DrJava User Documentation

DrJava User Documentation

Table of Contents

Chapter 1. Introduction

DrJava is a programming environment for Java, primarily intended to help students focus more on program design than on the features of a complicated development environment. DrJava also provides many advanced features for experienced developers. These features center around DrJava's Interactions Pane, which is a "read-eval-print loop" that allows users to easily develop, test, and debug Java programs in an interactive, incremental manner.

Home Page: http://drjava.org
Original Paper: http://drjava.org/papers/drjava-paper.shtml

Chapter 2. Getting Started

This chapter describes the basics for how to start using DrJava, including where to get the program and how to run it.

Philosophy

The general idea behind DrJava is to provide powerful development tools that are as easy to use as possible. For this reason, we try to make DrJava easy to run and easy to understand, through a simple user interface with few panes and a legible toolbar. Meanwhile, we want to help novice users become comfortable with writing Java code by allowing them to quickly evaluate expressions in DrJava's Interactions Pane. All of our powerful features try to build on this simple and interactive interface.

The rest of this chapter will walk you through downloading and running DrJava, but if you have the DrJava `.jar` file, you can just double-click it to get started.

Downloading DrJava

You can download the newest releases of DrJava as a `.jar` file from our home page, http://drjava.org, or directly from our Project Filelist [http://sourceforge.net/project/showfiles.php?group_id=44253] page on SourceForge.

Stable, Beta and Development Releases. We make a distinction between Stable, Beta and Development releases of DrJava. All releases must pass our rigorous suite of unit tests and should be safe to use, but we have found that a period of beta-testing can be helpful for finding additional bugs. Any large new features are first released as a Beta release and go through a beta-testing period before being included in Stable releases, ensuring these releases are safe for all users. Our Development releases contain newer features that are under development. We believe these releases are also ready to use, but they have not been widely beta-tested, so some users may prefer to use Beta or Stable releases, or perhaps only Stable releases.

Running DrJava

On many platforms, DrJava can be started by simply double-clicking on the `.jar` file you downloaded. DrJava can also be started from a command prompt, where you can optionally give it a list of source files to open at startup:
```
java   -jar   drjava-RELEASE-DATE-rREVISION.jar  [-config  [CONFIG_FILE]]
[filename.java...]
```

Replace RELEASE-DATE-rREVISION with the appropriate value for your version of DrJava, e.g. **java -jar drjava-stable-20080904-r4668.jar**. The "config" argument is optional and allows you to specify a custom configuration file, rather than the `.drjava` file stored in your home directory.

Running DrJava on Mac OS X. If you are using Mac OS X, you can download DrJava as an Application from our website. Download the `drjava-RELEASE-DATE-rREVISION-osx.tar.gz` file and decompress it. You can then copy the DrJava icon into your Applications folder or keep it on your Dock.

Running DrJava on Windows. If you are using Windows, you can download DrJava as an executable file from our website. Download the `drjava-RELEASE-DATE-rREVISION.exe` file. You can then run it like a normal Windows program.

System Requirements

DrJava requires Java version 5 or later. Note that you will need to have the JDK (not the JRE) installed if you wish to use the debugger or create Javadoc within DrJava.

We recommend downloading and using Oracle's JDK 6 (from http://www.oracle.com/technetwork/java/javase/downloads/index.html) for Solaris, Linux, and Windows. Other users should use the Java virtual machine that comes with their operating system (including Mac OS X).

Mac OS X 10.7 "Lion" does not include a default Java runtime anymore. Therefore, the operating system may prompt you to install Java if you have not installed it before. You can also download Java for OS X Lion [http://support.apple.com/kb/DL1421] from Apple's website, or search for Java downloads for Mac OS X [http://support.apple.com/kb/index?page=search&fac=Downloads&q=java%20for%20os%20x&src=support_site.kbase.search.advanced].

DrJava uses two Java Virtual Machines (one for the main program, and one for the Interactions Pane) that use Java's Remote Method Invocation (RMI) to communicate with each other. RMI uses TCP/IP as the default transport mechanism, so you must have those drivers installed. Without TCP/IP, DrJava will not start properly.

Note that there is an incompatibility between Java's Swing GUI library and the Compiz window manager on Linux. We, the developers of DrJava, cannot do anything to fix this problem. We hope that future versions of Java and Compiz will address the incompatibility. In the meantime, we recommend that you disable Compiz if you experience problems. We also suggest that you use the latest versions of Compiz and Java, so you can benefit from possible bug fixes made by Oracle and the Compiz developers. For more information, see http://drjava.org/compiz/.

License

Copyright (c) 2001-2009, JavaPLT group at Rice University (drjava@rice.edu) All rights reserved.

Redistribution and use in source and binary forms, with or withoutmodification, are permitted provided that the following conditions are met:

• Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

• Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

• Neither the names of DrJava, the JavaPLT group, Rice University, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR

Chapter 3. Editing Programs

DrJava's core component is an editor for writing Java source code. Like most text editors, it supports a wide range of editing features such as "Find/Replace", "Go to Line", "Go to File", while also providing more advanced features like syntax coloring, automatic indentation, brace matching, and even a limited notion of auto-completion.

Definitions Pane

The Definitions Pane is the main window of DrJava, displaying the currently active source file. As you edit files in this window, DrJava helps out with several useful features.

Syntax Coloring. DrJava colors special types of text differently to help make the structure of the program more apparent. Comments appear in green, while Java keywords and types appear in blue. Strings are colored red and characters are colored magenta, with all other text colored black. These colors are all configurable (see Configuring DrJava).

One notable difference between syntax coloring in DrJava and other common editors (such as Emacs) is that DrJava uses *fully correct* coloring as the document is edited. For example, simply typing the beginning of a block comment ("/*") will immediately update the coloring of the entire document, unlike some other editors which will only update the color of a line when that line is edited. Having an accurate view of the program is an important aspect of understanding its structure.

Automatic Indentation. The key to indenting code in DrJava is the Tab key. Rather than simply inserting a tab or spaces, pressing Tab properly indents the current line (or selected text) using common coding conventions. As you type multiple lines of code into the Definitions Pane, DrJava automatically indents each line using the same technique. By default, two spaces are used for each indentation level, although this can be configured in the Preferences window. (In DrJava, code is always indented with spaces, and never with actual tab characters.)

Brace Matching. To help you match open and close braces, DrJava highlights the region enclosed by a pair of braces. If you place the cursor immediately after a close brace, parenthesis, or bracket, all text between that character and the corresponding open brace is highlighted in another color. Like syntax coloring, brace matching is also done in a *fully correct* manner, updated with each keystroke. In addition, when the cursor is right after a closing curly brace, the line containing the matching open brace is displayed in the lower status bar.

Commenting / Uncommenting. To help you easily write multi-line comments, DrJava automatically adds spaces and an asterisk on each new line. In addition, there is an option in the "Miscellaneous" section of the Preferences window that will tell DrJava to automatically close multi-line comments for you. Commands in the Edit menu are also available to comment out or uncomment a block of selected code using winged comments ("//"). The key bindings for these commands default to Ctrl+Slash and Ctrl+Shift+Slash respectively. Commenting out a block of code will place "//" markers at the start of each line in the block, preserving the indentation of the code.

Context Menu. The Definitions Pane has a context menu, which can be used by right-clicking in the pane. (Mac users should use Ctrl+Click or Option+Click.) This menu provides shortcuts to useful features such as cut, copy, and paste, as well as indenting, commenting, and setting breakpoints and bookmarks.

Auto-Completion. DrJava supports a very limited notion of auto-completion that is nonetheless useful. This feature is accessible as "Auto-Complete Word Under Cursor" in the Edit menu, and it is also bound to the keyboard shortcut Ctrl-Shift-Space by default. When invoked, DrJava will look at the word to the left of the cursor and attempt to auto-complete it, based on a list of documents currently open. If there

is no unique auto-completion match, DrJava displays a predictive input dialog with the auto-completion candidates.

When a project is open, DrJava can also be configured to scan all class files after a compile to obtain the auto-completion information (see Configuring DrJava). With that option set, DrJava can auto-complete the names of all classes in your project, even those of inner classes.

On the "Auto-Complete Word Under Cursor" diaog, there is a checkbox labeled "Java API". If this is checked, then DrJava will also use the class names from the Standard Java API, JUnit 3.8.2 and the user-specified additional libraries as suggestions for auto-completion. If it is disabled, only class names from your own source files are used.

Next to the "OK" button is the "Fully Qualified" button. If the class "Integer" is selected, and the user presses "OK", DrJava will auto-complete the word to "Integer". If, however, "Fully Qualified" is used to close the dialog, DrJava will enter the entire fully qualified class name, "java.lang.Integer" in this case.

Please note that auto-completion currently only works for class names, and completely ignores all context except for the word to the left of the cursor (i.e. it may generate code that does not compile).

Clipboard History. Any text you copy or cut out of DrJava documents will be placed in the clipboard history, and the last 10 entries are kept (that number is configurable, see Configuring DrJava). To access one of the entries in the history, use the "Paste from History" command in the Edit menu or press Ctrl +Shift+V. In the dialog that opens up, you can browse the history and select the entry to paste. In addition to inserting the text at the cursor, the selected entry will also be moved to the top of the clipboard history, and will therefore subsequently be available with the regular paste command.

The clipboard history is a great tool to minimize scrolling and document switching: Instead of going back and forth several times, you can just "copy, copy, copy" several pieces of code in a row, then go to another place in the code and do "paste, paste from history, paste from history".

Right Margin. DrJava can display a line after a specified number of columns, representing the right margin of the document. By default, this line is displayed after 120 columns. You can type past this line, and it has no effect on the saved files or executed programs, but the right margin line can help you format your source code more uniformly.

You can enable or disable the right margin line in the Display category of the Preferences window. The color of the line can be changed in the Colors category of the Preferences. (See Configuring DrJava.)

Multiple Documents

Most Java programs have several closely related source files, so DrJava supports having multiple documents open at the same time. A list of all of the names of the open documents appears in a pane to the left of the Definitions Pane, listing files in the order in which they were opened. You can view and edit a particular document by selecting it in the list, or by using the Previous and Next Document commands in the Edit Menu. These commands will switch to the next for previous document in alphabetical order. (These commands have keyboard shortcuts as well: Ctrl+Comma and Ctrl+Period.) You can also press Ctrl +Back Quote to switch between recently active documents. This short cut is similar to Window's shortcut for switching windows. Hold down Control and press Back Quote to activate it. A small window will show the filename of the document about to be switched to. Press Back Quote to cycle filenames and release Control to switch docments.

The full file path of the current document is displayed both in the title bar and in the status bar at the bottom of the window.

Context Menu. The Document List also has a context menu, which can be used by right-clicking on any document in the list. (Mac users should use Ctrl+Click or Option+Click.) This menu provides shortcuts

to document-related commands, such as saving, reverting, printing, compiling, testing, running Javadoc, and calling the main method.

Source Navigation

DrJava has many simple features to help you edit and navigate source files.

