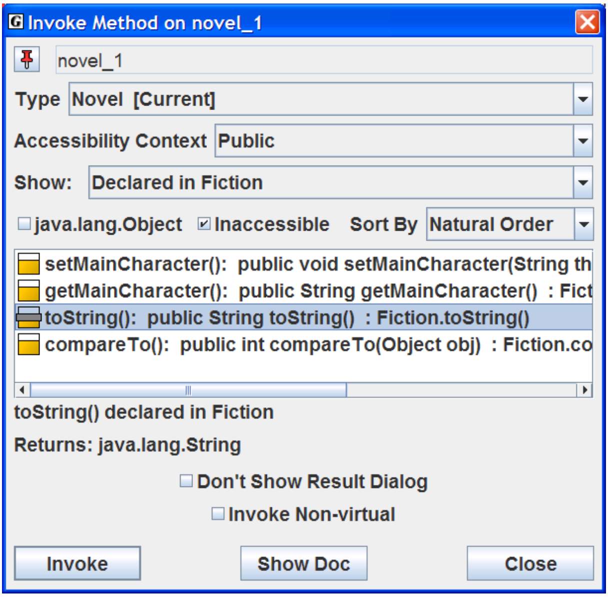
Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

🟨 setMainCharacter(): public void setMainCharacter(String th
🟨 getMainCharacter(): public String getMainCharacter() : Fict
🟨 compareTo(): public int compareTo(Object obj) : Fiction.co
🟩 toString(): public String toString() : Novel.toString()

Figure 9-13. Invoking a method for novel_1

Now let's look at another category of method on the Invoke Method dialog for novel_1. Click the drop-down list on the dialog (see info box for Figure 9-13) and select "Declared in Fiction". Notice that the *toString* method in Figure 9-14 has a gray bar through its gold method symbol to indicate that it has been overridden by the *toString()* method defined for Novel. This means that if you select and invoke the *toString* method listed in Figure 9-14, the *toString* defined in Novel will be the one that gets called. Remember, it is the object itself that determines which method is called. In your Java program, if you wanted to call an overridden method for an object, you would need to call the method non-virtually. jGRASP provides a short cut for doing this on the workbench with the "Invoke Non-virtual" check box on the dialog. In the example in Figure 9-14, if you invoke the *toString* method without checking the box for Invoke Non-virtual, Novel's *toString* method is called, and you get the result shown in Figure 9-15. However, if you invoke the method with the box checked, Fiction's *toString* method is called, and you get the result in Figure 9-16. Notice that the only difference is that Novel's *toString* method includes one more line of text ("Number of sequels: 0") than Fiction's *toString* method.



Invoke Method on novel_1

novel_1

Type Novel [Current]

Accessibility Context Public

Show: Declared in Fiction

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

setMainCharacter(): public void setMainCharacter(String th
getMainCharacter(): public String getMainCharacter() : Fict
toString(): public String toString() : Fiction.toString()
compareTo(): public int compareTo(Object obj) : Fiction.co

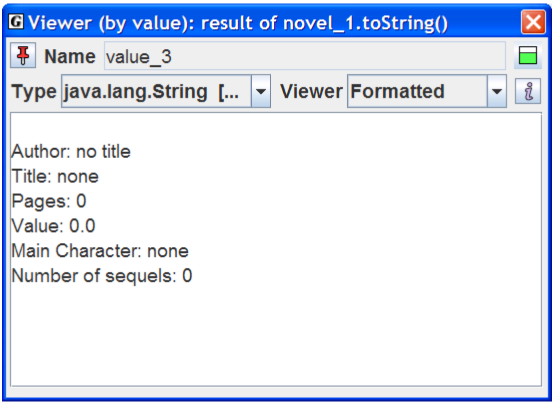
toString() declared in Fiction

Returns: java.lang.String

□ Don't Show Result Dialog

□ Invoke Non-virtual

Figure 9-14 Methods declared in superclass Fiction



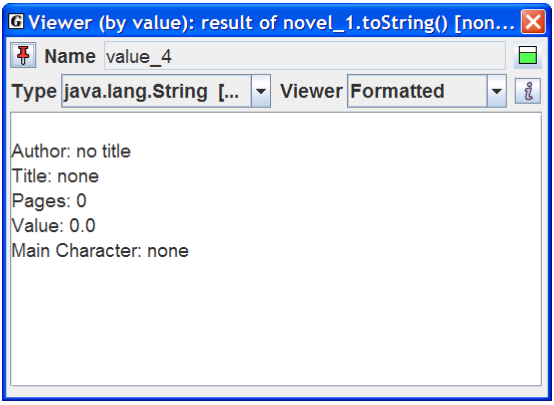
Viewer (by value): result of novel_1.toString()

Name value_3

Type java.lang.String [... Viewer Formatted

```
Author: no title
Title: none
Pages: 0
Value: 0.0
Main Character: none
Number of sequels: 0
```

Figure 9-15. Viewing superclasses for novel_1



Viewer (by value): result of novel_1.toString() [non…

Name value_4

Type java.lang.String [... Viewer Formatted

```
Author: no title
Title: none
Pages: 0
Value: 0.0
Main Character: none
```

Figure 9-16. Viewing superclasses for novel_1

Two other check boxes (“java.lang.Object” and “Inaccessible Methods”) are located below **Show** and above the method list. The first includes the methods inherited from the Object class along with the other methods in the selected category. The second can be used to display inaccessible methods such as inherited private methods.

To wrap up this section, you are invited to select among the other **categories** of methods that can be displayed on the Invoke Method dialog for novel_1: **Default**, **All**, **Visible**, **Declared in java.lang.Object**, **Delcared in Book**, **Declared in Fiction**, **Declared in Novel**, **Declared in java.lang.Comparable**.

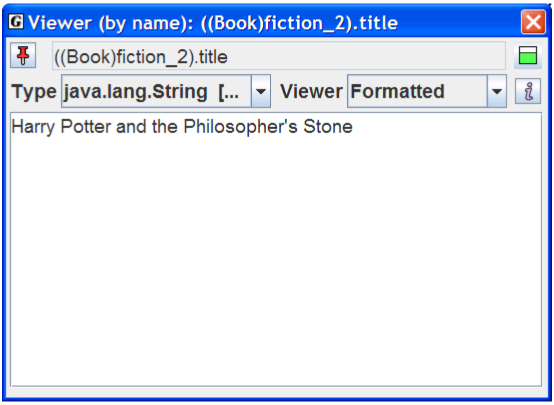
Notice that novel_1 inherits a large number of methods from java.lang.Object. The most inclusive category is "All" which includes all available methods. Perhaps now you see why the default category does not show all methods.

9.8 Opening Object Viewers

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

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

Figure 9-17 shows an object viewer for the *title* field of fiction_2 which is a String object in an instance of Novel. *Formatted* is the default "view" for a String object which is especially useful when viewing a String object with a large value (e.g., a page of text). In Figure 9-18, the *Basic* view has been selected and expanded to show the gory details of the String object. Notice that the first field is value which is a character array (char[40]) holding the actual value of the string. If we open a separate viewer on value, we have a nice *Presentation* view of the character array. This is the same view you get when a *Presentation* view for String title is opened as shown in Figure 9-19. In the *Viewers for Data Structures* tutorial, additional Presentation views will be discussed. You are encouraged to open separate viewers for the objects on the workbench. In addition to providing multiple views of the object, each viewer includes an Invoke Method button 🟩 for the object being viewed.


Viewer (by name): ((Book)fiction_2).title

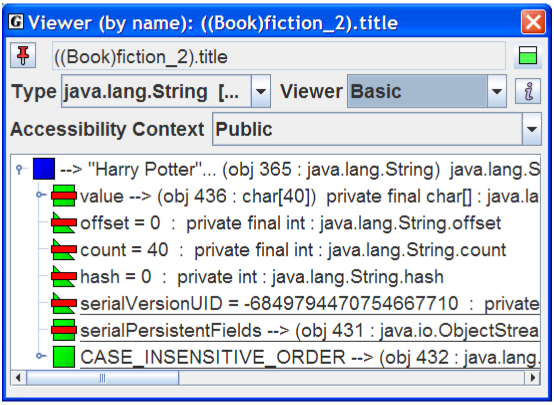
((Book)fiction_2).title

Type java.lang.String [...

Viewer Formatted

Harry Potter and the Philosopher's Stone

Figure 9-17. Viewing a String Object



Viewer (by name): ((Book)fiction_2).title

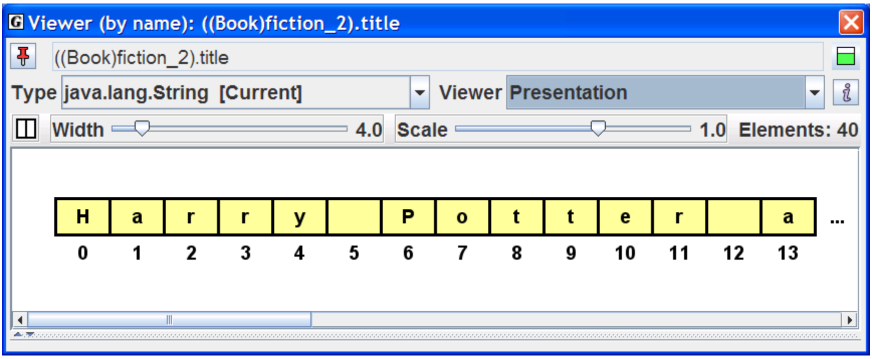
((Book)fiction_2).title

Type java.lang.String [... Viewer Basic

Accessibility Context Public

```
--> "Harry Potter"... (obj 365 : java.lang.String)  java.lang.S
   value --> (obj 436 : char[40])  private final char[] : java.la
   offset = 0  :  private final int : java.lang.String.offset
   count = 40  :  private final int : java.lang.String.count
   hash = 0  :  private int : java.lang.String.hash
   serialVersionUID = -6849794470754667710  :  private
   serialPersistentFields --> (obj 431 : java.io.ObjectStrea
   CASE_INSENSITIVE_ORDER --> (obj 432 : java.lang.
```

Figure 9-18. Basic view of a string (expanded to see fields)



Viewer (by name): ((Book)fiction_2).title

((Book)fiction_2).title

Type java.lang.String [Current] Viewer Presentation

Width 4.0 Scale 1.0 Elements: 40

H	a	r	r	y		P	o	t	t	e	r		a	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	

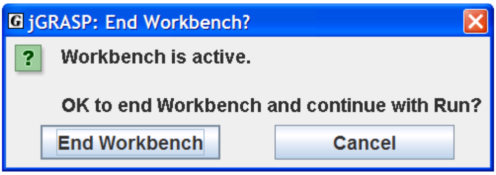
Figure 9-19. Presentation view of fiction_2.title

9.9 Running the Debugger on Invoked Methods

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

9.10 Exiting the Workbench

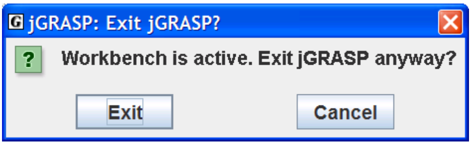
The workbench is *running* whenever you have objects on it. If you attempt to do an operation that conflicts with workbench (e.g., recompile a class, switch projects, etc., jGRASP will prompt you with a message indicating that the workbench process is active and ask you if it is OK to end the process (Figure 9-20). When you try to exit jGRASP, you will get a similar message (Figure 9-21). 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**.


jGRASP: End Workbench?

Workbench is active.

OK to end Workbench and continue with Run?

Figure 9-20. Making sure it is okay to exit the Workbench


jGRASP: Exit jGRASP?

Workbench is active. Exit jGRASP anyway?

Figure 9-21. Making sure it is okay to exit

Notes

10 Viewers for Data Structures

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

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

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

10.1 Introduction

10.2 Opening Viewers

10.3 Setting the View Options

10.4 Selecting Among Views

10.5 Presentation Views for LinkedList, HashMap, and TreeMap

10.6 Presentation Views for Code Understanding

10.6.1 LinkedListExample.java

10.6.2 BinaryTreeExample.java

10.6.3 Configuring Views generated by the *Structure Identifier*

10.7 Using the Viewers from the Workbench

10.8 Summary of Views

10.9 Exercises

10.1 Introduction

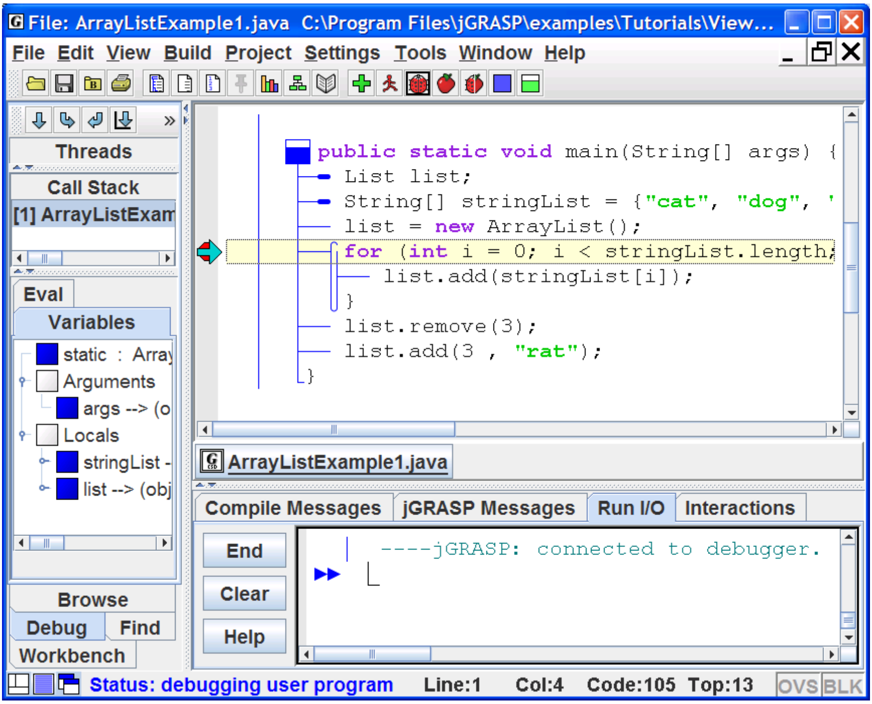
jGRASP viewers are tightly integrated with the workbench and debugger. They can be opened for any primitive, object, or field of an object in the Debug or Workbench tabs. To use viewers with the debugger, (1) set a breakpoint in your program, (2) run Debug (🐞), (3) after a local variable has been created, drag it from the Debug tab, (4) step through program and observe the object in the viewer. To use viewers with the workbench, (1) create an instance from the UML window, the CSD window (🟦), or Interactions tab, (2) drag the instance from the Workbench tab, (3) invoke methods on the instance and observe the object in the viewer.

Note that once an instance is in the workbench tab or debug tab, its methods can be invoked via the Invoke Method dialog or by entering Java statements and/or expressions in the Interaction tab. When methods are invoked on the instance, any open viewers on it are updated as appropriate.

jGRASP includes a general view for data structures called *Presentation – Structure Identifier (SI)* which automatically detects linked lists, binary trees, and array wrappers (lists, stacks, queues, etc.) when a viewer is opened on one of these during debugging or workbench use. For linked structures, this is an animated view that shows nodes being added and deleted from the data structure. This view is also configurable with respect to the structure mappings and the fields to display. jGRASP also includes custom *Presentation* views for many of the classes in the Java Collections Framework (e.g., ArrayList, Stack, LinkedList, TreeMap, and HashMap). These non-animated views are optimized for large numbers of elements.

10.2 Opening Viewers

Let's begin by opening one of the example programs that comes with the jGRASP installation. After you have started jGRASP, use the Browse tab to navigate to the jGRASP\examples\Tutorials folder. If you have been working with the examples in the "Hello" or "PersonalLibrary" folders, you'll need to go up one level in the Browse tab by clicking the up arrow. [Note that you should copy this folder to a personal directory. This will need to be done using a file browser rather than jGRASP.] In the Tutorials folder you should find a folder called ViewerExamples. Open this folder by double-clicking on the folder name, and you should see a file called *ArrayListExample1.java*. Open this file by double-clicking the file name (Figure 10-1).



```
public static void main(String[] args) {
   List list;
   String[] stringList = {"cat", "dog", '
   list = new ArrayList();
   for (int i = 0; i < stringList.length;
      list.add(stringList[i]);
   }
   list.remove(3);
   list.add(3 , "rat");
}
```

```
----jGRASP: connected to debugger.
```

Figure 10-1. ***ArrayListExample1.java***

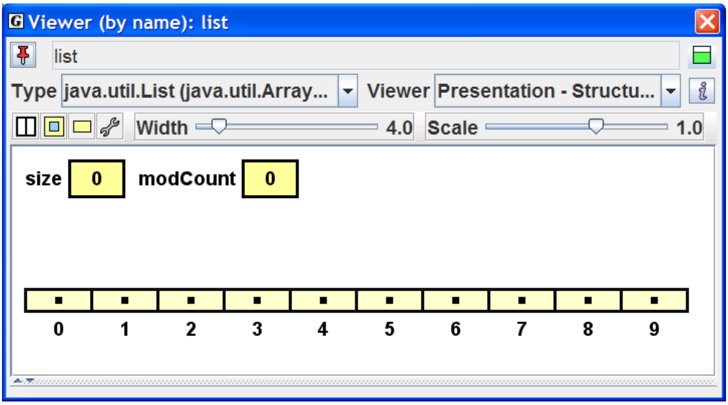
A quick review of the program shows that it creates an ArrayList called *list* and adds strings to it from an array called *stringList*. Compile the program by clicking the green plus ✚. Since the viewers are for visualizing objects and primitives as the program executes, let's set a breakpoint on the first line of the *for* statement. To do this, move the mouse over the left margin next to the statement until you see the breakpoint symbol ● and then left-click. You should see the ● in the margin if you have successfully set the breakpoint. Now start the debugger by clicking 🐞 on the toolbar. Figure 10-1 shows the program stopped at the *for* statement. At this point in the program, the *list* object has been created, and it is shown in the Debug Variables tab. However, no elements have been added to *list*.

A separate ***Viewer*** window can be opened for any object (or field of an object) in the debug tab (or on the workbench). The easiest way to open a viewer is to left-click on an object and drag it from the debug tab (or workbench) to the location where you want the viewer to open. When you start to drag the object, a viewer symbol should appear to indicate a viewer is being opened. [Note: You can also open a viewer by right-clicking on the *list* object and selecting either

***View by Name** or **View Value**.*] Let's left click on *list* and drag it from the Debug tab. When you release the left mouse button, the viewer should open.

Figure 10-2 shows a viewer opened on *list* before any elements have been added. Note that the default *View* for an instance of ArrayList is *Presentation – Structure Identifier*. This view shows the fields for *size* and *modCount* along with the underlying array with its default size of 10.

To add elements to *list*, step through the program by clicking the "Step" button ⇩ on the Debug tab. Since the viewer is updated on each step, you should see the elements being added to the list. Red text indicates a change or, in this case, a new element. Figure 10-3 shows the view of *list* after going through the loop three times. As you continue to step through the program, notice that when elements are removed, the value stays in the array but the *size* is decremented.



Viewer (by name): list

list

Type java.util.List (java.util.Array...

Viewer Presentation - Structu...

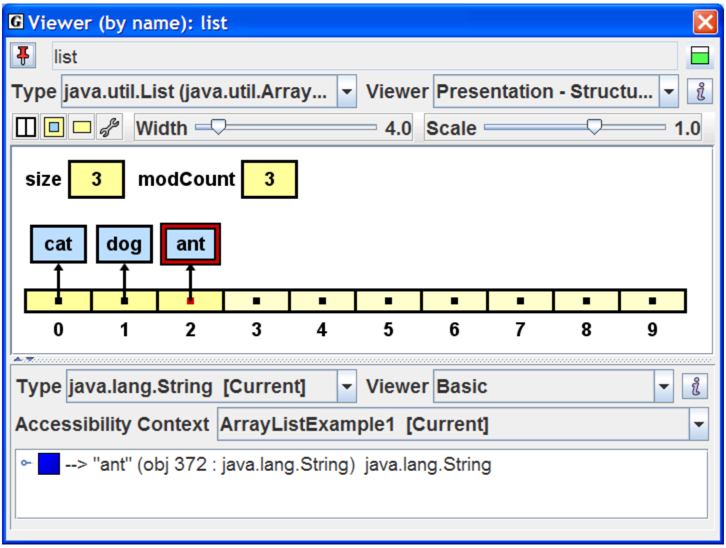
Width 4.0

Scale 1.0

size 0 modCount 0

Figure 10-2. View of *list* with no elements

Each jGRASP Presentation viewer provides one or more subviews. When an element is selected as indicated by a red border, a view of the element itself is shown in the subview. Figure 10-3 shows "ant" selected in the ArrayList view. Since "ant" is a String, the subview is a String viewer for which the *formatted* view is the default.



Viewer (by name): list

list

Type java.util.List (java.util.Array... Viewer Presentation - Structu...

Width 4.0 Scale 1.0

size 3 modCount 3

Type java.lang.String [Current] Viewer Basic

Accessibility Context ArrayListExample1 [Current]

--> "ant" (obj 372 : java.lang.String) java.lang.String

Figure 10-3. View of *list* with 3 elements

10.3 Setting the View Options

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

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

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

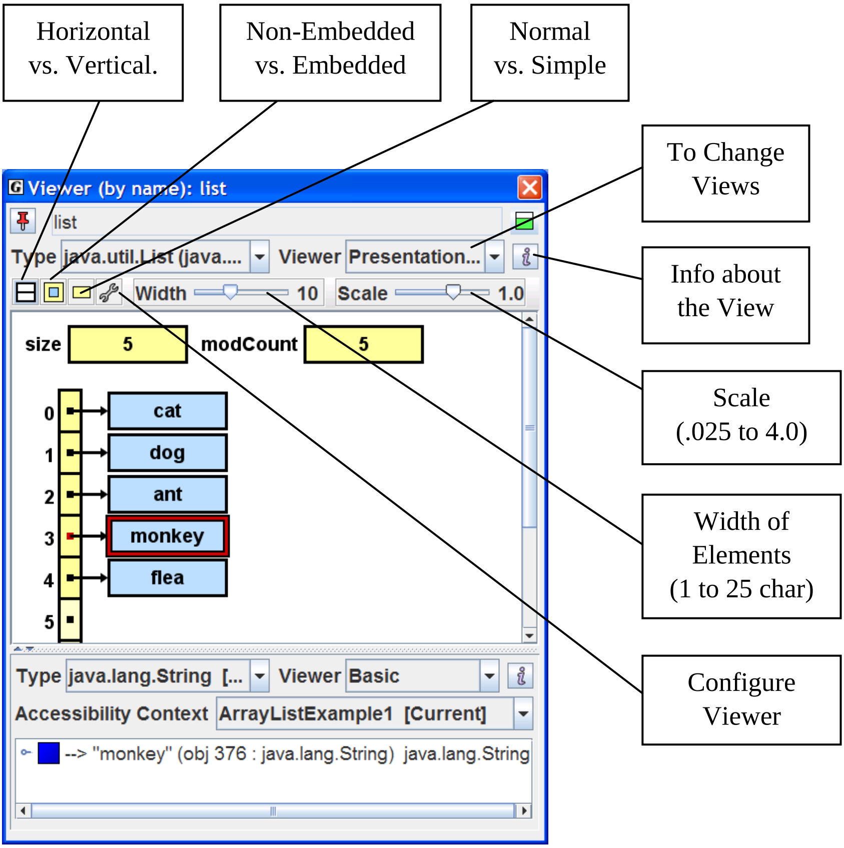
Normal vs. Simple – shows node pointer from inside or from edge of structure.

Configure View – opens dialog to configure the structure-to-view mapping as well as which fields to display in the viewer (discussed in Section 9.5).

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

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

Figure 10-4 indicates the location of the buttons and sliders for each view option. Click on each of these and notice the change in the view. The ArrayList is shown vertically after the display orientation is changed. The location of the **View** drop down list and the Information button ℹ is also indicated below.



Viewer (by name): list

list

Type java.util.List (java.... Viewer Presentation...