Find/Replace. DrJava has a Find and Replace utility, which is conveniently displayed as one of the tabs at the bottom of the window, rather than as a dialog blocking part of the window. The tab is first displayed when you Find/Replace from the Edit Menu (or using the keyboard shortcut, Ctrl+F), and it can be closed by clicking on the "X" button in the upper right corner of the tab (or by hitting the Escape key).

To search for a term, type it in the Find text field and click "Find Next" or "Find Previous" (or press Enter). To replace the term with another, type the new term in the Replace text field, find an occurrence using "Find Next" or "Find Previous", and then click "Replace". The "Replace/Find Next", "Replace/FindPrevious" and "Replace All" buttons help to speed up this process.

The "Find All" button accumulates all occurrences and displays them in a new, separate pane labeled "Find: word", where "word" is replaced with the search string. You can keep as many "Find All" panes open as you like. The panes keep the occurrences sorted by document and line number and allow you to jump to the source location with the "Go to" button, bookmark an occurrence using the "Bookmark" button, or to remove an occurrence from the list with the "Remove" button. Occurrences can also be underlined with different colors to make them easier to find in the Definitions pane.

There are also four checkboxes to customize each search: "Match Case","Search All Documents", "Whole Word", "No Comments/Strings", and "No Test Cases". Unchecking the first box will return case-insensitive results, checking the second box will tell DrJava to search through all of the open documents in sequence, checking the third box will ignore partial matches (i.e. it will ignore "hello" if the search text is "lo"), and checking the last box ignores instances found within comments and strings. Checking the "No Test Case" box will cause Java to ignore matches in files with test cases. Currently, this is being determined by examining the file name: If the file name ends with "Test.java", it is considered a test case and will be skilled.

Note that if "Search All Documents" is enabled, the search will not wrap to the beginning of the current document until all other documents have first been searched. DrJava can also search across more than one line of text (i.e. the search string can include line breaks). For detailed instructions on its usage, see the "Find and Replace" section in the Quickstart documents under the Help menu.

The last checkbox is "Search Selection Only". Checking this checkbox, allows the user only to Find/ Replace All and disables the Search All Documents as well as No Test Cases checkboxes. The search will be limited to a highlighted portion of the document. The Find Again option after Find All with Search Selection Only has been checked only searches within the selected region likewise.

Go to Line. Selecting "Go to Line" from the Edit Menu (or hitting Ctrl+G) will display a dialog allowing you to scroll to a particular line number.

Go to File. With the "Go to File" dialog from the Edit Menu (or hitting Ctrl+Shift+G), you can quickly jump to another file. It will open a predictive input dialog, ask you to type the name of the document, and quickly narrow down the list of candidates. You can then use the cursor keys and Enter to select the file you want to view.

The "Go to File" dialog also incorporates the function of the "Go to Line" dialog: If you enter a colon (":") followed by a line number at the end of your input text, the editor will select the file and then jump to the entered line number. Example: "FooBa:123" may take you to the FooBar.java file at line 123.

Go to File Under Cursor. "Go to File Under Cursor", also in the Edit Menu and bound to the shortcut F6), is a special form of "Go to File": It considers the word the cursor is currently on and uses it as a starting point for your search. If there is a unique match, DrJava will open that file immediately; otherwise, this feature behaves just like "Go to File".

Fast Switching. With Fast Switching, you can easily switch between recently active documents. Simple hold down Control, and press the Back Quote key to navigate through the filenames of recently active documents. Release the Control key to switch to the document with that filename.

Line Numbering. DrJava displays the cursor's current line number and column number on the right side of the status bar at the bottom of the window. The line number is shown on the left and starts at 1, and the column number is shown on the right and starts at 0.

All line numbers can also be displayed in the left margin of the Definitions Pane, using the "Show All Line Numbers" option in the "Display Options" section of the Preferences window. The line number font can be changed in the "Fonts" section. (See Configuring DrJava.)

Bookmarks. DrJava allows you to bookmark places in your code that you deem important. If you have a project open, the bookmarks even get saved to and loaded from the project file. By pressing Ctrl-Shift-M or using the "Bookmarks" item in the Tools menu, you can display the list of bookmarks in the "Bookmarks" pane. The bookmarks are sorted by document and line number.

With the Ctrl-M keyboard shortcut or the "Toggle Bookmark" items from the Tools menu or the Definition pane's context menu, you can add and remove bookmarks. If no text is selected, "Toggle Bookmark" will add or remove a bookmark for the entire line the cursor is on. If text is selected, the selected text is bookmarked.

By selecting a bookmark in the "Bookmarks" pane and pressing the "Go to" button, you can jump to the location associated with the bookmark. You can also select one or more bookmarks and remove them with the "Remove" button, or clear the entire list with the "Remove All" button.

Predictive Input Dialogs

DrJava uses "predictive input dialogs" in several places, e.g. in the "Go to File" and "Open Java API Javadoc" features. This type of dialog presents you a list of candidates and allows you to quickly select one of them based on its name.

The top portion of the dialog displays a list of candidates that match your current choice. The text field below allows you to enter text to narrow down the list. On the bottom, there is an "OK" button to accept the current selection, a "Cancel" button to leave the dialog, and a drop-down box to choose the way candidates are selected. Matching is always done case-insensitively.

Fragments. With this matching strategy, you can enter word fragments separated by spaces, and the list will display all the items that contain all the fragments. Example: If you enter "foo bar", the items "FooBar" and "SomeFooMooBar" will be displayed, but "Foo" or "FumBar" will not.

Prefix. With this strategy, only items that begin with the text entered will be displayed in the list. If you enter "foo", the items "Foo" and "FooBar" will be displayed, but "BarFoo" will not.

RegEx. This matching strategy allows you to enter Perl-style regular expressions (as implemented by the java.util.regex package), and the list will contain all the items that match the regular expression. As an example, the regular expression ".*" will display all items, while "[a-m].*" will display all that begin with the letters 'A' through 'M'.

Detachable Tabbed Panes

For a long time, the Interactions Pane, the Console Pane, Find/Replace, and the Compiler and JUnit Panes were always attached to the bottom of the DrJava main frame. Users who desire the Definitions Pane to be as large as possible, or users with multi-monitor displays, may wish to use the new "Detach Tabbed Panes" option in the "Edit" menu.

When this option is enabled, the tabbed panes will be detached from the bottom of the main frame and displayed freely floating in their own window. The window position is saved, so it's possible to create a nice layout, quit DrJava, and have the same layout restored when DrJava is started again. Another nice side-effect is that all panes can display a lot more items without the need for scrolling. To re-attach the tabbed panes to the DrJava main frame again, simply disable the "Detach Tabbed Panes" option again.

The panes in the free-floating window otherwise behave exactly the same as when they are attached. It's just a different screen layout.

Chapter 4. Project Facility

Overview

DrJava includes a project facility for managing many files. The project facility allows you to save your open files in a project file, and reopen the project file at a later time to work on some or all of the project files.

New Projects. To create a new project, either select "New" in the Project menu for a project that is initially empty, or select "Save As" in the project menu when you have one or more files already open.

Selecting "Save As" in the Project Menu will create a new project out of the files currently open.

Saving a Project. To save a project, either select "Save" in the Project menu, or select "Save As" in the Project Menu. Note that when you save the project, it only saves the list of files that are in the project, not the files themselves. Saving the project does not save the individual files that are members of the project. Use "Save All" if you wish to save all files as well as the project file to which they belong.

Saving a project will also save which document is currently active, as well as the cursor location in every open document. It will also remember the layout of the project tree, so if some folders are collapsed when the project is saved, then the folders will be collapsed the next time the project is opened.

Opening a Project. To open a project, select "Open" under the Project menu, then select a previously saved project file. You can also open previously open projects in the recent project file list in the project menu. Simply open the Project Menu and click the name of the project file to open that project.

Compiling a Project. There are two options for compiling a project: compiling the open project files, or compiling the entire project. To compile all open project files, select "Compile Open Project Files" under the Project menu, or right click the root of the tree and select "Compile Open Project Files." This will compile all files in the project view including auxiliary files. All files in the external branch (Under the External Files folder) will not be compiled.

Similarly, to compile all project files, even if they are not currently open in DrJava, select "Compile Project" from the Project Menu or the Context menu for the root of the tree. This will compile every source file in the project directory as well as source files in the Auxiliary Files branch.

When not in project view, the "Compile All" button compiles all open files, whereas in project view, "Compile All" only compiles the open project.

Testing a Project. There are two options for testing a project: testing the open project files, or testing the entire project. To test all open project files, select "Test Open Project Files" under the Project menu. This will test all JUnit test files currently open in the Source Files project branch as well as the Auxiliary Files. All files in the external branch (Under the External Files folder) will not be tested.

To test all project files, including files not open in DrJava, choose "Test Project" in the project menu. This will search the project directory (the directory that the project file is saved in) for source files, and test any and all junit test cases it finds. This will also test all test files in the Auxiliary Files branch of the project tree.

Running a Project. To run the main method of a project, select "Run Main Class" under the Project menu. This option is only available if you have specified a file containing the project's main method in the "Project Properties" dialog in the Project menu.

Create Jar File from Project. You can create a jar file that contains the project's source code, its compiled class files, or both by selecting "Create Jar File from Project" in the Project menu. This will

display a dialog that allows you to specify the jar file's location and what gets put into it. If you are placing class files into the jar file, you can make the jar file executable by selecting "Make executable" and entering the main class. For more control over the properties of the jar, you may enter a custom manifest by selecting "Custom Manifest" and pressing the "Edit Manifest" button. You may opt to include all source files in the jar, embeded in a seperate jar, by selecting "Jar source files". You can also include all files in the project directory by selecting "Jar All files".

Note that if you have not specified a build directory in the Project Properties all classes found in the same directory contaning the project file will included if you place class files in the jar. For class files to be included successfully you must have recently compiled the project.

Tree View

Overview. When using the project facility, the navigator pane on the left hand side of the DrJava window displays the files in a tree view, giving you a graphical representation of where the project files are located in the project directories. Files are organized into three main branches: Source Files, Auxiliary Files, and External Files. The exact characteristics of each of these branches will be described in the following paragraphs.

Some of the menu items behaviors change slightly when a project is open. The "Compile All" button will compile only project Source Files instead of every open file. Likewise, "Test All" will only test the files that are in the "Source Files" and "Auxiliary Files" branches. You can manually compile or test the other branches by right clicking on the folder and selecting "Compile Folder" or "Test Folder" respectively.

Only one project can be opened at a time.

Source Files. A file is categorized as a Source File if it is located at or below the directory in which the project file is saved. We call the directory that the project file is saved the "project directory." This means that the location of the project file in your filesystem will determine which Java files will be considered part of your project.

External Files. Files located outside of the project directory will automatically be added to the External Files branch. External Files are not compiled or tested when you compile or test the project. Also, the list of external files that are currently open is not saved in your project file.

Included External Files. Included External Files are files that are located outside of the project's root directory and are explicitly added to the project. Included External Files are compiled and tested when the project is compiled or tested. Also, the list of Included External Files is saved to the project file when the project is saved. Included External Files will also be automatically opened when the project is opened. Only a file in External Files can be moved to the Included External Files branch.