Width 10 Scale 1.0

size 5 modCount 5

0 cat
1 dog
2 ant
3 monkey
4 flea
5

Type java.lang.String [... Viewer Basic

Accessibility Context ArrayListExample1 [Current]

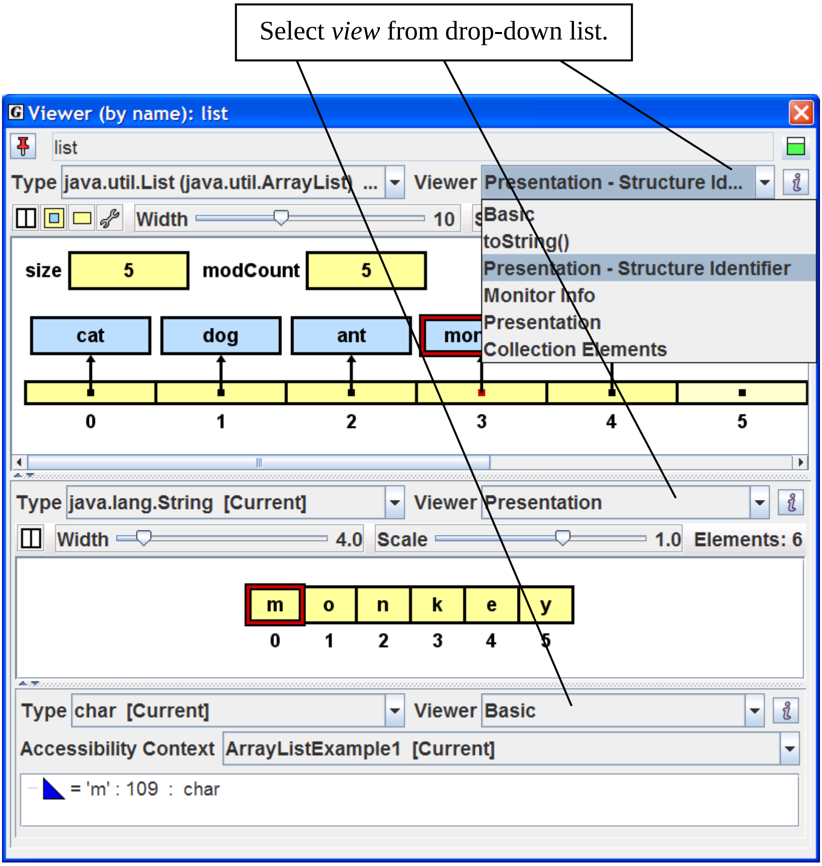
```
--> "monkey" (obj 376 : java.lang.String)  java.lang.String
```

Figure 10-4. View of *list* in vertical mode, width set to 8, Scale set to 1.2, *monkey* selected and shown in subview

10.4 Selecting Among Views

Each viewer and subviewer provides one or more views among which you may select from the respective *View* drop down lists. Let’s take a closer look at our ArrayList of Strings example. The *Presentation – Structure Identifier* view is the default for ArrayList and the other classes in the Java Collections Framework. Other views include *Basic*, *toString()*, *Presentation*, and *Collection Elements*.

Figure 10-5 shows the view options for ArrayList on the drop-down list (combo box) with *Presentation – Structure Identifier* view selected. When *monkey* is selected in the ArrayList, a String subview is opened. When this view is set to *Presentation* view, the character array for *monkey* is displayed. Selecting the first element in the array opens a subview for character *m* in monkey which is set to the default *Basic* view.



Select *view* from drop-down list.

Viewer (by name): list

list

Type java.util.List (java.util.ArrayList) ... Viewer Presentation - Structure Id...

Width 10

size 5 modCount 5

Type java.lang.String [Current] Viewer Presentation

Width 4.0 Scale 1.0 Elements: 6

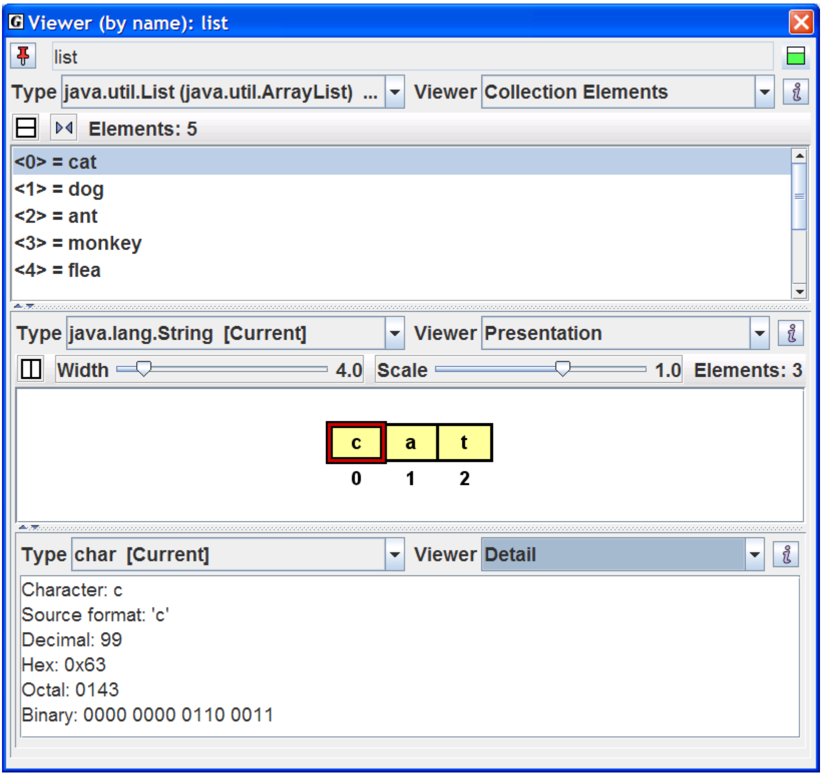
Type char [Current] Viewer Basic

Accessibility Context ArrayListExample1 [Current]

= 'm' : 109 : char

Figure 10-5. Selecting the *Collection Elements*

Figure 10-6 shows the viewer after the *Collection Elements* view is selected. If *list* has many elements, this may be a more appropriate view than the *Presentation* view. The *Collection Elements* view was specifically designed to handle larger numbers of elements efficiently. As the number of elements increases, additional navigational controls appear on the viewer for moving about in the ArrayList. Notice that two subviews are also shown in Figure 10-6. When element 0 (indicated by "**<0> = cat**") is selected in the *Collection Elements* view, a subview for String opens below the main view. Notice that he view for String has been set to *Presentation* in the figure; the default for String is *Formatted*. When the 'c' in "cat" is selected, a second subview is opened for the primitive type *char*, for which *Basic* is the default view. However, in the figure, the view has been set to *Detail*, which displays additional information about 'c' including its value in hexadecimal, octal, and binary.


Viewer (by name): list
list
Type java.util.List (java.util.ArrayList) ...
Viewer Collection Elements
Elements: 5

<0> = cat
<1> = dog
<2> = ant
<3> = monkey
<4> = flea

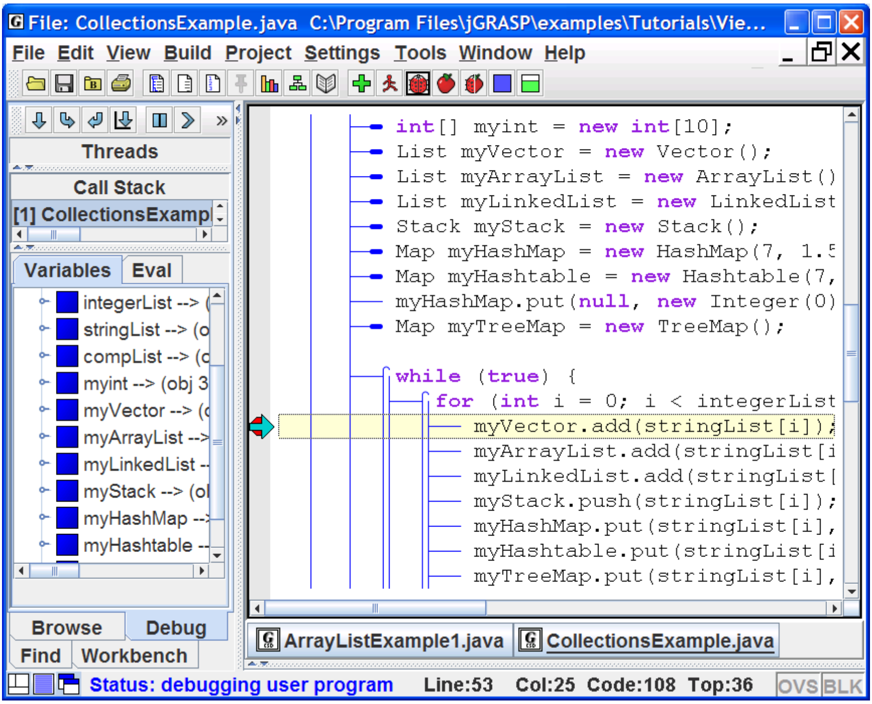
Character: c
Source format: 'c'
Decimal: 99
Hex: 0x63
Octal: 0143
Binary: 0000 0000 0110 0011

Figure 10-6. The *Collection Elements* view of *list* with two subviews: *Presentation* view for String "cat" and *Detail* view for *char* 'c'

10.5 Presentation Views for LinkedList, HashMap, and TreeMap

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

In the Browse tab, locate CollectionsExample.java, and double-click on it to open it in the CSD window. Compile the program by clicking the green plus ✚. Set a breakpoint on any executable statement in the program. Now start the debugger by clicking 🐞. Figure 10-7 shows the program stopped at a breakpoint on the line in the inner loop that adds an element to myVector. Notice that prior to the breakpoint, the variables myVector, myArrayList, myLinkedList, myStack, myHashMap, and myTreeMap were declared and their respective instances were created. With the program stopped at the breakpoint, we can open viewers for each of the variables listed in the Debug tab.


File: CollectionsExample.java C:\Program Files\jGRASP\examples\Tutorials\Vie...
File Edit View Build Project Settings Tools Window Help
Threads
Call Stack
[1] CollectionsExamp
Variables
Eval
Browse
Debug
Find
Workbench
ArrayListExample1.java
CollectionsExample.java
OVS BLK

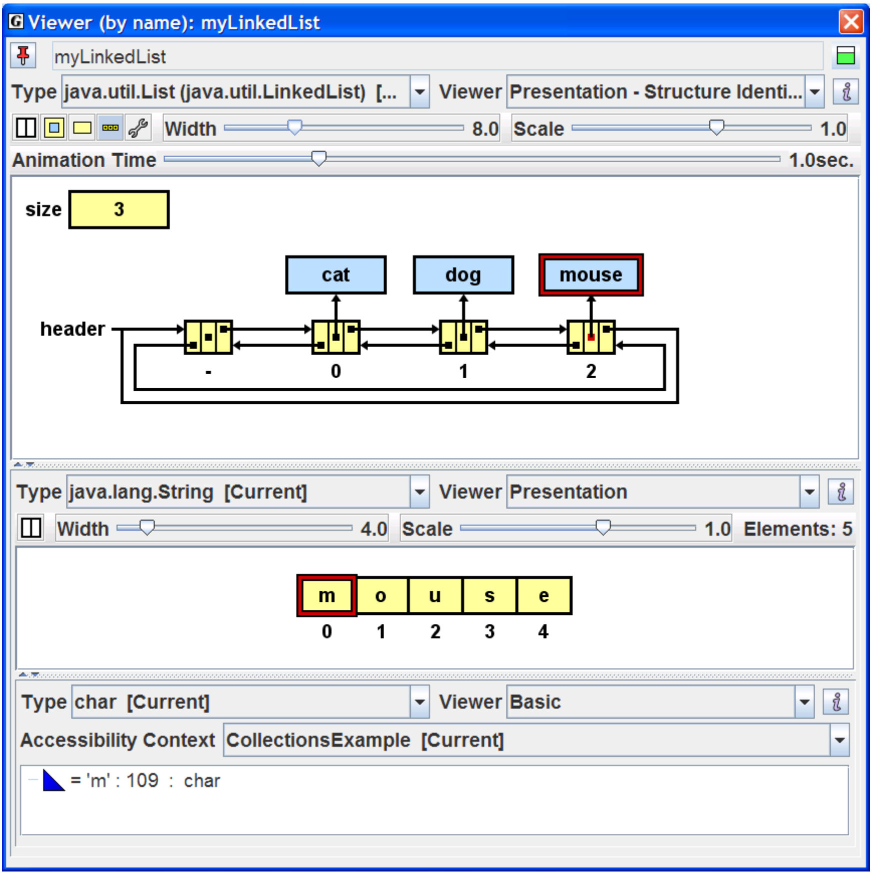
integerList --> (
stringList --> (o
compList --> (c
myint --> (obj 3
myVector --> (c
myArrayList -->
myLinkedList -
myStack --> (ol
myHashMap --
myHashtable --

```
      int[] myint = new int[10];
      List myVector = new Vector();
      List myArrayList = new ArrayList()
      List myLinkedList = new LinkedList
      Stack myStack = new Stack();
      Map myHashMap = new HashMap(7, 1.5
      Map myHashtable = new Hashtable(7,
      myHashMap.put(null, new Integer(0)
      Map myTreeMap = new TreeMap();

      while (true) {
         for (int i = 0; i < integerList
            myVector.add(stringList[i]);
            myArrayList.add(stringList[i
            myLinkedList.add(stringList[
            myStack.push(stringList[i]);
            myHashMap.put(stringList[i],
            myHashtable.put(stringList[i
            myTreeMap.put(stringList[i],
```

Status: debugging user program Line:53 Col:25 Code:108 Top:36

Figure 10-7. CollectionsExample.java stopped at a breakpoint

Figure 10-8 shows a viewer set to *Presentation - Structure Identifier* view opened on myLinkedList after three elements have been added to it. Notice that myLinkedList is a doubly-linked list with a header node. Width has been set to 8.0, and the element *mouse* is selected in the main view and shown in the subview in *Presentation* view for the String Class. In this view, the *m* in *mouse* is selected, and the character subview is shown set to the *Basic* view.



Viewer (by name): myLinkedList

myLinkedList

Type java.util.List (java.util.LinkedList) [... Viewer Presentation - Structure Identi...

Width 8.0 Scale 1.0

Animation Time 1.0sec.

Type java.lang.String [Current] Viewer Presentation

Width 4.0 Scale 1.0 Elements: 5

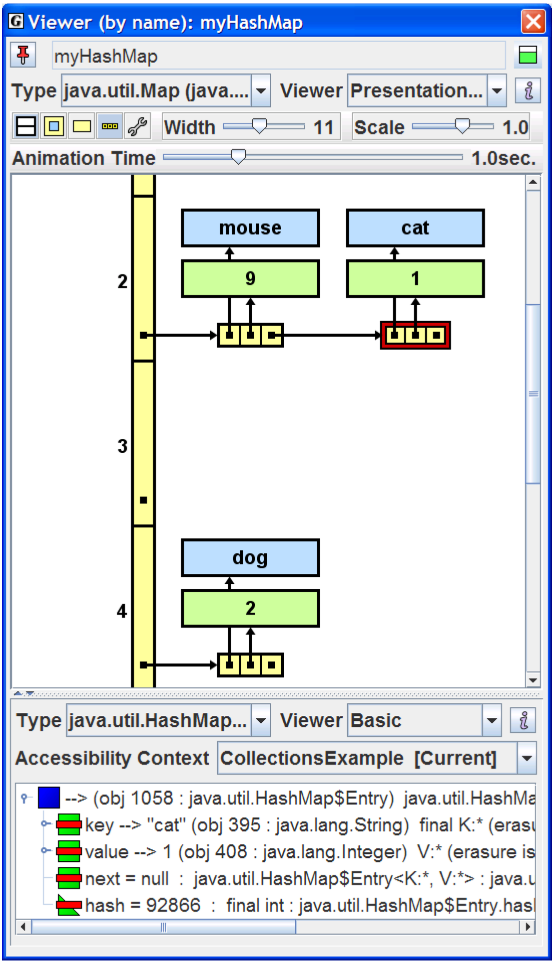
Type char [Current] Viewer Basic

Accessibility Context CollectionsExample [Current]

= 'm' : 109 : char

Figure 10-8. View of *myLinkedList* after three elements have been added

Figure 10-9 shows a viewer set to *Presentation - Structure Identifier* view opened on the variable myHashMap after three elements have been added. A hashmap entry is selected, as indicated by the red border, and its *Basic* view is shown in the subview with fields: *key*, *value*, *hash*, and *next*. As elements are added to the HashMap, it is useful to use the Scale slider to zoom in and out on the structure so that the "topology" of its elements can be seen.


Viewer (by name): myHashMap

myHashMap

Type java.util.Map (java.... Viewer Presentation...

Width 11 Scale 1.0

Animation Time 1.0sec.

Type java.util.HashMap... Viewer Basic

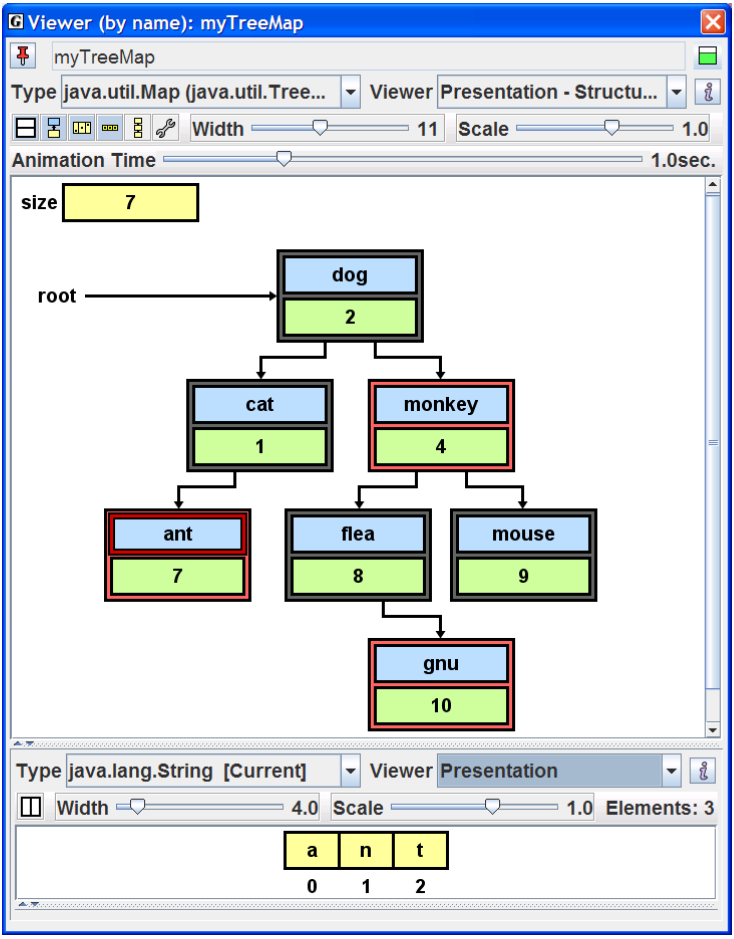
Accessibility Context CollectionsExample [Current]

```
--> (obj 1058 : java.util.HashMap$Entry) java.util.HashMa
    key --> "cat" (obj 395 : java.lang.String) final K:* (erasu
    value --> 1 (obj 408 : java.lang.Integer) V:* (erasure is
    next = null : java.util.HashMap$Entry<K:*, V:*> : java.u
    hash = 92866 : final int : java.util.HashMap$Entry.hasl
```

Figure 10-9. View of *myHashMap* after three elements have been added

Figure 10-10 shows a viewer opened on myTreeMap after seven elements have been added. TreeMap uses a *Red-Black* tree as its underlying storage structure, and the default *Presentation – Structure Identifier* view indicates the red and black nodes by coloring their borders light red and dark gray respectively. As you step through the program and put items in the TreeMap, you should see the red-black node rotations.

In the figure, width has been set to 11.0, and in the red node containing "ant", the key field "ant" has been selected as indicated by an additional dark red border. The String subview for *ant* is set to *Presentation*.



Viewer (by name): myTreeMap

myTreeMap

Type java.util.Map (java.util.Tree... Viewer Presentation - Structu...

Width 11 Scale 1.0

Animation Time 1.0sec.

size 7

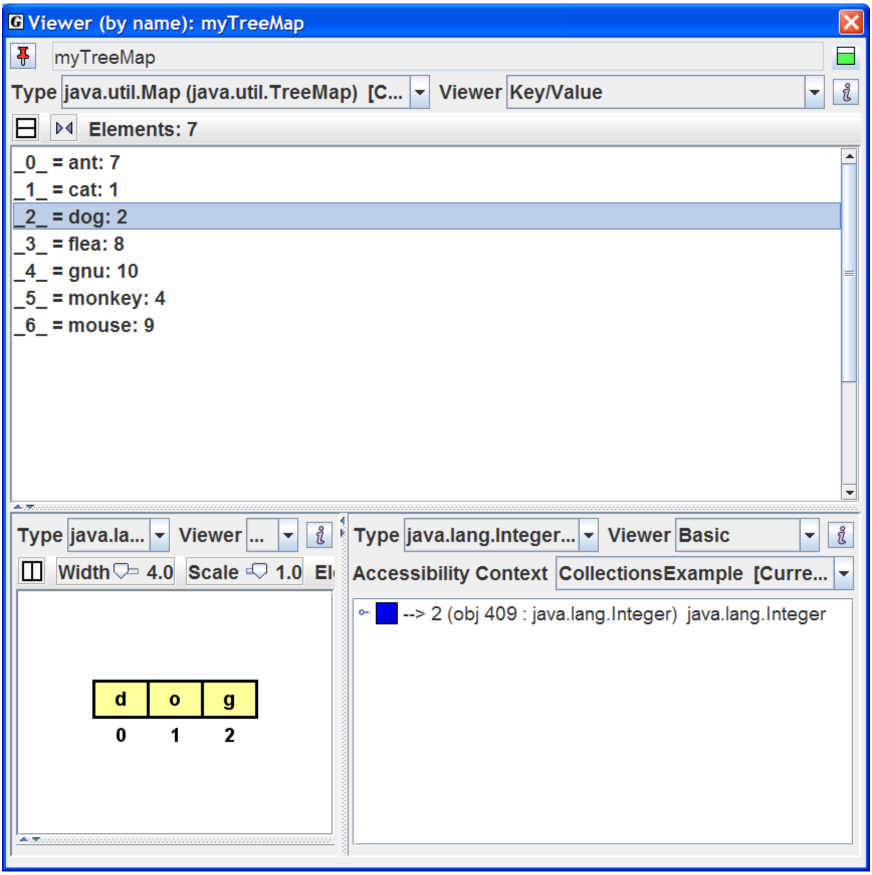
Type java.lang.String [Current] Viewer Presentation

Width 4.0 Scale 1.0 Elements: 3

Figure 10-10. *Presentation - Structure Identifier* View of myTreeMap with six elements

Figure 10-11 shows a second viewer opened on myTreeMap with the view set to *Key/Value*. The node for *dog* has been selected and two subviews have been opened: one for the *key* and one for the *value*. In the figure below, the node with *key* = “dog” and *value* = 2 has been selected.

In the left subview for String *key*, the view is set to *Presentation* as it was in the previous figure. In the right subview, we have the *Basic* view of v*alue* which is an object; specifically, it is an instance of java.lang.Integer, the wrapper class for the Java primitive *int*.



Viewer (by name): myTreeMap

myTreeMap

Type java.util.Map (java.util.TreeMap) [C...

Viewer Key/Value

Elements: 7

```
_0_ = ant: 7
_1_ = cat: 1
_2_ = dog: 2
_3_ = flea: 8
_4_ = gnu: 10
_5_ = monkey: 4
_6_ = mouse: 9
```

Type java.la... Viewer ...

Width 4.0 Scale 1.0 El

d	o	g
0	1	2

Type java.lang.Integer... Viewer Basic

Accessibility Context CollectionsExample [Curre...

```
--> 2 (obj 409 : java.lang.Integer)  java.lang.Integer
```

Figure 10-11. ***Key/Value*** **view of myTreeMap with seven elements**

10.6 Presentation Views for Code Understanding

Now we turn our attention to the details of the *Presentation - Structure Identifier* viewer when it is used in conjunction with user classes for data structures including most textbook examples. When this viewer is opened on an object, it automatically attempts to determine if the object represents a common data structure; if so, it verifies relevant links, displays nodes referenced by local variables, and provides animation for the insertion and deletion of nodes. The structure mappings that are determined by the viewer and the fields that are displayed in the view can be configured by the user while the viewer is open on the object.