To add an External File to the Included External Files branch, right click the filename in the External Files list and select the "Include With Project" option. To remove a file from the Included External Files, right click the filename and select the "Do Not Include With Project" option.

Project Properties

Project Root. The project root specifies the directory that corresponds to the default package of the project. All project files should be located in this directory or one of its subdirectories.

Build Directory. The project facility allows the user to set a build directory where class files will be compiled. This gives the user the ability to separate source and class files. This setting is required for the "Clean Build Directory" and "Create Jar from Project" features to work correctly.

To clean the build directory, open the Project Menu and click "Clean Build Directory." This will remove all .class files and empty directories in the build directory.

Working Directory. The working directory corresponds to "current" directory of the project, i.e. **new File(".")** or **System.getProperty("user.dir")**.

Main Class. The project facility allows the user to specify a "Main Class" for your project. When the "Run Project" button is pressed or the "Run Main Class of Project" is then selected in the Project menu, the main method of the "Main Class" specified will be executed. If no "Main Class" is specified, the "Run Main Document of Project" item will not be available, and the "Run Project" button is replaced by the "Run" button that runs the main method of the current document. The "Main Class" may be specified as a file, using the file selection dialog, or as a fully qualified class name entered directly into the field. Note that to run inner classes, the name must be entered directly.

Extra Classpath. In the "Extra Classpath" area, you may add additional directories or jar files to the project's classpath.

Chapter 5. Interactions Pane

One of the key distinguishing features of DrJava is its Interactions Pane, which allows you to enter and evaluate Java statements and expressions on the fly. This is remarkably useful for beginning students, who no longer have to learn to write main methods, recompile, and run programs from a command line simply to test how a new class or method behaves. From a teaching standpoint, the Interactions Pane is a very easy way to help students learn to write Java without having to explain the full meaning of syntax like "public static void main", and it also provides an ideal way to perform demonstrations in class. The Interactions Pane can also be used to experiment with your own programs or new libraries, or even to create graphical user interfaces interactively.

How to Use. The Interactions Pane supports the execution of any valid Java statements as well as the evaluation of Java expressions. Simply define variables and call methods as you would in an ordinary method, or even define new classes and methods and call them interactively. In general, any statement or expression ending without a semicolon will display its result in the Interactions Pane, while those ending with a semicolon will complete without displaying a result. Result objects are displayed using the object's toString() method. Any system output will be displayed in the Interactions Pane in green (as well as in the Console tab), while system errors will be displayed in red by default. Any system input will cause a box to be inserted in the Interactions Pane where you can type what you want System.in to read. This text will be colored purple. These colors can be modified in the "Colors" section in the Preferences window.

Here is a simple interactions session, to demonstrate how the Interactions Pane can be used to experiment with objects or show GUI components.

```
Welcome to DrJava.
> String s = "Hello World";
> s
"Hello World"
> s.length()
11
> import javax.swing.*;
> JFrame frame = new JFrame("My JFrame");
> frame.show();
>
```

Intelligent Newlines. DrJava parses your input each time Enter is pressed. If it finds that the input is not complete (unmatched braces or a missing semicolon, for example), it will automatically insert a newline, prompting you for more input on the next line. This feature makes declaring loops, methods, and classes very clean.

Resetting the Interactions Pane. You can reset the Interactions Pane if you wish to start from scratch or if a method call hangs and does not return. Resetting removes any variables from scope and terminates any methods running in the Interactions Pane. To do this, simply choose the "Reset Interactions" command from the Tools menu. This will also reset the Debugger and any JUnit tests that are currently running.

Running the Main Method. For convenience, DrJava supports calling the main method of a class in the Interactions Pane by simply entering "java" followed by the class name and any arguments. For example, to run MyClass with args "arg1" and "arg2", type the following into the Interactions Pane:

```
java MyClass arg1 arg2
```

Note that this feature does not support passing special flags or arguments to the JVM itself, as is supported on the command line.

Another shortcut for this feature is the "Run Document's Main Method" command, which can be found in both the Tools menu and the context menu of the document list. This command will simply insert the appropriate `"java MyClass"` text into the Interactions Pane to run the current class's main method.

DrJava also displays either a "Run Project" or a "Run" button in its toolbar, depending on whether you have specified a "Main Class" for the project or not, respectively. "Run Project" will run the main method of the project's "Main Class", while "Run" while execute the main method of the currently open document.

If "Smart Run Command" option is enabled, DrJava will also run applets and ACM Java Task Force programs using the "Run" or "Run Project" buttons (see Configuring DrJava).

The same smart behavior of running regular Java classes, applets and ACM Java Task Force programs can also be achieved by entering "run" followed by the class name and any arguments in the Interactions Pane. For example, to run MySomething with args "arg1" and "arg2", type the following into the Interactions Pane:

```
run MySomething arg1 arg2
```

Running the Document as Applet. For users who write Java applets, DrJava has a built-in applet viewer that supports calling the `run()` method of a class in the Interactions Pane by simply entering "applet" followed by the class name. Any arguments will be passed to the constructor as strings. For example, to create `MyApplet(String a, String b)` with arguments "arg1" and "arg2" and then call the object's `run()` method, type the following into the Interactions Pane:

```
applet MyApplet arg1 arg2
```

Another shortcut for this feature is the "Run Document as Applet" command, which can be found in both the Tools menu and the context menu of the document list. This command will simply insert the appropriate `"applet MyApplet"` text into the Interactions Pane to run the current document as applet.

Keyboard Shortcuts. Many actions in the Interactions Pane have keyboard shortcuts for ease of use. Use the Up and Down arrow keys to scroll through a history of the previously entered commands, or Ctrl +B to clear the current command. You can also use Shift+Enter to insert newlines into statements in the Interactions Pane. DrJava also now supports searching backwards through history. To use this feature, type in the first few characters of the command you wish to repeat and hit the Tab key. The last command that matches the characters you typed will appear. Hitting Tab repeated searches farther back, while Shift-Tab will move you forward in the history.

Setting the Classpath. To interact with any class within the Interactions Pane, it must be included on the Interactions Classpath, which can include more than the user's own classpath. Any class which is opened in the Definitions Pane of DrJava is automatically added to this classpath, but additional classes and directories can be added using the "Extra Classpath" configuration option. (See Configuring DrJava.) The current classpath of the Interactions Pane can be viewed at any time by selecting "View Interactions Classpath" from the context menu.

Saving the Interactions History. You can save all of your past interactions to a file at any time, using the "Save Interactions History" command in the Tools and popup menus. This command gives you the option to edit any part of the history before saving it, through a separate window that supports editing. By default, up to 500 of the most recent Interaction commands are kept in this history, though this number is configurable. Histories are saved in files with a `.hist` extension, and they can be later executed in the Interactions Pane with the "Execute Interactions History" command in the Tools menu. Saving and executing histories can be particularly useful for initial setup of an often repeated task, such as importing several packages and initializing frequently used variables. To help manage the history, a "Clear Interactions History" command is also provided in the Tools menu.

Loading a History File as a Script. You can load a history file as a script that can be executed one line at a time, using the "Load Interactions History as Script" command in the Tools and popup menus.

A panel will appear on the right side of the Interactions Pane with buttons that allow you to display the previous interaction, execute the current interaction, display the next interaction, and close the panel. This feature is useful during presentations because you can step through a series of interactions that has been prepared in advance, allowing the audience to see the result of each interaction.

Lifting Interactions to Definitions. One common use of the Interactions Pane is to test a line of code intended for a program, even before it is written in the program itself. For example, this can be the case when experimenting with method calls to determine their results. In this situation, it is convenient to copy a working line from the Interactions Pane into the Definitions Pane. This can be done quickly with the "Lift Current Interaction to Definitions" command in the Tools menu, which simply copies the text at the current prompt and pastes it at the cursor position in the Definitions Pane.

Context Menu. The Interactions Pane has a context menu, which can be used by right-clicking in the pane. (Mac users should use Ctrl+Click or Option+Click.) This menu provides shortcuts to useful commands for the Interactions Pane, including cut, copy, and paste, as well as resetting the Interactions Pane, executing, loading, and saving history files, viewing the current classpath, and copying the current interaction to the Definitions Pane.

Tiger Features. DrJava provides support for Tiger (JDK 1.5) features in the interactions pane. These include enum types, static import, for each statements, methods with variable arguments, generics, and autoboxing. Note that you must be running Java 2, version 1.5.0 or later to use 1.5 features in the Interactions Pane. Also, DrJava does not currently type check generics, so while they may be used in the interactions pane without a syntax error, we do not yet provide full support for type checking of generics.

System.in and Closing the Input Stream

Your program can ask for text input from the user by invoking `System.in`. You can also use the `System.in.read()` method in the Interactions Pane directly. When the input box appears, type your text and then either press Return.

You can choose to close the input stream by selecting the menu item "Tools, Interactions & Console, Close System.in", or by pressing the keyboard shortcut for it, which is `Ctrl-D`. The shortcut is labeled `Close System.in` in the Key Bindings section of the preferences.

Here is an example of closing the input stream. The text in square brackets was entered by the user.

```
Welcome to DrJava.  Working directory is /Users/Shared/drjava
> System.in.read()
 [1]
49
> System.in.read()
10
> System.in.read() // press Ctrl-D now
 []
-1
>
```

The user first types '1' and then presses Return. This lets DrJava read a 49, which is the ASCII code for the character '1', and then 10, which is the ASCII code for the new line created by Return. In the second input box, the user pressed Ctrl-D immediately to close the input stream. This lets DrJava read -1, indicating of the end of the stream.

Imports in the Interactions Pane

Auto-Import. When you use a class name in the Interactions Pane that is not yet known, probably because the class or its package have not been imported yet, DrJava displays an error:

```
Welcome to DrJava.  Working directory is /Users/Shared/drjava
> File f
Static Error: Undefined class 'File'
```

At this time, DrJava will open a predictive input dialog populated with all Java API classes. The initial suggestion is a class that matches the unknown class as closely as possible. After choosing the desired class, DrJava will import that class and re-execute the line that caused the error:

```
Welcome to DrJava.  Working directory is /Users/Shared/drjava
> File f
Static Error: Undefined class 'File'
> import java.io.File; // auto-import
File f
>
```

In the predictive input dialog, there is also a checkbox that allows you to import the entire package that contains the selected class, e.g. java.io.* instead of just java.io.File:

```
Welcome to DrJava.  Working directory is /Users/Shared/drjava
> File f
Static Error: Undefined class 'File'
> import java.io.*; // auto-import
File f
>
```

Default Imports. In DrJava's preferences under "Miscellaneous", the user can add or remove classes and packages that the user would like to automatically import in the Interactions Pane whenver the pane has been reset. After the Interactions Pane has been reset, these classes and packages are immediately available.

Chapter 6. Compiling Programs

Java compilers check your programs for errors and translate them to class files which can be used. Any time you change the source file for a class, it must be compiled before it can be used. To do this in DrJava, you can simply click on the "Compile All" button on the toolbar to compile any open documents. Any resulting errors will be highlighted in the document.

Compiling Files

To compile the documents you have open in DrJava, click on the "Compile All" button on the toolbar, or choose either "Compile All" or "Compile Current Document" from the Tools menu. Before the compilation can begin, all open files must be saved, even if only the current document is being compiled. This is necessary because one file can depend on other files, and it is important that no files have been modified when errors are displayed. (Otherwise, an error could be highlighted on a line which has changed.) Once a compilation completes successfully, the Interactions Pane is reset so that the new class files can be used. The output on the Console Tab is also reset to begin the new session, unless the "Clear Console After Interactions Reset" option in the "Miscellaneous" section of the Preferences is unchecked.

In project mode, you have the option to compile all project source files, even if those files are not currently open in DrJava. To compile all source files, click "Compile Open Project Files" in the Project Menu. This option is also available by right clicking the root of the project tree.

Viewing Compiler Errors

If the compiler finds any errors in your program, DrJava displays them in the Compiler Output tab at the bottom of the window. A summary of each error is displayed in the list, including the file name and line number. You can click on any error to highlight the corresponding line in the document itself. (Note that a file will be opened automatically if it contains errors detected by the compiler.) Similarly, if the cursor is moved to a line of code that contains an error while the Compiler Output tab is displayed, that line and the corresponding error message are highlighted. You can disable highlighting compiler errors in the source by unchecking the "Highlight Source" checkbox on the Compiler Output tab, or by closing the Compiler Output tab.

Selecting a Compiler

DrJava supports the use of different Java compilers, such as different versions of the "Javac" compiler supplied with the JDK. DrJava will attempt to locate the your Java compiler on startup by searching for standard installation directories, but sometimes it is unable to find a compiler. In this case, it will use the included Eclipse compiler instead. Note that the location of the compiler can be specified in the Preferences (see Configuring DrJava). If more than one compiler is specified, the active compiler can be selected from a menu on the Compiler Output tab itself.

DrJava will only display one compiler for each major version even though you have more than one update the JDK installed (Example: you have JDK 6 Updates 10 and 14 installed; DrJava will only display JDK 6 Update 14). If you want to display all compiler versions that were found, enable "Show all compiler versions" in the Preferences (see Configuring DrJava).

The preferred compiler can also be selected permanently by setting the Compiler Preference in the Compiler Options category of the Preferences. DrJava will then choose the selected compiler at startup.

Chapter 7. Testing using JUnit

While compilers can look for structural problems in a program, they cannot tell whether the results of a program or method are correct. Instead, all developers test their programs to ensure that they behave as expected. This can be as simple as calling a method in the Interactions Pane to view its results, but this technique requires you to think about the answers you expect every time you run any tests. A much better solution is to give the tests the answers you expect, and let the tests themselves do all the work.

Thankfully, a technique known as unit testing makes this quite easy. You write many small tests that create your objects and assert that they behave the way you expect in different situations. A unit test framework known as JUnit (http://www.junit.org) automates the process of running these tests, letting you quickly see whether your program returns the results you expect.

DrJava makes the process of running unit tests very simple by providing support for JUnit. Once you have written a JUnit test class (as described in the next section), you can simply choose the "Test Current Document" command from the Tools menu to run the tests and view the results. The name of the tests being run will be shown in the Test Output tab, with each test method turning green if it completes successfully and red if it fails. Any failures will be displayed after the list of methods in the same way as the compiler errors. A progress bar will keep you updated on how many tests have been run.

Also, clicking the "Test" button on the toolbar or choosing "Test All Documents" from the Tools menu will run JUnit on any open testcases, making running multiple test files very simple.

Writing Unit Tests with JUnit

With the JUnit framework, unit tests are any public classes that extend the `junit.framework.TestCase` class, and that have any number of methods with names beginning with the word `"test"`. JUnit provides methods to easily assert things about your own classes, as well as the ability to run a group of tests.

The requirements for writing unit test classes are described below, with an example provided in the next section. In general, though, the intent is for you to create instances of your classes in the test methods, get results from any methods that you call, and assert that those results match your expectations. If these assertions pass, then your program has behaved correctly and your tests have succeeded.

Writing a Test Case. To use DrJava's Test command on a document, you must use the programming conventions outlined below. You can also choose the "New JUnit Test Case" command from the File menu to automatically generate a template with these conventions.

- At the top of the file, include:

  `import junit.framework.TestCase;`

- The main class of the file must:

  - be `public`

  - extend `TestCase`

- Methods of this class to be run automatically when the Test command is invoked must:

  - be `public` and *not* `static`

  - return `void`

- take no arguments

- have a name beginning with `"test"`

- Test methods in this class can call any of the following methods (among others):

  - `void assertTrue(String, boolean)`

    which issues an error report with the given string if the boolean is false.

  - `void assertEquals(String, int, int)`

    which issues an error report with the given string if the two integers are not equal. The first int is the expected value, and the second int is the actual (tested) value. Note that this method can also be called using any primitives or with Objects, using their `equals()` methods for comparison.

  - `void fail(String)`

    which immediately causes the test to fail, issuing an error report with the given string.

  Test methods are permitted to throw any type of exception, as long as it is declared in the `"throws"` clause of the method contract. If an exception is thrown, the test fails immediately.

- If there is any common setup work to be done before running each test (such as initializing instance variables), do it in the body of a method with the following contract:

  ```
  protected void setUp()
  ```

  This method is automatically run before any tests in the class. (Similarly, you can write a `protected void tearDown()` method to be called after each test.)

- If you would rather control which methods are called when running the tests (rather than using all methods starting with `"test"`), you can write a method to create a test suite. This method should be of the form:

  ```
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new <classname>("<methodname>"));
    ...
    return suite;
  }
  ```

  It is then also necessary to import `TestSuite` and `Test` from `junit.framework`. There is also a version of the `addTest` method that takes a `Test`, so test suites can be composed.

A simple example of a TestCase class is given in the next section. There are many other ways to use JUnit, as well. See the JUnit Cookbook at http://junit.sourceforge.net/doc/cookbook/cookbook.htm for more examples and information.

Simple Test Example

Suppose you are writing a Calculator class that can perform simple operations on pairs of integers. Before you even write the class, take a moment to write a few tests for it, as shown below. (By writing tests early, you start thinking about which cases might cause problems.) Then write the Calculator class, compile both

classes, and run the tests to see if they pass. If they do, write a few more test methods to check other cases that you have realized are important. In this way, you can build up programs with a great deal of confidence.

```
import junit.framework.TestCase;
public class CalculatorTest extends TestCase {

  public void testAddition() {
    Calculator calc = new Calculator();
    // 3 + 4 = 7
    int expected = 7;
    int actual = calc.add(3, 4);
    assertEquals("adding 3 and 4", expected, actual);
  }

  public void testDivision() {
    Calculator calc = new Calculator();
    // Divide by zero shouldn't work
    try {
      calc.divide(2, 0);
      fail("Should have thrown an exception!");
    }
    catch (ArithmeticException e) {
      // Good, that's what we expect
    }
  }

}
```

Viewing Test Failures

If one or more test methods in a JUnit test class fails, each one is displayed in the Test Output tab at the bottom of the window. This list of failures is similar to the list of compiler errors, in that a summary of the error is given in the tab, and clicking on it highlights the corresponding line in the file (as long as the "Highlight Source" checkbox is checked). Note that DrJava displays a warning message if the test class has been modified since the last time it was compiled, since the changes will not be reflected in the behavior of the test. Closing the Test Output tab resets the current set of JUnit failures.

Aborting Tests. If a suite of tests takes a long time or goes into an infinite loop, you can abort the tests by choosing the "Reset Interactions" command from the Tools menu. An error will be displayed in the Test Output tab showing that the tests were aborted.

Viewing the Stack Trace. When a JUnit test fails or throws an exception, it is sometimes helpful to view the entire stack trace when diagnosing the problem. To view the stack trace for any test failure, right click on the failure in the Test Output tab and select "Show Stack Trace."

Chapter 8. Language Level Facility

The Java Language Level Facility provides a student-friendly introduction to Java. As of April 2010, there is a single, simplified language level called *Functional Java*. When used with a complementary curriculum, this language level forms a powerful learning tool.

In the past, there are three levels: *Elementary*, *Intermediate*, and *Advanced*, and each level progressively introduces students to more Java features. Files written using these three language levels can still be compiled, but not be created anymore. The new Functional Java language level can compile files written for the old Elementary and Intermediate language leves; the old Advanced level can be compiled by the regular Java compiler.

Using the Java Language Level Facility

The Java Language Level Facility is fully integrated into DrJava. To select the level at which you wish to work, click on the "Language Levels" menu in the menu bar and select a level. If you do not want to use any Language Level, select "Full Java".

Selecting your level affects how new files are saved. Each file's extension specifies its level. Functional Java files are given the *.dj* extension; Full Java files have the usual *.java* extension.

In the past, files using the Elementary, Intermediate, and Advanced Levels had the *.dj0, .dj1* , and *.dj2* extensions, respectively. These files can still be loaded and compiled, but DrJava suggests that you rename .dj0 and .dj1 files to .dj files, and .dj2 files to .java files.

You can reference classes defined at any language level regardless of the level you are currently using, but you cannot reference Full Java files (or old Advanced Level files) unless you have first compiled them seperately.

Because each level is a restricted subset of Full Java, instructors must be careful when allowing their students to use library functions. For instance, at the Elementary level, "null" is not a valid keyword, so functions that may return null should not be used. Similar caution must be exercised with arrays. Arrays are not introduced until the Advanced level, so library functions that rely on arrays should not be used until then. Also, although we do not allow the mutation of fields or variables at the Elementary and Intermediate levels, we do not prohibit the use of library classes with their own mutation methods such as `java.util.LinkedList`. Because of this, the students and their teachers must be careful to not use mutable datatypes.

Internally, the Language Level Facility translates *.dj*, *.dj0*, *.dj1* and *.dj2* files to *.java* files with the same name. For example, the file `Example.dj` creates the hidden file named `Example.java`. Please make sure that you do not have both a Java Language Level file and a *.java* file with the same name, as this would overwrite your *.java* file. We also advise against opening the generated *.java* files. If you do open them, DrJava will warn you when you compile and ask you to close the *.java* files.

What Does the Functional Java Level Provide?

The Functional Java language level roughly corresponds to the old Intermediate Level. More details to follow.

What Did the Old Levels Provide?

The general presentation of concepts in the old Language Levels Facility corresponds to the book *How To Design Programs* by Felleisen, Findler, Flatt, and Krishnamurthi. Basically, the levels provide an

incremental introduction to data structures and operations on those structures while helping to simplify syntax for the students.

The levels are also designed to allow the introduction of *design patterns*--models for writing code with certain abstract behavior that are essential to teaching elegant object-oriented programming.

Here is a summary of what is allowed at each of the three old language levels.