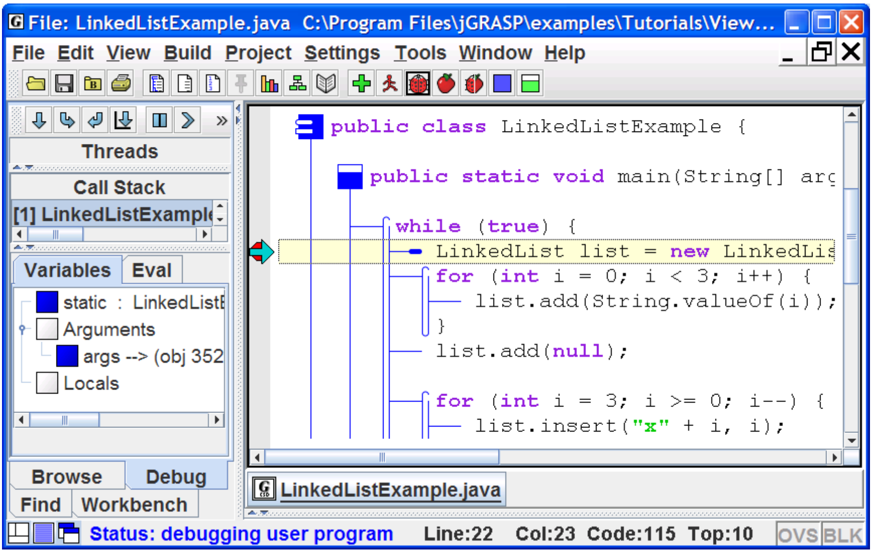
Custom *Presentation* views (as opposed to the more general *Presentation - Structure Identifier* view) are available for many of the Java Collections Framework classes. Each of these views is generated by a non-verifying viewer implemented specifically for the respective class. Because these viewers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

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

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

10.6.1 LinkedListExample.java

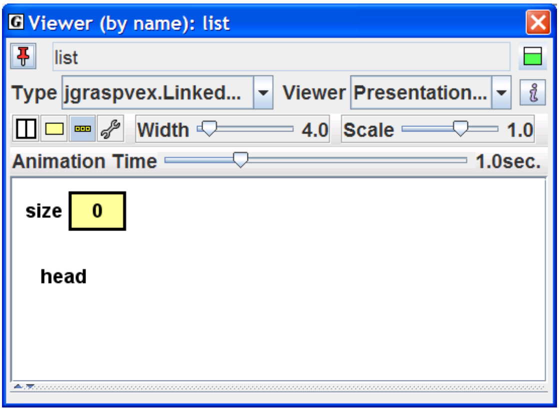
In the Browse tab, navigate to the *ViewerExamples* directory and open the file LinkedListExample.java by double-clicking on it. Generate the CSD, and then compile the program by clicking ✚ on the toolbar. Set a breakpoint ● in the left margin on a line inside the inner loop (e.g., on the line where *list* is declared and a new *LinkedList* object is created). Now click the Debug button 🐞 on the toolbar. Figure 10-11 shows the program after it has stopped at the breakpoint prior to creating an instance of *LinkedList* called *list*. Click Step ⇩ on the controls at the top of the Debug tab. When *list* is created, you should see it in the Variables tab of the Debug window.



```
public class LinkedListExample {

   public static void main(String[] arg

      while (true) {
         LinkedList list = new LinkedLis
         for (int i = 0; i < 3; i++) {
            list.add(String.valueOf(i));
         }
         list.add(null);

         for (int i = 3; i >= 0; i--) {
            list.insert("x" + i, i);
```

Figure 10-11. LinkedListExample.java stopped at a breakpoint

Now open a viewer on *list* by selecting and dragging *list* from the Debug window. Figure 10-12 shows a view of *list* before any elements have been added. Add two elements to the linked list by stepping (⇩) through the inner loop twice. Figure 10-13 shows a view of *list* after two elements have been added. Note that the viewer is set to *Presentation – Structure Identifier* view, which is the default. *Basic* and *Monitor* views are also available. The latter view displays any Java owning or waiting threads for the monitor associated with the object. This is used for multi-threading and synchronization. After experimenting with the other views, change the *View* to *Presentation - Structure Identifier* by selecting this on the drop down list as shown in Figure 10-13.


Viewer (by name): list

list

Type jgraspvex.Linked...

Viewer Presentation...

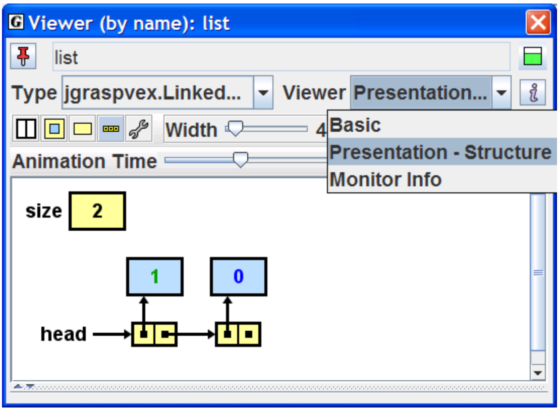
Width 4.0

Scale 1.0

Animation Time 1.0sec.

size 0

Figure 10-12. View of *list* with no elementsadded


Viewer (by name): list

list

Type jgraspvex.Linked...

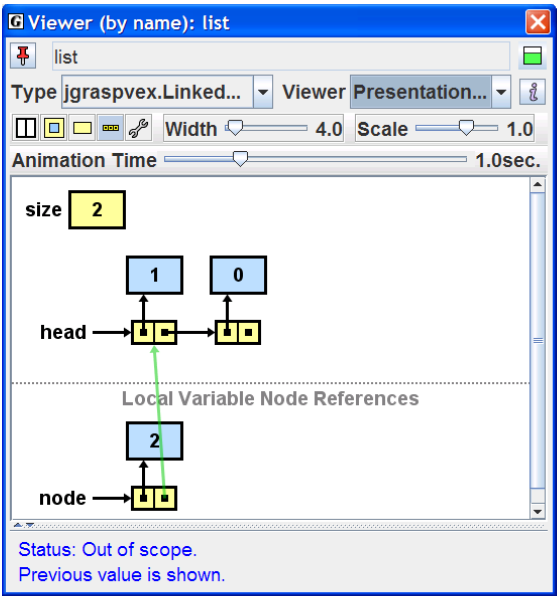
Viewer Presentation...

Width 4

size 2

Figure 10-13. View of *list* with two elements

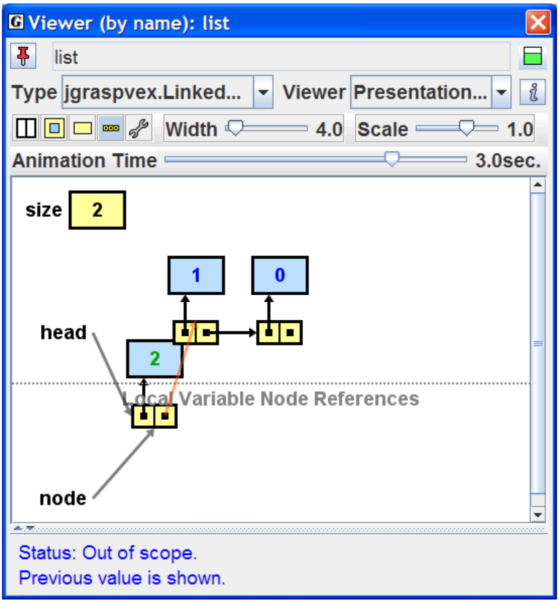
Now you are ready to see the animation of a local node being added to the linked list. You need to step into the *add()* method by clicking the *Step in* button ↳ at the top of the debug tab. Each time you click ↳, the program will either step into the method indicated or step to the next statement if there is no method call in the statement. Figure 10-14 shows *list* after *node.next* for the new node has been set to *head*. Figure 10-15a shows *list* after *head* has been set to *node*, and the new node begins to move into *list*. Figure 10-15b shows *list* after the new node has been inserted. As you repeatedly *step in*, you should see added and inserted nodes "slide" up into *list* and removed nodes slide out of *list*. Note that the Call Stack in the Debug tab indicates the methods into which you have stepped.



Type jgraspvex.Linked...
Viewer Presentation...
Width 4.0
Scale 1.0
Animation Time 1.0sec.

Status: Out of scope.
Previous value is shown.

Figure 10-14. Node about to be added to *list*



Viewer (by name): list

list

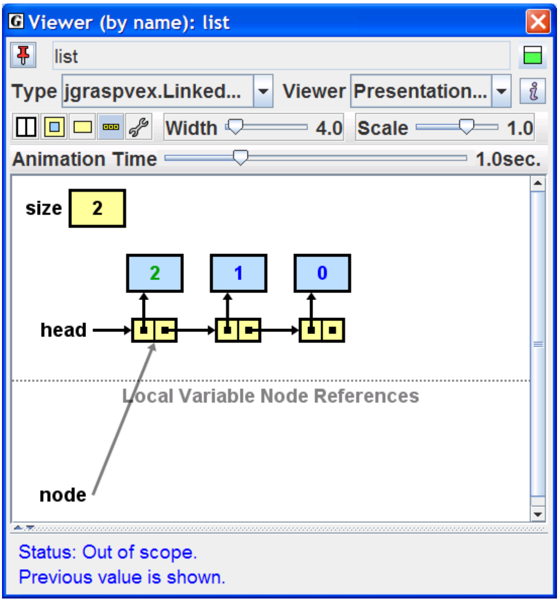
Type jgraspvex.Linked... Viewer Presentation...

Width 4.0 Scale 1.0

Animation Time 3.0sec.

Status: Out of scope.
Previous value is shown.

Figure 10-15a. As node is it is being added to *list*



Viewer (by name): list

list

Type jgraspvex.Linked... Viewer Presentation...

Width 4.0 Scale 1.0

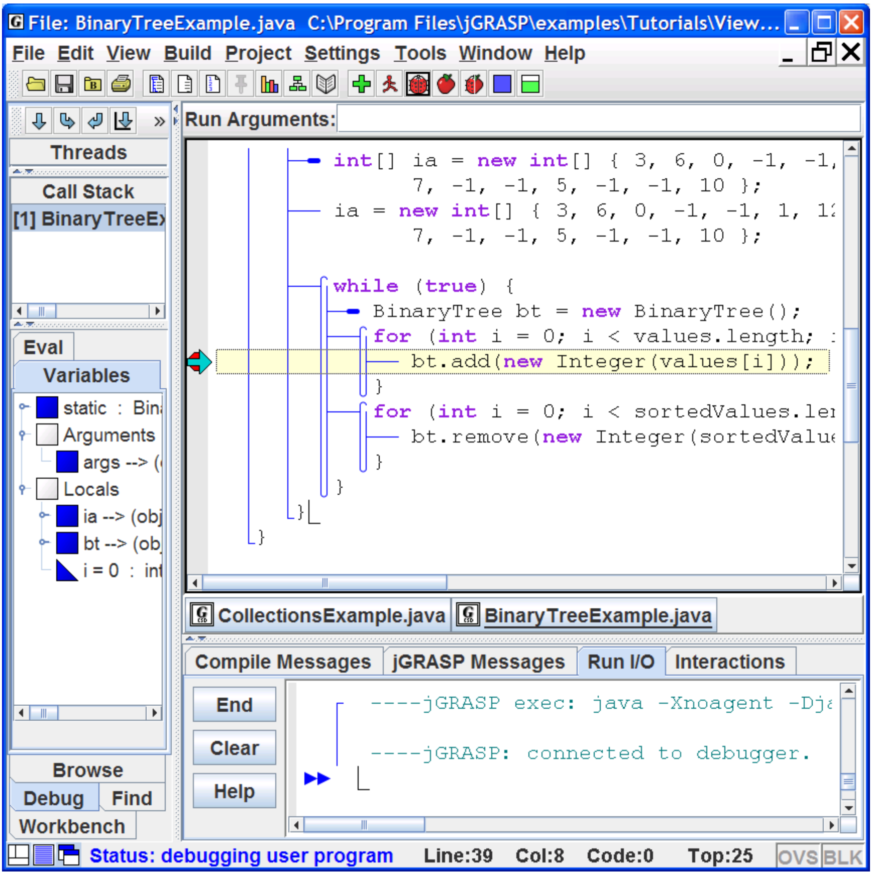
Animation Time 1.0sec.

Status: Out of scope.
Previous value is shown.

Figure 10-15b. After node has been added to *list*

10.6.2 BinaryTreeExample.java

Now let's take a look at an example of another common data structure, the *binary tree*. In the Browse tab, navigate to the *ViewerExamples* directory and open the file BinaryTreeExample.java by double-clicking on it. After compiling it, set a breakpoint ● in the left margin on a line inside the inner loop (e.g., on the line where *bt.add(..)* is called). Now click the Debug button 🐞 on the toolbar. Figure 10-16 shows the program after it has stopped at the breakpoint prior to adding any nodes to *bt*. Now open a viewer on *bt* by selecting and dragging it from the Debug window. The *Structure Identifier* automatically determines that the object is a binary tree and provides an appropriate view for *bt*. Add two elements to *bt* by stepping (⇩) through the inner loop twice.