Language Features in the DrJava Language Level Facility				
Language Features	Elementary	Intermediate	Advanced	Full Java
Classes (concrete and abstract) Non-void methods (concrete and abstract) Special support for Junit test classes Implicitly final fields and variables **int, double, char, and boolean types** Operators (except bitwise operators) **if** statement Automatic convenience method generation Autoboxing (if compiler-supported)	■	■	■	■
Explicit constructor declarations **import** statements **static** fields Method, class visibility can be specified Casts **null** value Exceptions and **try-catch** statements Anonymous inner classes Interfaces		■	■	■
package statements Mutable fields and local variables Explicit use of **final** modifier Nested classes and nested interfaces Field visibility can be specified **while, for,** and **do** loops **switch** statement **void** methods Arrays **break** and **continue** keywords			■	■
All other primitive types Static or instance initialization blocks **native** methods **synchronized, volatile, Thread** classes Bitwise operators Labeled Statements Conditional Operator **instanceof** Operator Generic types and other Java 1.5 features				■

And here is a summary of the code augmentation performed at each level.

Code Augmentation in the DrJava Language Level Facility			
Code Augmentation	Elementary	Intermediate	Advanced
Special support for Junit test classes	▩		
Methods public by default Fields, variables automatically final Constructor generation **toString()** generation **equals(...)** generation **hashCode()** generation Accessor generation Instance fields automatically private	▩	▩	
Static fields automatically public		▩	

Read on for a more detailed explaination of each level.

The Old Elementary Level

The old Elementary Level provides support for programming in functional Java, which can be taught with only algebraic data types--types that are inductively defined, such as *integers*, *booleans*, *lists*, and *trees*. Because of this, only a small subset of the Java language is necessary. Most importantly, all fields and variables are immutable; in other words, they are automatically made final, so their values cannot be changed once they are set.

Because of this immutability of data, *for*, *while*, and *do* loops cannot be used at the Elementary level. We also do not allow arrays because they are commonly used with a procedural-style approach to programming (loops) rather than an object-oriented one and are inherently mutable. And since void return methods have few uses without mutable data, they are only allowed in JUnit test cases at this level. We also simplify the language for the student by not allowing *null* as a keyword. This protects beginning students from getting null-pointer exceptions but also means that instructors should not allow the students to use library functions that return null. In addition, interfaces are not allowed at the Elementary Level; only classes and abstract classes are allowed. By waiting to introduce interfaces until after the students are familiar with abstract classes, we hope students will have an easier time differentiating between interfaces and abstract classes. We also disallow the use of explicit access modifiers (*final, private, static,* etc.), and instead automatically make all fields and local variables private and all methods public. The one exception to this is that classes and methods can be denoted as *abstract*. We also do not allow the use of *package* or *import* statements in order to simplify the concepts that beginning students must learn. Of course, students' classes can still reference other files in the same directory, and they also have access to all classes in the *java.lang* package, and if students extend "TestCase" we automatically import *junit.framework.TestCase* for them.

We automatically generate a *constructor* for each class that students write at the Elementary Level. Each class's constructor takes a value for each of its fields, and sets the fields to those values. None of a class's fields can be set outside of the automatically generated constructor. We consider fields to be any field in the class or one of its superclasses that also has a visible accessor (a method of the same name as the field that returns its type). We automatically generate *accessors* for each field (for example, field my_field would have an accessor my_field()). We also generate a *toString()* method that returns a description of the object--its class name and field values, an *equals()* method that determines if two objects are equal by comparing their class types and the values for each of their fields, and a *hashCode()* method that follows the Java Language Specification that if two objects are equal, their hash codes are the same. Students cannot override any augmented methods at this level.

Although at first glance this level seems limited, even the basic functionality provided is powerful and flexible. The *Composite, Union, Interpreter,* and *Factory* patterns can all be taught at the Elementary level.

The Old Intermediate Level

At the old Intermediate level, the focus is still on functional programming with immutable data. However, there is an added twist: functions can now be used as data. Because of this, we introduce anonymous inner classes as a new construct. These anonymous inner classes can be stored in variables and passed as arguments to methods.

Although we allow anonymous inner classes, students still cannot use nested classes or nested interfaces at this level. They introduce a level of complexity in naming and referencing that is best deferred to the Advanced Level.

In addition to anonymous inner classes, several new concepts are introduced at this level. We now allow interfaces, which should be intuitive to students since they have been working with abstract classes at the Elementary Level. In addition, we allow *package* and *import* statements to broaden the scope of classes the student has access to (including the Java libraries), and to help them learn how to modularize their own projects. We allow the *null* keyword to be used and also allow explicit visibility specifiers such as *public*, *private*, and *protected* for all constructs except fields and variables, and the keyword *static*. However, only fields can be static at this level; static methods are still prohibited. We also introduce casts because they are useful with the *Visitor* design pattern--frequently the arguments to and return type of visitors is Object, and if a more specific contract for the specific function is known, the data can be cast. Students can also define their own constructors at this level, though they must make sure that all of a class's non-static fields are given a value in the constructor. Non-static fields still cannot be assigned outside of a constructor, and static fields must be given a value where they are defined. All fields are still made final.

The code augmentation is the same as that done at the Elementary Level, except that if a student defines a constructor that takes in all the fields of the class, we do not generate a duplicate constructor, and accessors are not generated for static fields, so static field values are not included in the constructor, equals(), toString(), and hashCode() methods. Students cannot override any autogenerated methods besides the constructor.

Both the *Command* and *Visitor* design patterns deal with passing functions as data and should be taught at this level. The *Singleton* design pattern can also be taught here.

The Old Advanced Level

As students move to the old Advanced level, they are introduced to mutable data for the first time. This change in perspective allows us to make several extensions to the language.

The introduction of mutable data goes hand in hand with the explicit use of the *final* keyword--students must now mark the data they do not want to be mutable as "final". Explicit visibility modifiers may now be used for fields and variables. *For*, *while*, and *do* loops, as well as *switch* statements are now allowed, though students cannot make assignments in the conditional expression of the loops and the switch expression of the switch statement. This should help students avoid a common error of using "=" instead of "==". Students can get around this restriction by nesting the assignment in parentheses--for instance ((i=5)) rather than (i=5). *Break* and *continue* statements are also allowed because they are useful with loops. We restrict *switch* statements further by not allowing fall through in any switch case, including the last one, and by enforcing that if there is a default case, it must be the last case in the block. A label in a switch statement can only be a character constant, integer constant, or a negative sign followed by an integer constant--not an arbitrary constant expression. Mutable data also means that arrays are now a useful data type, so we support the use of arrays at this level. And, mutable data means that *void* return methods make sense in some cases, so they are also allowed.

The language is more flexible in other areas at this level as well. We allow non-static fields to be assigned a value where they are declared, thus giving students more freedom in what arguments need to be passed to the constructor. In addition, we allow static methods as well as static fields at the Advanced level. Nested classes and interfaces, both static and dynamic, are also supported at the Advanced level.

No code augmentation is done at the Advanced Level.

The *Strategy*, *State*, *Decorator*, and *Model-View-Controller* design patterns can all be taught at this level.

Please note that this final level is still under development and may not be fully functional. It will be finished soon.

Chapter 9. Debugger

DrJava provides advanced tools for debugging your programs in the Interactions Pane. You can set breakpoints in source files in the Definitions Pane, call methods that stop at the breakpoints in the Interactions Pane, and then interact with programs while they are suspended at breakpoints. Once a breakpoint is reached, users can interact with any variables, fields, or methods that are in scope in the suspended method. Users can also resume the method call, or step through it a line at a time. Finally, the values of local variables and fields can be watched in a table as the method call progresses.

Using the Debugger

To use DrJava's debugger, select the "Debug Mode" command from the Debugger menu. An informational panel will be displayed between the Definitions Pane and the Interactions Pane, and several menu items in the Debugger menu will become enabled.

A Note on Modifying Files. When using the debugger, it is essential to remember that any modifications to source files will not be reflected in the behavior of the Interactions Pane or the debugger until the classes are recompiled. Changing a source file while the debugger is running is not recommended, since lines which are highlighted by the debugger may no longer correspond to the lines in the running class file. To help notify you of this danger, DrJava displays a warning message in the Debug Panel if the current document is out of sync with its class file.

Because the debugger depends on the classes used in the Interactions Pane, the debugger is automatically reset each time any files are compiled, or when the Interactions Pane is reset.

Breakpoints

You can set a breakpoint on almost any line of code in a source file in the Definitions Pane, using either the "Toggle Breakpoint on Current Line" command in the Debugger menu, the "Toggle Breakpoint" command on the context (right-click) menu in the Definitions Pane, or by pressing Ctrl+B. When a breakpoint is set, the line will be highlighted in red and an entry will appear in the "Breakpoints" panel, which can be displayed using the "Breakpoints" command in the Debugger menu, or by pressing Ctrl+Shift+B. When a method is called in the Interactions Pane and the control flow reaches a line of code where a breakpoint has been set, DrJava suspends the execution of the program, highlights the line in bright blue and prints a message to the Interactions Pane. DrJava then displays a new prompt in the Interactions Pane, allowing you to interact with the suspended program until it is resumed (see Interactions at a Breakpoint). Breakpoints are considered part of a project and are therefore saved to and loaded from a project file.

When setting breakpoints, it is important to remember that only lines with actual executable code can be used. Blank lines and comments will never trigger a breakpoint, even if the line is highlighted in red. (Note that we do not yet support breakpoints on method contracts either, although this will be supported in a later version of DrJava.)

Breakpoints Panel. The "Breakpoints" panel can be displayed using the "Breakpoints" command in the Debugger menu, or by pressing Ctrl+Shift+B. It lists all breakpoints that have been set in the currently open documents, sorted by document and line number. There are several buttons on the right side of the panel that help you manage the breakpoints:

You can select one or more breakpoints and press the "Enable" or "Disable" button to enable or disable the selected breakpoints. When a breakpoint is disabled, it remains set, but the program will not be suspended when the breakpoint is reached. This is useful if you may need the breakpoint again later, but want the program to ignore it right now.

If you select exactly one breakpoint, you can use the "Go to" button to jump to the location associated with the breakpoint.

With the "Remove" button, you can remove one or more breakpoints that you have selected. You may also clear the entire list using the "Remove All" button.

Debugging JUnit Tests. DrJava will also stop at breakpoints during JUnit tests. Simply set a breakpoint on a line of a test method or in a method called by a test, and then choose the "Test Using JUnit" command from the Tools menu. When control flow reaches the breakpoint, the test will be suspended.

Interactions at a Breakpoint

When DrJava reaches a breakpoint during a method call, it prints a new prompt to the Interactions Pane. This new interpreter has the context of the program which has been suspended, including all variables, fields (even "`this`"), and methods that are in scope in the suspended method. (The prompt text itself contains the name of the thread which has been suspended. In most cases, this name will include the text being interpreted.) You can type the name of any variable or field to view its value or assign it a new value, or you can call any method in scope to observe its behavior. Existing lines of code in the program cannot be changed or skipped, however, and the "`return`" keyword is not available. Any changes you make to variables or fields will be reflected in the program when it resumes execution, either by stepping or resuming.

Stepping and Resuming

After DrJava reaches a breakpoint, the method being called is suspended, and several commands in the Debug Menu and Debug Panel become available. Choosing "Resume" allows the current method to finish execution, at least until another breakpoint is reached. If any other threads are suspended when you resume, the Interactions Pane will switch to the most recently suspended thread. Otherwise, the original ("top level") prompt is restored. Alternatively, you can use the Step commands in the Debug menu to step through the execution of the method, one line at a time. Each time a step completes, the debugger highlights the next line of code to be executed. If the code is in another file, the debugger will look for the file on the Sourcepath and attempt to open it.

Step Into. The Step Into command will walk into any method calls that occur in the code, possibly opening additional files.

Step Over. The Step Over command will not enter any new method calls, treating them instead as a single operation to be stepped over.

Step Out. The Step Out command will finish running the current method and stop at the next line of code from where the method was called.

Sourcepath and Step Options. The sourcepath is the set of directories in which to look for source files when stepping. It can be set in the Debugger section of the Preferences window (which can be opened from the Edit menu). This section in the Preferences also contains options for controlling which classes will be included as part of stepping. By default, DrJava will never step into its own source, nor its Java Interpreter (DynamicJava), nor Java itself. If you are interested, and have downloaded the source files, you can enable these options to see how Java or DrJava works. You can also specify which classes and packages you want to exclude when stepping. To exclude specific classes, type in the qualified class name (the package name followed by a period and the class name). To exclude entire packages (as well as their subpackages), type the package name followed by a period and an asterisk. Each class or package name must be separated by a comma.

Automatic Trace. The Automatic Trace command allows the user to execute periodic "Step Into" commands, by default every 1000 milliseconds. This will cause DrJava to execute the program line by

line, entering called methods. After such a periodic step, the user has the option oof disabling the automatic trace by pressing the "Disable Automatic Trace" button. If the automatic trace is not disabled, the program will run its course, and the automatic trace will be turned off at the end of the program. The user can change the delay interval for stepping located in DrJava's preferences.

Debug Panel

The Debug Panel appears when Debug Mode is on, with several informational tabs and buttons. DrJava displays currently watched fields and variables with their values in a table in the Watches tab. The Stack tab displays the current stack trace any time a method has been suspended, and the Threads tab shows all current threads along with their status at that point in time. Most of these tabs provide context (right-click) menus for easy access to related commands, such as scrolling to a line in a stack frame, or resuming a suspended thread.

Watching Fields and Variables. You can watch the values of class fields and local variables by entering the field or variable name into a row in the Watches table. Any time a method is suspended (e.g. after a breakpoint or step), the current value of the field or variable (if any) is displayed. Watches can be removed from the table by deleting the name and pressing Enter. You cannot enter expressions that need to be evaluated into the watch table. For example, "s.length" is not a valid watch entry. Type expressions like these into the Interactions Pane to see their values.

Detachable Debug Panel

When the Debug Panel is visible, it is normally attached to DrJava's main frame and displayed just above the bottom panes, and below the editor pane. To conserve screen space or make better use of dual monitors, the Debug Panel can also be detached.

To do this, click on the "Detach Debugger" menu item in the "Debug" menu and make sure that a checkmark appears next to "Detach Debugger". All debugger controls are now displayed in a separate window called "Debugger". To re-attach the "Debugger" window to DrJava's main frame, remove the checkmark next to "Detach Debugger" or close the "Debugger" window.

Chapter 10. Documentation with Javadoc

Documenting your code is crucial to help others understand it, and even to remind yourself how your own older programs work. Unfortunately, it is easy for most external documentation to become out of date as a program changes. For this reason, it is useful to write documentation as comments in the code itself, where they can be easily updated with other changes. Javadoc is a documentation tool which defines a standard format for such comments, and which can generate HTML files to view the documentation from a web broswer. (As an example, see Oracle's Javadoc documentation for the Java libraries at http://download.oracle.com/javase/6/docs/api/.)

You can easily run Javadoc over your programs from within DrJava, using the "Javadoc All Documents" and "Preview Javadoc for Current Document" commands in the Tools menu. These commands will generate Javadoc HTML files from the comments you have written and display them in a browser. This chapter gives a brief overview of these commands and how to write Javadoc comments. More detailed information on writing Javadoc comments can be found online at http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

Writing Javadoc Comments

In general, Javadoc comments are any multi-line comments ("/** ... */") that are placed before class, field, or method declarations. They must begin with a slash and two stars, and they can include special tags to describe characteristics like method parameters or return values. The HTML files generated by Javadoc will describe each field and method of a class, using the Javadoc comments in the source code itself. Examples of different Javadoc comments are listed below.

Simple Comments. Normal Javadoc comments can be placed before any class, field, or method declaration to describe its intent or characteristics. For example, the following simple Student class has several Javadoc comments.

```
/**
 * Represents a student enrolled in the school.
 * A student can be enrolled in many courses.
 */
public class Student {

  /**
   * The first and last name of this student.
   */
  private String name;

  /**
   * Creates a new Student with the given name.
   * The name should include both first and
   * last name.
   */
  public Student(String name) {
    this.name = name;
  }
```

```
}
```

Using Tags. Tags can be used at the end of each Javadoc comment to provide more structured information about the code being described. For example, most Javadoc comments for methods include "@param" and "@return" tags when applicable, to describe the method's parameters and return value. The "@param" tag should be followed by the parameter's name, and then a description of that parameter. The "@return" tag is followed simply by a description of the return value. Examples of these tags are given below.

```
/**
 * Gets the first and last name of this Student.
 * @return this Student's name.
 */
public String getName() {
  return name;
}

/**
 * Changes the name of this Student.
 * This may involve a lengthy legal process.
 * @param newName This Student's new name.
 *                Should include both first
 *                and last name.
 */
public void setName(String newName) {
  name = newName;
}
```

Other common tags include "@throws e" (to describe some Exception "e" which is thrown by a method) and "@see #foo" (to provide a link to a field or method named "foo").

How to Use Javadoc in DrJava

In general, Javadoc HTML files are most useful when they are generated in large batches, since the HTML files for each of the related classes can link to each other. For this reason, DrJava's "Javadoc All Documents" command looks for all source files in the folders and subfolders of the open documents and includes them all in the documentation, saving the files in a "doc" folder nearby. (This folder will be placed either in the current folder or the top-level folder of the current package.) On the other hand, it is occasionally useful to view the Javadoc HTML for a single class, to quickly get a feel for its structure. Therefore, DrJava also provides a "Preview Javadoc for Current Document" command that only generates Javadoc for the current open document without saving it to a permanent location. (This command saves the file in a temporary location that will be automatically deleted when you quit DrJava.) If either of these commands finds errors in the source files, they will report them in a tab like compiler errors.

Viewing Javadoc. When either of the "Javadoc All Documents" or "Preview Javadoc for Current Document" commands complete successfully (or find only warnings and no errors), DrJava displays the resulting HTML files in a new window. For Windows and Mac OS X users, these files will be displayed in the system's default web browser. On other platforms, the files will be displayed in a simple viewer, unless the "Web Browser" option has been configured in the "Resource Locations" section of the Preferences (see Configuring DrJava).

Configuring Javadoc. You can configure many aspects of how Javadoc files are generated. Most prominent is the ability to hide fields and methods below a particular access level (eg. public or

protected). By default, no private fields or methods are shown. Other options include specifying a URL to link to the Java library API (which defaults to Oracle's own website), specifying a default destination directory for all Javadoc files, and specifying any custom parameters to pass to the Javadoc tool itself. Finally, for programs with many nested packages (folders), DrJava provides an option to always generate Javadoc for all packages in the program, rather than just the sub-packages of the open documents.

Java API Javadoc

If you have access to the Java API Javadoc pages, DrJava allows you to quickly open a Javadoc page for a class: With the "Open Java API Javadoc" item in the Tools menu (bound to Shift+F6), you can open a predictive input dialog populated with all Java API classes. After you have selected a class, DrJava will try to open the corresponding Javadoc page in a browser.

The "Open Java API Javadoc for Word Under Cursor" feature, also in the Tools menu and bound to Ctrl +Shift+F6, looks at the word the cursor is on and uses it as a starting point for the search. If there is a unique match, then DrJava will open the Javadoc page immediately; otherwise this feature works just like "Open Java API Javadoc".

Note that this feature requires access to the Java API Javadoc pages, e.g. on the internet. DrJava will use the version and location set for "Java Version for 'Open Java API Javadoc'" in the "Javadoc" pane of the "Preferences" dialog (see Configuring DrJava) and access that location the first time this feature is used during a DrJava session. You can use Java API Javadoc pages on your local system by entering a URL beginning with `file://` followed by the directory the `allclasses-frame.html` file is in. You may also have to set the right values for the "Web Browser" and "Web Browser Command" settings in the "Resource Locations" portion of the "Preferences" dialog.

You can also open the Javadoc pages for JUnit 3.8.2 by entering a URL in the "JUnit 3.8.2 URL" field of the "Javadoc" pane of the "Preferences" dialog. You can open the Javadoc for user-defined libraries by entering the URLs in the "Additional Javadoc URLs" list. Please enter the URL to the directorythat contains the `allclasses-frame.html` file. For example, to open DrJava's Javadoc, enter `http://drjava.org/javadoc/drjava`.

Chapter 11. External Process Facility

DrJava includes an interface to execute external command line programs and supply them with values from inside the DrJava editor.

Executing External Processes

TODO

Simple Programs. TODO

Follow File

Using the "Follow File" option in the "Tools" menu, the user can open a text file in one of the panes on the bottom of DrJava's main frame and keep an eye on how it changes. DrJava checks every once in a while (by default every 300 milliseconds) if the file has changed and updates the display accordingly. The time between the updates and how many lines are displayed (by default only the last 100) can be controlled on the "Miscellaneous" pane of the "Preferences" frame (see Configuring DrJava).

Chapter 12. Other Dialogs

There are several other dialogs in DrJava that you may encounter.

Check for New Version

The DrJava development team works year-around to improve DrJava. To make it easier for users to determine whether there is a newer version available, DrJava now periodically polls the website for the latest version (by default, this is done once a week). If a new version is found, the user can press the "Manual Download" button to be taken directly to the web page with the latest version of DrJava. There is also an "Automatic Update" button that downloads the new version automatically and then restarts DrJava.

The "Check for New Version" dialog appears automatically when DrJava is started and a new version is found, but the user can perform a manual check by selecting "Check for New Version" in the "Help" menu.

Check for (type of new version). The DrJava developers release different kinds of versions: Stable versions, which should not cause major problems; beta versions, which are stable but not yet well tested; and development versions, which contain new features or changes that may be problematic.

By default, DrJava will only notify the user of new stable or beta versions, but this setting can be changed by picking a different "Check for" condition -- the user can check for stable versions only, or be interested in all kinds of versions.

Send System Information to DrJava Developers

It is tremendously helpful for the DrJava developers to know what operating systems and versions of Java are being used. To get a better overview, DrJava now includes a simple and completely anonymous survey.

This survey transmits only the version number of DrJava that is being used, the name and version of the operating system, and the version and vendor of Java that is being used. This information can in no way be traced back to anyone's computer.

Every few months, DrJava will ask the user if these pieces of information may be sent to the DrJava developers.

When DrJava notices that the user's system has changed or that a new version of DrJava is being used, it may also ask the user to participate in the survey. This usually happens automatically when DrJava starts, but the user can also elect to send information by using the "Send System Information" item in the "Help" menu (multiple submissions will be ignored).