File: BinaryTreeExample.java C:\Program Files\jGRASP\examples\Tutorials\View...

static : Bin
Arguments
args --> (
Locals
ia --> (obj
bt --> (ob
i = 0 : int

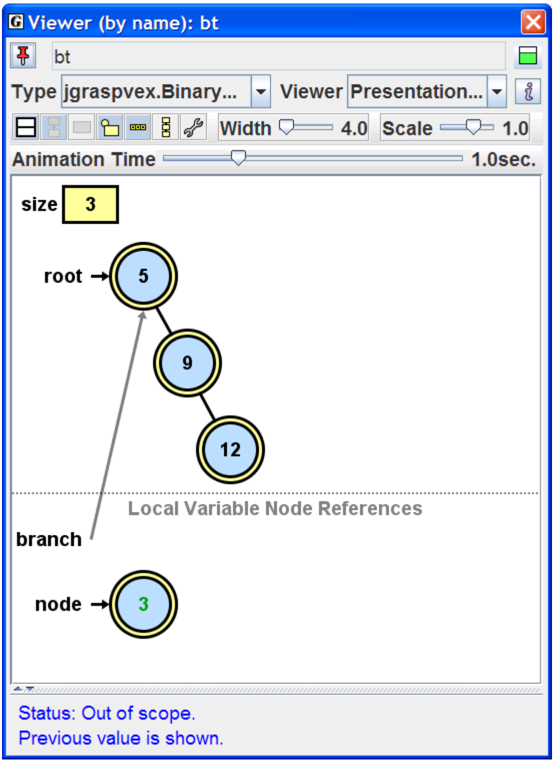
```
int[] ia = new int[] { 3, 6, 0, -1, -1,
      7, -1, -1, 5, -1, -1, 10 };
ia = new int[] { 3, 6, 0, -1, -1, 1, 1:
      7, -1, -1, 5, -1, -1, 10 };

while (true) {
   BinaryTree bt = new BinaryTree();
   for (int i = 0; i < values.length; :
      bt.add(new Integer(values[i]));
   }
   for (int i = 0; i < sortedValues.le1
      bt.remove(new Integer(sortedValue
   }
}
```

```
----jGRASP exec: java -Xnoagent -Dja

----jGRASP: connected to debugger.
```

Status: debugging user program Line:39 Col:8 Code:0 Top:25

Figure 10-16. BinaryTreeExample.java stopped at a breakpoint

Now you are ready to see the animation of a local node being added to the binary tree. You need to step into the *add* method by clicking the *Step in* button ↳ at the top of the debug tab. Each time you click ↳, the program will either step into the method indicated or step to the next statement if there is no method call in the statement. The Call Stack in the Debug tab indicates the methods into which you have stepped. Figure 10-17 shows *bt* after *root* has been passed into the *add()* method as *branch*, and Figure 10-18 shows *bt* after *branch.left* has been set to *node*. As you repeatedly *step in*, you should see added and inserted nodes "slide" up into *bt* and removed nodes slide out of *bt*. Note that since *bt* is a local variable declared in the main method, when you step in to a method as we done in this example, *bt* is no longer in scope. This is indicated by the message at the bottom of the viewer. Because *bt* is a reference variable, the *previous value* still points to the instance of BinaryTree that we are viewing.


Viewer (by name): bt

bt

Type jgraspvex.Binary... Viewer Presentation...

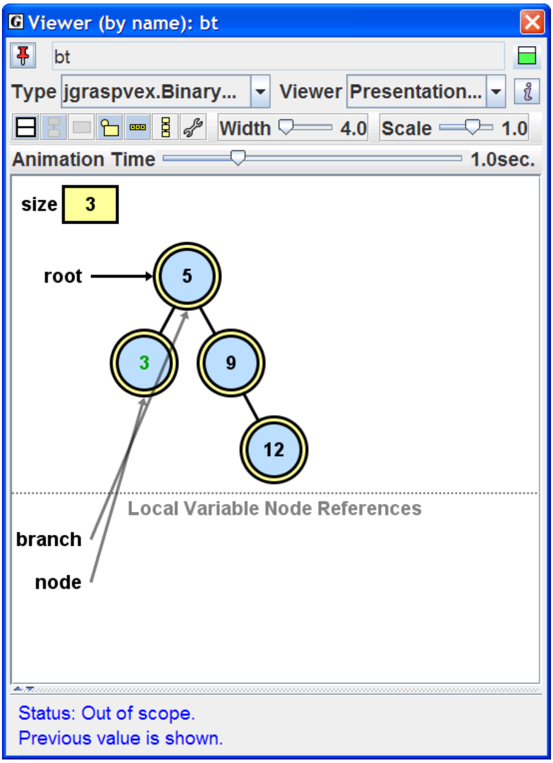
Width 4.0 Scale 1.0

Animation Time 1.0sec.

size 3

Status: Out of scope.
Previous value is shown.

Figure 10-17. Binary tree example as node is about to be added to *bt*


Viewer (by name): bt

bt

Type jgraspvex.Binary... Viewer Presentation...

Width 4.0 Scale 1.0

Animation Time 1.0sec.

Status: Out of scope.
Previous value is shown.

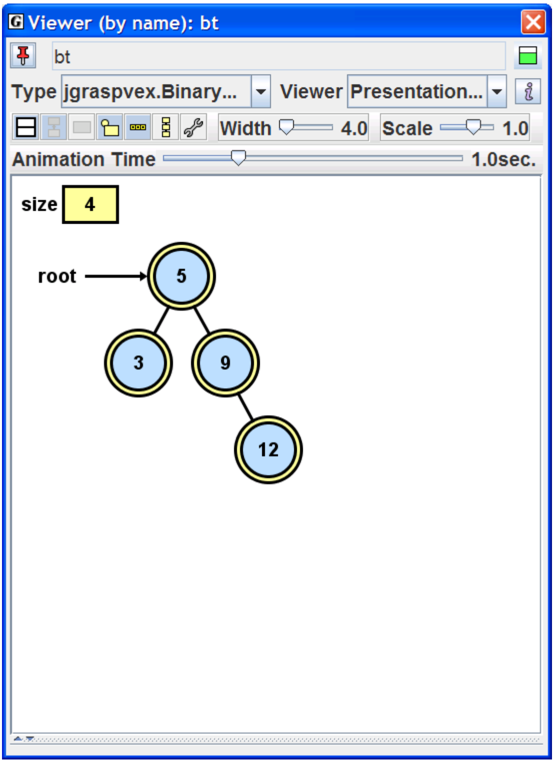
Figure 10-18. Binary tree example after node is added to *bt*

This is a good time to do some experimenting on your own with this example. For example, click the Debug button 🐞 to start the program. Click *step* (⇩) until *bt* is created, then open a viewer on it. Now, as you *step* (⇩) through the code, try to understand exactly what is happening in the program with respect to the diagram in the viewer.

Now repeat the process above, but this time click *step in* (↳) repeatedly. The viewer will show the relationship between the data structure and local nodes in its methods, and the animation should help you understand the code in these methods.

10.6.3 Configuring Views generated by the *Structure Identifier*

The *Structure Identifier* uses a set of heuristics in its attempt to determine if the object for which a view is being opened is a linked list, binary tree, etc. Since the view it provides is only a best guess, some additional configuration may be needed in order to attain an appropriate *Presentation* view. Consider the viewer in Figure 10-19. Figure 10-20 shows the result of (1) clicking the *Configure* button 🔧 (located to the left of the *Width* slider). Figure 10-21 shows the dialog after modifying **Value Expression** by inserting **"Value: " +** (don't forget to enter the plus sign). Figure 10-22 shows the binary tree after clicking **OK** or **Apply** on the Configure dialog. Note that *Width* has been changed to 8.0 to accommodate the new node value.



Viewer (by name): bt

bt

Type jgraspvex.Binary...

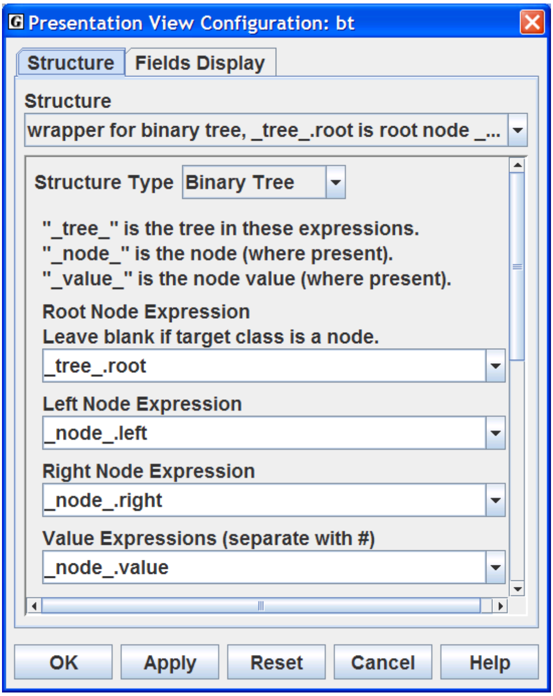
Viewer Presentation...

Width 4.0

Scale 1.0

Animation Time 1.0sec.

Figure 10-19. Binary tree example



Structure
wrapper for binary tree, _tree_.root is root node _...

Structure Type Binary Tree

"_tree_" is the tree in these expressions.
"_node_" is the node (where present).
"_value_" is the node value (where present).

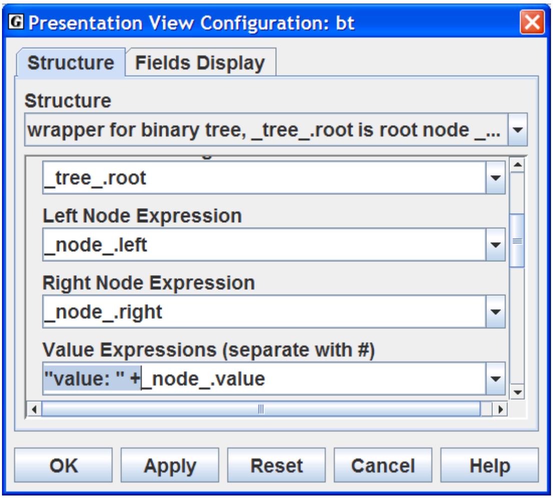
Root Node Expression
Leave blank if target class is a node.
tree.root

Left Node Expression
node.left

Right Node Expression
node.right

Value Expressions (separate with #)
node.value

Figure 10-20. Configuration dialog (🔧)



Structure
wrapper for binary tree, _tree_.root is root node _...

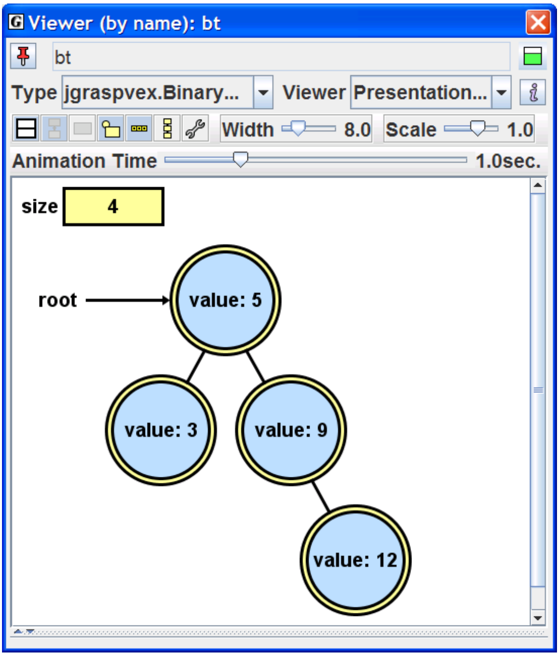
tree.root

Left Node Expression
node.left

Right Node Expression
node.right

Value Expressions (separate with #)
"value: " + _node_.value

Figure 10-21. Configuration dialog with Value Expression modified



Viewer (by name): bt

bt

Type jgraspvex.Binary... Viewer Presentation...

Width 8.0 Scale 1.0

Animation Time 1.0sec.

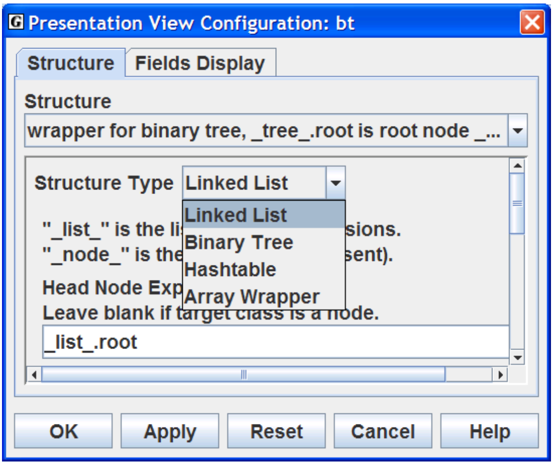
Figure 10-22. Binary tree example after OK (or Apply) on Configuration dialog

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

Continuing with the binary tree example, Figure 10-23 shows the *Structure Type* for *bt* after it has been changed from *Binary Tree* to *Linked List*. Figure 10-24 shows the data structure after the configuration change has been applied (i.e., OK or Apply clicked). Notice that transparent red arrows represent links that are not correct for a linked list.

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

The *Fields Display* tab provides some options with respect to which of the object's fields should be displayed. This is the most common configuration operation to perform on the view provided by the *Structure Identifier*. For some data structures one (or more) of the fields is treated as a formal part of the conceptual diagram itself. For example, the binary tree example has two fields, *size* and *root*, and the viewer treats *root* as part of the diagram, but considers *size* to be optional (however, it is included by default). Only the fields that are not part of the diagram are listed on the *Fields Display* tab.



Presentation View Configuration: bt

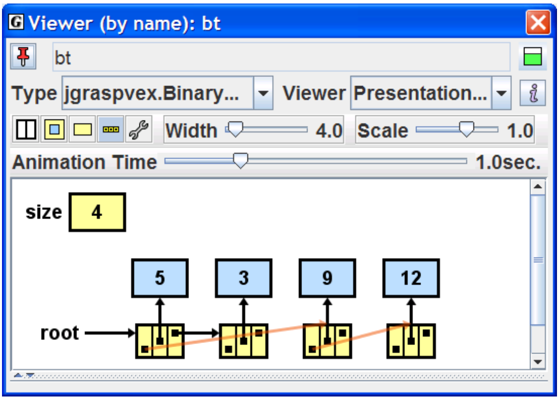
Structure: wrapper for binary tree, _tree_.root is root node _...

Structure Type: Linked List

"_list_" is the li... sions.
"_node_" is the ... sent).
Head Node Exp...
Leave blank if target class is a node.

list.root

Figure 10-23. Changing structure type of *bt* from Binary Tree to Linked List



Viewer (by name): bt

bt

Type: jgraspvex.Binary... Viewer: Presentation...

Width 4.0 Scale 1.0

Animation Time 1.0sec.

Figure 10-24. *bt* shown as a linked list with red translucent links indicating it is not a linked list

10.7 Using the Viewers from the Workbench

Thus far, we have concentrated on opening viewers from the Debug tab while a program is being run in debug mode. In this section, we'll see how to use viewers from the Workbench tab. Objects can be created and placed on the workbench from the CSD window, the UML window, and/or by entering appropriate source code in the Interactions tab. After an object is placed on the workbench tab, a viewer can be opened by selecting the object and dragging it from the Workbench tab.

Let's begin by opening the project for the BinaryTreeExample we used in the previous section. In the Browse tab, navigate to the *ViewerExamples* directory and open the file BinaryTreeExample_Project.gpj by double-clicking on it. After this file is opened, you should see the project listed in the "Open Projects" section of the Browse tab. If the UML diagram is not displayed, double-click on the UML diagram symbol (🖧 <UML>) which should be the first entry under the project in the Browse tab. Figure 10-25 shows the UML diagram with three classes: BinaryTreeExample, BinaryTree, and BinaryTreeNode. Notice that the labels for BinaryTree and and BinaryTreeNode indicate they are contained in package jgraspvex (see the *jgraspvex* folder in the current directory).

If you are still running a program in jGRASP (e.g., in debug mode from the previous section), you should end it before you start the workbench.

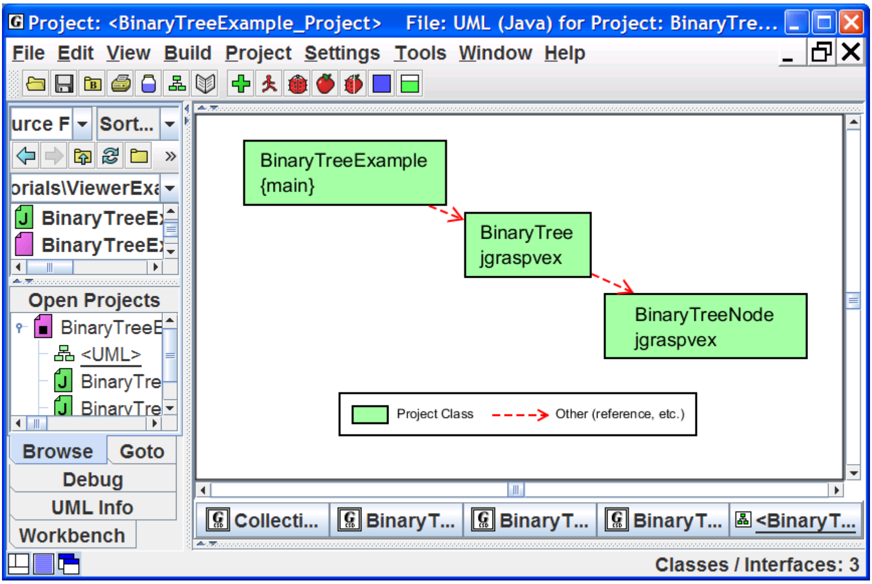


Figure 10-25. UML Class Diagram for BinaryTreeExample_Project

*For more information on creating projects and generating UML class diagrams, see **Getting Started with Objects**, **Projects**, and/or **UML Class Diagrams**.*

Now we are ready to create an instance of BinaryTree. Right-click on the BinaryTree class in the UML diagram as shown in Figure 10-26, then select the second entry on the pop-up list, *Create New Instance*. This brings up the Create New Instance dialog which lists the available constructors for BinaryTree. Figure 10-27 indicates that we are about to create an instance called "jgraspvex_BinaryTree_1" using BinaryTree's only constructor. When the *Create* button is clicked, the new object is placed on the workbench and listed in the Workbench tab as shown in Figure 10-28. Now let's open a viewer, as we've done before, by selecting and dragging the object from the Workbench tab. Figure 10-29 shows the BinaryTree object in the viewer with size 0. To add elements to the instance, we need to invoke its public *add()* method. Clicking on the Invoke Method button 🟩 located in the upper right corner of the viewer brings up the dialog shown in Figure 10-30. To make the dialog stay up so that we can add multiple objects, click on the stick pin 📌 in the upper left corner.

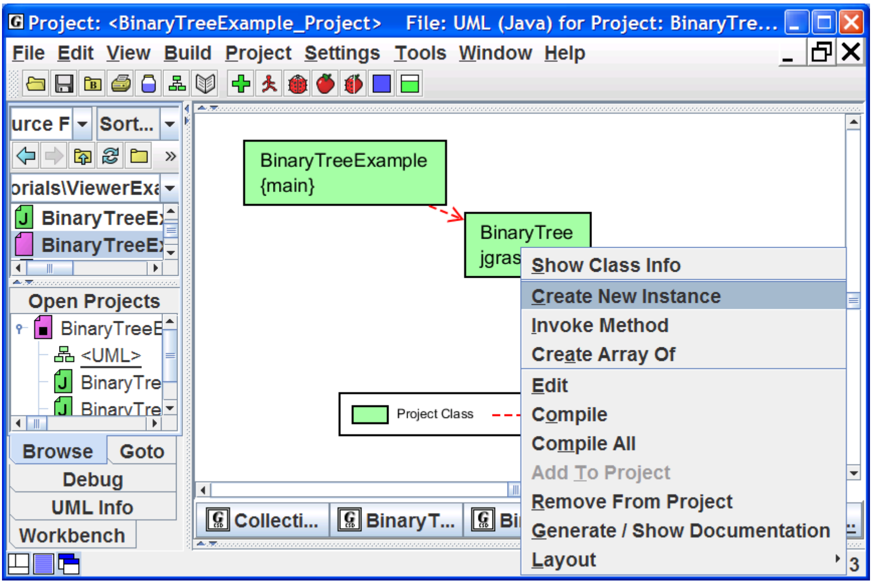
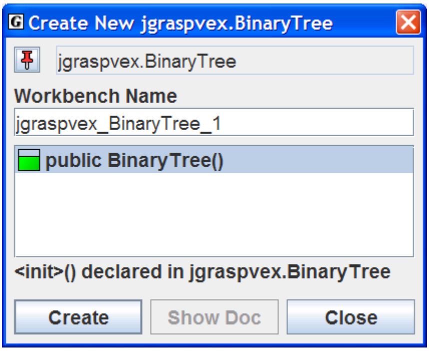


Figure 10-26. BinaryTree class selected to create new instance for workbench



Create New jgraspvex.BinaryTree

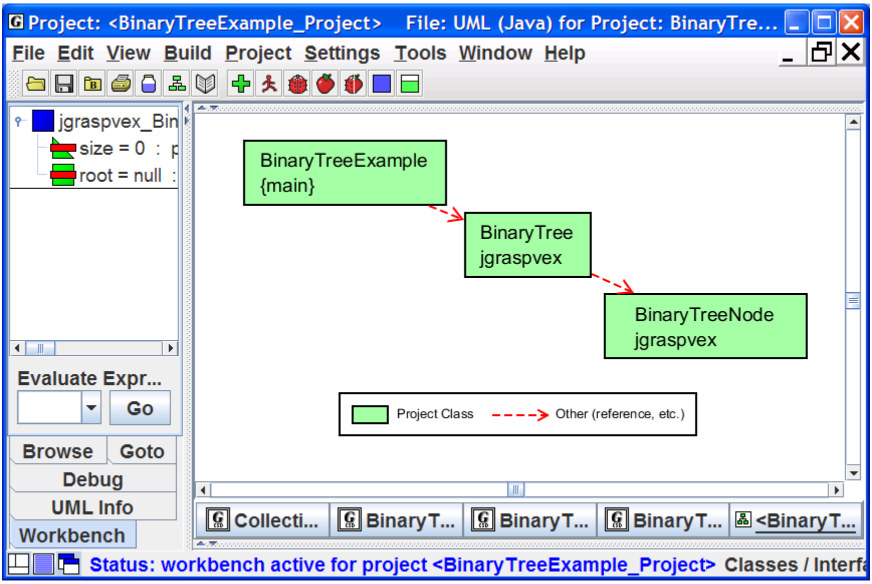
jgraspvex.BinaryTree

Workbench Name: jgraspvex_BinaryTree_1

public BinaryTree()

<init>() declared in jgraspvex.BinaryTree

Figure 10-27. Create New Instance dialog for the BinaryTree class

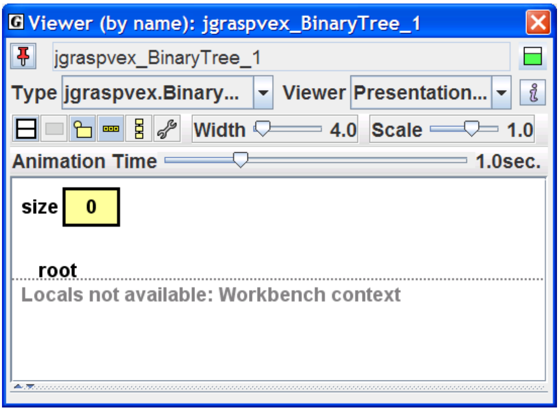


Project: <BinaryTreeExample_Project> File: UML (Java) for Project: BinaryTre...

jgraspvex_Bin
size = 0 : p
root = null :

Status: workbench active for project <BinaryTreeExample_Project> Classes / Interfa

Figure 10-28. BinaryTree object on the workbench (unfolded to show fields)



Viewer (by name): jgraspvex_BinaryTree_1

jgraspvex_BinaryTree_1

Type jgraspvex.Binary... Viewer Presentation...

Width 4.0 Scale 1.0

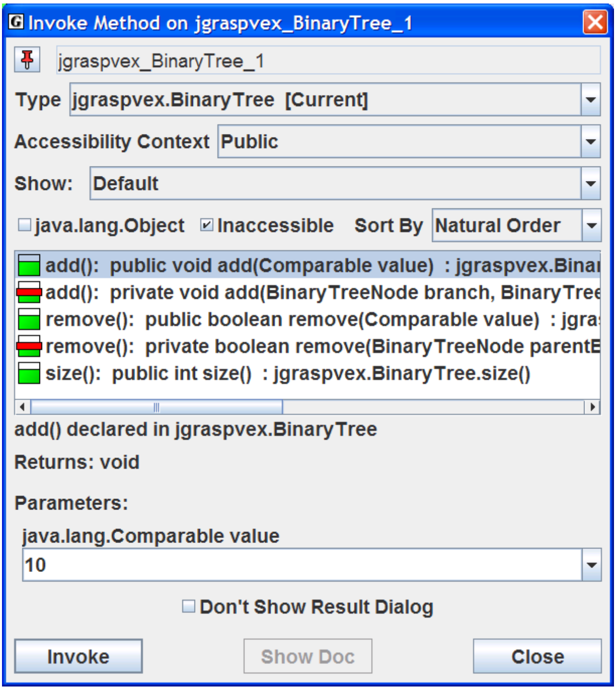
Animation Time 1.0sec.

size 0

root

Locals not available: Workbench context

Figure 10-29. Viewer opened on the object jgraspvex_BinaryTree_1 with 0 elements



Invoke Method on jgraspvex_BinaryTree_1

jgraspvex_BinaryTree_1

Type jgraspvex.BinaryTree [Current]

Accessibility Context Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

🟩 add(): public void add(Comparable value) : jgraspvex.Binai
🟥 add(): private void add(BinaryTreeNode branch, BinaryTree
🟩 remove(): public boolean remove(Comparable value) : jgra
🟥 remove(): private boolean remove(BinaryTreeNode parentE
🟩 size(): public int size() : jgraspvex.BinaryTree.size()

add() declared in jgraspvex.BinaryTree

Returns: void

Parameters:

java.lang.Comparable value

10

□ Don't Show Result Dialog

Figure 10-30. Invoke Method dialog for jgraspvex_BinaryTree_1 to add element 10

Let’s add the value 10 to the binary tree by selecting the public *add()* method. If you are using Java 1.5 or higher, you can enter 10 (without quotes) in the parameter box labeled java.lang.Comparable value as shown in Figure 10-30. Java’s autoboxing feature will convert this to an Integer object. Otherwise enter “10” (with quotes) to make the value a string. Clicking the *Invoke* button will cause the object to be inserted into the binary tree. Notice that the *Result* dialog pops up indicating the invocation was successful.