Never ask me again. If you do not want to be bothered again by this question, you can check the "Never ask me again" checkbox on the dialog, and DrJava will not ask you to participate in the survey anymore.

Thank you for helping us make DrJava better!

Set File Associations

DrJava can detect if .java, .drjava and .djapp files have not been associated with DrJava. If this is the case, DrJava may display the "Set File Associations?" dialog at startup. This dialog has four buttons.

Yes. Set file associations so .java, .drjava and .djapp files open in DrJava when double-clicked.

No. Do not set file associations.

Always. Set file associations, and do this automatically from now on without asking me again.

Never. Do not set file associations, and never ask me about it again.

These settings can also be changed within DrJava by opening the Preferences window and selecting the "File Types" pane (see Configuring DrJava).

Compiz Detected

DrJava suffers from an incompatibility between the Linux window manager Compiz and Oracle's Swing Java GUI library. We, the developers of DrJava, cannot do anything to fix this problem. We hope that future versions of Java and Compiz will address the incompatibility. In the meantime, we recommend that you disable Compiz if you experience problems. We also suggest that you use the latest versions of Compiz and Java, so you can benefit from possible bug fixes made by Oracle and the Compiz developers. For more information, see http://drjava.org/compiz/.

When DrJava detects that you are using Compiz, it will display the "Compiz Detected" dialog at startup and ask if you want to start DrJava nonetheless. The dialog has three buttons.

Yes. Start DrJava, even though Compiz is being used.

Yes, and ignore from now on. Start DrJava, and never complain about Compiz again.

No. Do not start DrJava.

Whether DrJava should display this warning can be changed by opening the Preferences window and selecting the "Notifications" pane (see Configuring DrJava).

Generate Custom drjava.jar File

This dialog can be found in the Tools, Advanced menu. This dialog lets you generate a custom drjava.jar file based on the currently running version of DrJava, and you can include additional jar files, zip files or directories. These additional files are added to the drjava.jar and are immediately available in the generated DrJava application without having to set up extra classpaths.

If a file is contained in more than one source, the file contained in the first source will be included; conflicting files from sources further down the list will be skipped. Files belonging to DrJava always take precedence.

Note: This implies that DrJava's manifest file will be used.

Please note that the added files may produce a copy of DrJava does not work as intended, and that it will be more difficult for us to help you with these problems. YOU ARE USING THE CUSTOM DRJAVA.JAR FILE AT YOUR OWN RISK.

The dialog contains the following components:

Output Jar File. The file that the custom drjava.jar should be written to.

Additional Sources. The list of additional jar or zip files or directories that should be added to the custom drjava.jar file. If a file is contained in more than one source, the file contained in the first source will be included; conflicting files from sources further down the list will be skipped. Files belonging to DrJava always take precedence.The file that the custom drjava.jar should be written to.

Check Conflicts. Pressing this button causes DrJava to scan itself and the additional sources to identify conflicts. If any conflicts are found, they are listed in a dialog.

Generate. Pressing this button lets DrJava generate the specified output file. It first checks for conflicts, though, and if conflicts are found they are displayed; the user then has the option of aborting the generation or generating anyway.

Close. This closes the dialog.

Appendix A. Configuring DrJava

DrJava has many configurable options which can be set using the Preferences command in the Edit menu, allowing the user to change both DrJava's appearance and behavior. Changes made to the configurable options are saved in a `.drjava` file in the user's home directory. The Preferences window is the preferred method for setting these options, although more experienced users may also edit the configuration file itself.

Preferences Window

The Preferences window is available in the Edit menu, and provides a graphical means to edit all configurable options in DrJava. It displays the options in several categories, each of which can be displayed as a panel of options. Each option has a tool tip with a short description, which can be displayed by holding the mouse arrow over the option.

The Apply button submits the changes on all panels and saves them to the config file, while the OK button does the same and also closes the window. The Cancel button closes the window without applying or saving the changes. Each panel also has a Reset to Defaults button, which resets each of the options on that panel to its original value. Resetting does not take effect until the changes are applied with the Apply or OK buttons.

Editing the Config File

All configured options are stored in the `.drjava` file in the user's home directory. (The location of this file varies on different platforms.) This file is a standard Java properties file, with one option on each line and with option keys and values separated by an equals sign. Any options not defined in this file will have their default value. While it is recommended to use the Preferences window in most cases, the config file can also be edited manually to adjust values as desired. The correct option keys and default values for each option are given in the Available Options section.

Note: All parameters are parsed as standard Java strings, so escape characters must be considered. Notably, to include a Windows-style path in a parameter value, all backslashes must be escaped. For example:

```
javac.location=c:\\jdk6\\lib\\tools.jar
```

Available Options

All available configuration options are displayed in the following sections. The option keys and default values are also provided for users who wish to edit their configuration file.

Resource Locations

These options specify where to find Java resources on your computer, such as compilers or classpath directories.

Web Browser (`browser.file = ""`)	Web Browser (`browser.file = ""`) and
Web Browser Command (`browser.string = ""`)	Web Browser Command (`browser.string = ""`) These two settings allow you to specify how Javadoc files and links from the Help files are opened. On Windows and Mac OS X, we

suggest that you leave both options blank so that the default browser of the OS will be used. On other platforms, e.g. on Linux, you need to set one or both of them to let DrJava successfully open HTML files.

The filename specified as "Web Browser", if one is set, is the executable that will be run. If no text has been entered as "Web Browser Command", then only the URL will be passed as parameter to the executable.

If text has been entered as "Web Browser Command", then any occurrence of "`<URL>`" will be replaced with the URL to open. If "`<URL>`" never occurs. then the URL to open will be appended to the very end.

Note that this means there are several ways of accomplishing the same thing. Let's assume that "`/usr/bin/mozilla`" is the filename of the browser. Then these settings all accomplish the same thing:

- "`/usr/bin/mozilla`" as "Web Browser" and nothing as "Web Browser Command"

- "`/usr/bin/mozilla`" as "Web Browser" and "`<URL>`" as "Web Browser Command"

- Nothing as "Web Browser" and "`/usr/bin/mozilla`" as "Web Browser Command"

- Nothing as "Web Browser" and "`/usr/bin/mozilla  <URL>`" as "Web Browser Command"

Useful settings for Linux: Leave the "Web Browser" setting blank and enter the text specified below as "Web Browser Command".

- Mozilla (if it is already running)

  `mozilla -remote "openurl(<URL>)"`

- Mozilla (if it is not already running)

  `mozilla <URL>`

- Konqueror (the KDE web browser)

  `konqueror <URL>`

Useful settings for Windows: If you do not want the system's default web browser, either specify the executable as "Web Browser" and leave the "Web Browser Command" blank, or leave the "Web Browser" setting blank, and enter the command line as "Web Browser Command". If the web browser's filename contains spaces, then the filename must be surrounded by double quotes in the "Web Browser Command". Example:

• Leave the "Web Browser" setting blank and enter the following text as "Web Browser Command":

    "C:\Program Files\Internet Explorer\iexp

Useful settings for Mac OS X: If you do not want the system's browser, we advise that you use the "Web Browser Command" and leave the "Web Browser" setting blank. If possible, use Mac OS X's "open" command as in the examples below:

• Open in Safari: Leave the "Web Browser" setting blank and enter the following text as "Web Browser Command":

    open -b com.apple.Safari <URL>

• Open in TextEdit: Leave the "Web Browser" setting blank and enter the following text as "Web Browser Command":

    open -b com.apple.TextEdit <URL>

Tools.jar Location (`javac.location = ""`)

Tools.jar Location (`javac.location = ""`)
Specifies the location of the JDK's `tools.jar`, which contains the classes necessary for the compiler and the debugger. This file is usually found in the JDK's `lib` directory.

Display All Compiler Versions
(`all.compiler.versions = false`)

Display All Compiler Versions
(`all.compiler.versions = false`)
By default, DrJava only displays one compiler per major version, even if multiple updates are found (Example: You have JDK 6 Updates 10 and 14 installed; DrJava will only display JDK 6 Update 14). To display all compiler versions, mark this checkbox. Note: You have to restart DrJava when you change this setting.

Extra Classpath (`extra.classpath = ""`)

Extra Classpath (`extra.classpath = ""`)
Used to specify any directories or `jar` files to append to the classpath of the Interactions window and the compiler. Separate the directories using the system-specific path separator (eg. colon on Unix, semicolon on Windows).

Display Options

These configurable options affect how DrJava's user interface is displayed.

Look and Feel (`look.and.feel = ""`)

Look and Feel (`look.and.feel = ""`)
Name of the Swing LookAndFeel class which determines the general appearance of DrJava. If this option is changed while DrJava is running, the changes will not apply until you restart.

Plastic Theme (`plastic.theme = "DesertBlue"`)

Plastic Theme (`plastic.theme = "DesertBlue"`)
If Plastic is selected as Look and Feel, then this setting changes the theme that is used to display DrJava. If Plastic is not selected, changing this setting has no effect. If this option is changed while DrJava is running, the changes will not apply until you restart.

Toolbar Buttons (`toolbar.icons.enabled = true, toolbar.text.enabled = true`)

Toolbar Buttons (`toolbar.icons.enabled = true, toolbar.text.enabled = true`)
These radio buttons control whether the toolbar buttons contain text, icons, or both. When set manually in the config file, each of the two options can be set to true or false, though icons will be displayed if both are set to false.

Show All Line Numbers (`lineenum.enabled = false`)

Show All Line Numbers (`lineenum.enabled = false`)
Whether to display all line numbers along the left margin of the Definitions Pane.

Show sample of source code when fast switching (`show.source.for.fast.switch = true`)

Show sample of source code when fast switching (`show.source.for.fast.switch = true`)
Whether a sample of the source code around the current caret position should be shown in the Fast Switch window.

Show Code Preview Popups (`show.code.preview.popups = false`)

Show Code Preview Popups (`show.code.preview.popups = false`)

	Whether a sample of the source code around the document location should be shown in the Breakpoints, Bookmarks and Find Results panes.
Size of Clipboard History (`clipboardhistory.store.size = 10`)	Size of Clipboard History (`clipboardhistory.store.size = 10`) How many entries are saved in the clipboard history.
Display Fully-Qualified Class Names in "Go to File" Dialog (`dialog.gotofile.fully.qualified = false`)	Display Fully-Qualified Class Names in "Go to File" Dialog (`dialog.gotofile.fully.qualified = false`) Whether the "Go to File" dialog should also include the fully-qualified class names. Example: There is a file myPackage/ MyClass.java (in the myPackage package). With this setting enabled, the "Go to File" dialog will contain both MyClass.java and myPackage.MyClass.java; with the setting disabled, it will only contain MyClass.java.
Scan Class Files For Auto-Completion After Each Compile (`dialog.completeword.scan.class.files = false`)	Scan Class Files For Auto-Completion After Each Compile (`dialog.completeword.scan.class.files = false`) When this option is enabled, a project is open, and a build directory has been set, DrJava will scan all class files after each compile and add their names to the auto-completion list. This allows DrJava to auto-complete the class names of all user classes, not just the names of the open document. This option requires additional disk accesses and therefore slows down compiles.
Consider Java API Classes for Auto Completion (`dialog.completeword.javaapi = false`)	Consider Java API Classes for Auto Completion (`dialog.completeword.javaapi = false`) When this option is enabled, DrJava will include the names of the standard Java API classes in the list of names used for auto-completion.
Display Right Margin (`display.right.margin = true`)	Display Right Margin (`display.right.margin = true`) Enable this option to let DrJava display a vertical line representing the right margin of the document.
Right Margin Position (`right.margin.columns = 120`)	Right Margin Position (`right.margin.columns = 120`)

This option controls the position of the right margin. By default, the right margin line is displayed after 120 columns, provided the "Display Right Margin" option above is enabled.