> *To prevent the Result dialog from popping up after each invocation, you can check the* ***Don’t Show Result Dialog*** *option located above the Invoke button.*

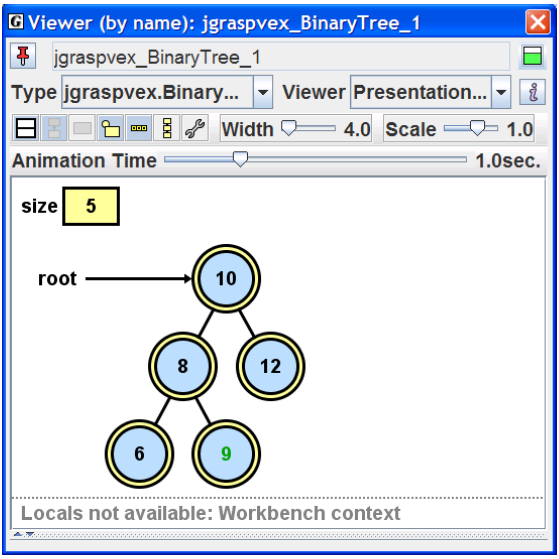
Now let’s add each of the following elements using the same steps we used above to add the element **10** to the tree: **8**, **12**, **6**, **9**

As you add each element, you should see the tree adjust to accept the new node. You can increase or decrease the animation time using the slider provided on the viewer. Decreasing the animation time speeds up the movement of the nodes. After adding these elements, your viewer should look similar to the one in Figure 10-31 with five elements.

Now let’s remove the node containing 8. On the Invoke Method dialog for *bt*, select the public *remove()* method and enter 8 as the parameter then click the *Invoke* button (see Figure 10-32). The node with value 8 is removed and the tree is adjusted accordingly. Now try adding 8 back to the tree and notice where it ends up.

> *In workbench mode, local nodes are not available, as indicated by the message in the viewer. However, if you set a breakpoint in the add() method and then invoke it, the desktop switches to debug mode and allows you to step through the method, at which time local nodes are displayed as appropriate. As soon as you step to the end of the method, the desktop returns to workbench mode. If you do set a breakpoint in a method that you are invoking from the workbench, remember to remove the breakpoint when you are done. Otherwise, each time you invoke the method in the future, you will have to step through it in debug mode.*

In the example above, we created the instance of BinaryTree by right-clicking on a class in the UML diagram. This approach assumes that the classes are in a jGRASP project and that a UML class diagram has been generated for it. Since most users spend much of their time reading and writing code in the CSD window, jGRASP provides a convenient way to create instances of a class for the workbench from the CSD window. The section concludes with an example using this method.



Viewer (by name): jgraspvex_BinaryTree_1

jgraspvex_BinaryTree_1

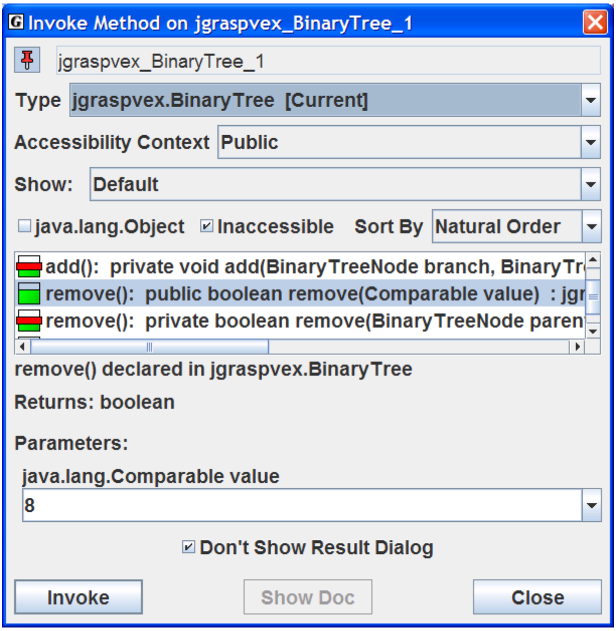
Type jgraspvex.Binary... Viewer Presentation...

Width 4.0 Scale 1.0

Animation Time 1.0sec.

Locals not available: Workbench context

Figure 10-31. Viewer opened on the object jgraspvex_BinaryTree_1 with 5 elements



Invoke Method on jgraspvex_BinaryTree_1

jgraspvex_BinaryTree_1

Type jgraspvex.BinaryTree [Current]

Accessibility Context Public

Show: Default

□ java.lang.Object ☑ Inaccessible Sort By Natural Order

🟥 add(): private void add(BinaryTreeNode branch, BinaryTr

🟩 remove(): public boolean remove(Comparable value) : jgr

🟥 remove(): private boolean remove(BinaryTreeNode paren

remove() declared in jgraspvex.BinaryTree

Returns: boolean

Parameters:

java.lang.Comparable value

8

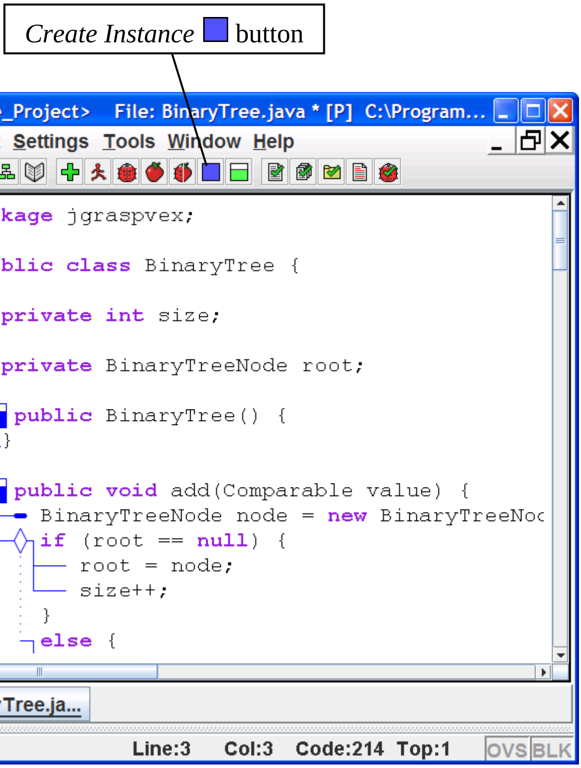
☑ Don't Show Result Dialog

Figure 10-32. Removing element 8

In the Browse tab, navigate to the *ViewerExamples* directory if you are not already there. In this directory, you should see the directory *graspvex* which contains the data structure classes for this tutorial. Find BinaryTree.java and double-click on it to open it in a CSD window. Figure 10-33 indicates the location of the *Create Instance* 🟦 button on the CSD window tool bar. Clicking this opens the Create New Instance dialog which lists the available constructors for BinaryTree as shown above in Figure 10-27.

You can find also create an instance from the menu by clicking **Build** > **Java Workbench** > **Create New Instance**. This is illustrated in Figure 10-34.

Regardless of the way you choose to create instances, the workbench provides a convenient way to test a class and its methods without the necessity of a driver program. When a viewer is opened for an instance of a data structure on the workbench, the opportunity for understanding the software is even greater.

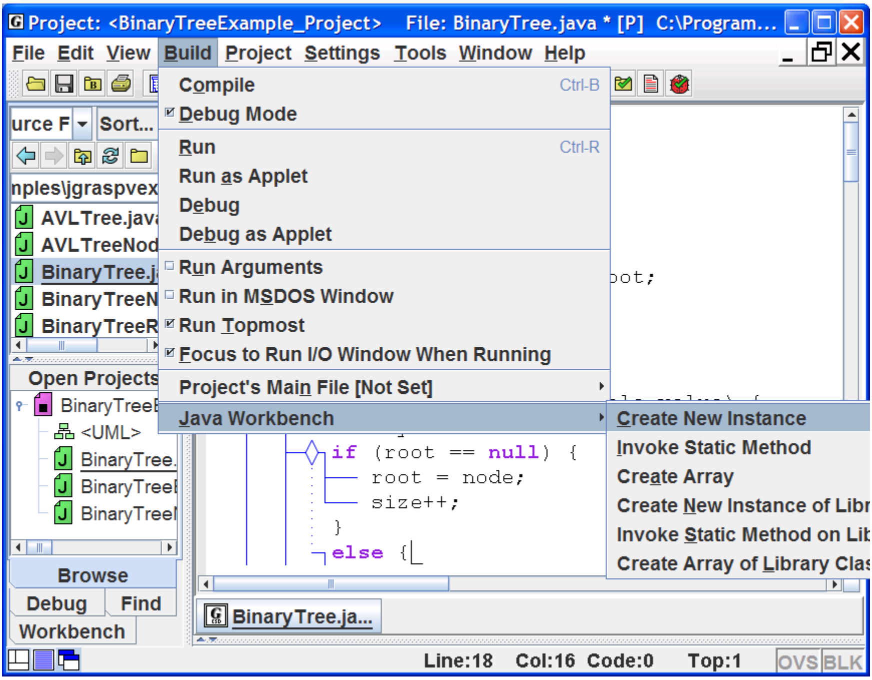


Create Instance 🟦 button

```
package jgraspvex;

public class BinaryTree {

   private int size;

   private BinaryTreeNode root;

   public BinaryTree() {
   }

   public void add(Comparable value) {
      BinaryTreeNode node = new BinaryTreeNoc
      if (root == null) {
         root = node;
         size++;
      }
      else {
```

Line:3   Col:3   Code:214   Top:1

Figure 10-33. CSD window with BinaryTree.java



```
oot;

if (root == null) {
   root = node;
   size++;
}
else {
```

Figure 10-34. Using the Build menu to an create instance

10.8 Summary of Views

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

General Description of Views

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

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

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

array, *String*, *ArrayList*, *Vector*, *Stack*, *LinkedList*

Presentation - Structure Identifier – A conceptual view is provided when a structure is automatically detected; typically handles a moderate number of elements efficiently. This view is listed on the *View* drop down list for many objects and if selected, the user has the opportunity to configure the viewer for a linked list or binary tree even if neither was automatically identified. The following structures are currently supported in jGRASP 1.8.7:

> linked lists, binary trees (including binary heap, red black trees, AVL trees), hashtables, and array wrappers (lists, stacks, queues, etc.)

10.9 Exercises

(1) Open *CollectionsExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops at the breakpoint, open a viewer on instances of one or more of the following, then step through the program:

a. array

b. ArrayList

c. LinkedList

d. TreeMap

e. HashMap

(2) Continuing with the program from above, let's use the Auto-Step feature of the jGRASP Debugger. With the program stopped at a breakpoint and one or more viewers open, select *Auto Step* ⇊ on the debug control panel and click the *Step* ⇩. You can control the speed of the steps with the slider bar beneath the step controls.

(3) Open *QueueExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops the breakpoint, open a viewer on queue. Select *Auto Step* ⇊ on the debug control panel. Now click the *Step* button ⇩. You can control the speed of the steps with the slider bar beneath the step controls on the debug control panel. You can control the speed of the animation with the slider bar on the viewer. By watching the queue in the viewer as the program executes, what can you learn about the implementation of the queue?

(4) Open *LinkedListExample.java*, set an appropriate breakpoint, and run it in debug mode. After the program stops at the breakpoint, open a viewer on list. Select *Auto Step* ⇊ on the debug control panel. Now click the *Step in* button ↳.

(5) Open BinaryTreeExample.*java* and repeat the task described in (4).

(6) Although *float* and *double* are primitive data types rather than data structures, the IEEE standard representation for floating point types is quite interesting.

Create floating point variable in your program by adding the statement:

```
double myDouble = 4096.0;
```

After compiling the program, set an appropriate breakpoint, and run the program in Debug mode. Open a viewer on `myDouble` and set the view to

Detail. The *Detail* view for *float* and *double* values shows the exponent and mantissa representation used for floating point numbers and how these are calculated.

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

Notes