Font Options

Each font option is specified as a string containing the font name, style, and size, separated by dashes. The style should be in upper-case (ie. PLAIN, BOLD, ITALIC, or BOLDITALIC), while the font name must be a valid font on the system. (In most cases, using the font chooser in the Preferences window is the simplest approach.)

Main Font (`font.main = Monospaced-PLAIN-12`)

Main Font (`font.main = Monospaced-PLAIN-12`)
This font is used for the definitions pane and the tabs at the bottom of the window.

Line Numbers Font (`font.doclist = Monospaced-PLAIN-12`)

Line Numbers Font (`font.doclist = Monospaced-PLAIN-12`)
This font is used for the line numbers on the left side of the Definitions Pane, if the "Show All Line Numbers" option in the "Display Options" section is enabled. The actual font size will be limited by the size of the Main Font.

Document List Font (`font.doclist = Monospaced-PLAIN-10`)

Document List Font (`font.doclist = Monospaced-PLAIN-10`)
This font is used in the list of all open documents on the left side of the window.

Toolbar Font (`font.toolbar = dialog-PLAIN-10`)

Toolbar Font (`font.toolbar = dialog-PLAIN-10`)
This font is used on the toolbar buttons, if the button names are configured to be displayed.

Color Options

Colors are defined similarly to HTML colors: as six hexadecimal digits preceded by a pound sign. The first two digits specify a red value, the next two specify a green value, and the next two specify a blue value. For example, `#00FF00` would be a bright green. (In most cases, using the color chooser in the Preferences window is the simplest approach.)

Syntax Colors for Definitions

Normal Color (`definitions.normal.color = #000000`)

Normal Color (`definitions.normal.color = #000000`)
Used as the default color for program text.

Keyword Color (`definitions.keyword.color = #0000FF`)

Keyword Color (`definitions.keyword.color = #0000FF`)

Used as the color for known keywords (eg. "public", "for").

Type Color (`definitions.type.color = #00007C`)

Type Color (`definitions.type.color = #00007C`)
Used for known primitive types (eg. "int") and capitalized words, which usually correspond to class names.

Comment Color (`definitions.comment.color = #007C00`)

Comment Color (`definitions.comment.color = #007C00`)
Used as the color for all comments.

Double-quoted Color (`definitions.double.quoted.color = #B20000`)

Double-quoted Color (`definitions.double.quoted.color = #B20000`)
Used as the color for strings, which use double quotation marks.

Single-quoted Color (`definitions.single.quoted.color = #FF00FF`)

Single-quoted Color (`definitions.single.quoted.color = #FF00FF`)
Used as the color for characters, which use single quotation marks.

Number Color (`definitions.number.color = #00B2B2`)

Number Color (`definitions.number.color = #00B2B2`)
Used as the color for all numbers.

Other Colors

Background Color (`definitions.background.color = #FFFFFF`)

Background Color (`definitions.background.color = #FFFFFF`)
Used as the background color for all panes.

Brace-matching Color (`definitions.match.color = #BEFFE6`)

Brace-matching Color (`definitions.match.color = #BEFFE6`)
Used as the highlight color when matching braces.

Compiler Error Color (`compiler.error.color = #FFFF00`)

Compiler Error Color (`compiler.error.color = #FFFF00`)
Used as the highlight color for compiler errors and JUnit test failures.

Bookmark Color (`bookmark.color = #00FF00`)

Bookmark Color (`bookmark.color = #00FF00`)
Used as the highlight color for bookmarks.

Find Results Color 1/2/3/4 (`find.results.color1 = #FF9933`, `find.results.color2 = #30C996`, `find.results.color3 = #30FCFC`, `find.results.color4 = #FF66CC`)

Find Results Color 1/2/3/4 (`find.results.color1 = #FF9933`, `find.results.color2 = #30C996`, `find.results.color3 = #30FCFC`, `find.results.color4 = #FF66CC`) Used as the highlight color for find results.

Debugger Breakpoint Color (`debug.breakpoint.color = #FF0000`)

Debugger Breakpoint Color (`debug.breakpoint.color = #FF0000`) Used as the highlight color for breakpoints.

Disabled Debugger Breakpoint Color (`debug.breakpoint.disabled.color = #800000`)

Disabled Debugger Breakpoint Color (`debug.breakpoint.disabled.color = #800000`) Used as the highlight color for disabled breakpoints.

Debugger Location Color (`debug.thread.color = #64FFFF`)

Debugger Location Color (`debug.thread.color = #64FFFF`) Used as the highlight color for the location of the current thread in the debugger, shown after a breakpoint is hit or a step has occurred.

System.out Color (`system.out.color = #007C00`)

System.out Color (`system.out.color = #007C00`) Used as the color for text from System.out.

System.err Color (`system.err.color = #FF0000`)

System.err Color (`system.err.color = #FF0000`) Used as the color for text from System.err.

System.in Color (`system.in.color = #7C007C`)

System.in Color (`system.in.color = #7C007C`) Used as the color for text to be read by System.in.

Interactions Error Color (`interactions.error.color = #B20000`)

Interactions Error Color (`interactions.error.color = #B20000`) Used as the color for text that indicates errors in the Interactions Pane.

Debug Message Color (`debug.message.color = #0000B2`)

Debug Message Color (`debug.message.color = #0000B2`) Used as the color for text displayed by the debugger.

DrJava Errors Button Background Color (`rjava.errors.button.color = #FF0000`)

DrJava Errors Button Background Color (`rjava.errors.button.color = #FF0000`) Used as background color for the "DrJava Errors" button that is displayed in case of an internal DrJava error.

Right Margin Color (`right.margin.color = #CCCCCC`)

Right Margin Color (`right.margin.color = #CCCCCC`) Used as line color for the vertical line representing the right margin of the document, if the line is enabled.

Window Positions

DrJava can save the last position of the main window and its dialogs and restore them the next time DrJava is started.

Save Main Window Postion (`window.store.position = true`)

Save Main Window Postion (`window.store.position = true`) Whether to save the position and size of the DrJava window between sessions.

Save "xxx" Dialog Postion (several choices for "xxx": `dialog.clipboardhistory.store.position = true`, `dialog.gotofile.store.position = true`, `dialog.openjavadoc.store.position = true`, `dialog.completeword.store.position = true`, `dialog.jaroptions.store.position = true`)

Save "xxx" Dialog Postion (several choices for "xxx": `dialog.clipboardhistory.store.position = true`, `dialog.gotofile.store.position = true`, `dialog.openjavadoc.store.position = true`, `dialog.completeword.store.position = true`, `dialog.jaroptions.store.position = true`) Whether to save the position and size of the indicated dialog between DrJava sessions.

Reset "xxx" Dialog Postion and Size (several choices for "xxx")

Reset "xxx" Dialog Postion and Size (several choices for "xxx") By pressing this button, you can reset the position and size of the indicated dialog to its default value. This is useful if the dialog somehow appeared outside the screen and is not accessible, e.g. when switching from a dual-screen computer to a single-screen computer.

Detach Tabbed Panes (`tabbedpanes.detach = false`)

Detach Tabbed Panes (`tabbedpanes.detach = false`) By default, the tabbed panes are attached to the bottom of DrJava's main window. By selecting this option, DrJava displays the tabbed panes in their own separate window.

Detach Debugger (`debugger.detach = false`)

Detach Debugger (`debugger.detach = false`) By default, the debugger pane is displayed in DrJava's main window. By selecting this option, DrJava displays the debugger in its own separate window.

Key Bindings

Most menu items in DrJava have configurable keyboard shortcuts, along with several other navigational commands (such as moving to the beginning or end of a line). All such options are displayed on the Key Bindings panel in the Preferences window, along with their current value. Clicking on the value displays a window which allows the user to type a new key, showing any conflict with an existing key if there is one. Users may bind multiple keys to a single action as long as there are no conflicts and may add or remove key bindings as desired. (We recommend editing these options in the Preferences window.)

Compiler Options

Configurable options relating to compiling source code in DrJava. Note that Compiler Warnings are not shown when compiling any Java Language Level files. Also note that the Compiler Warnings options are all passed using the "-Xlint:" flag, which is a non-standard option and may not work with all implementations of the JDK.

Unchecked Warnings
(`show.unchecked.warnings = true`)

Unchecked Warnings (`show.unchecked.warnings = true`)
Passes the "-Xlint:unchecked" warning to javac. This will give more detail for unchecked conversion warnings that are mandated by the Java Language Specification.

Deprecation Warnings
(`show.deprecation.warnings = true`)

Deprecation Warnings (`show.deprecation.warnings = true`)
Passes the "-Xlint:deprecation" warning to javac. According to the JLS, this will show a description of each use or override of a deprecated member or class.

Path Warnings (`show.path.warnings = false`)

Path Warnings (`show.path.warnings = false`)
Passes the "-Xlint:path" warning to javac. According to the JLS, this will warn about non-existent path (classpath, sourcepath, etc.) directories.

Serial Warnings (`show.serial.warnings = false`)

Serial Warnings (`show.serial.warnings = false`)
Passes the "-Xlint:serial" warning to javac. According to the JLS, this will warn about missing serialVersionUID definitions on serializable classes.

Finally Warnings (`show.finally.warnings = false`)

Finally Warnings (`show.finally.warnings = false`)
Passes the "-Xlint:finally" warning to javac. According to the JLS, this will warn about finally clauses that cannot complete normally.

Fall-through Warnings
(show.fallthrough.warnings = false)

Fall-through Warnings
(show.fallthrough.warnings = false)
Passes the "-Xlint:fallthrough" warning to javac. According to the JLS, this will check switch blocks for fall-through cases and provide a warning message for any that are found. Fall-through cases are cases in a switch block, other than the last case in the block, whose code does not include a break statement, allowing code execution to "fall through" from that case to the next case.

Compiler Preference
(default.compiler.preference = No Preference)

Compiler Preference
(default.compiler.preference = No Preference)
Controls which compiler DrJava should choose when it starts. If no preference is set, DrJava will choose the newest compiler it can find.

Interactions Pane

Configurable options relating to interpreting code in the Interactions Pane.

Size of Interactions History (history.max.size = 500)

Size of Interactions History
(history.max.size = 500)
Specifies how many commands will be remembered in the history of the Interactions window. Previous commands can be recalled using the up and down arrow keys.

Enable the "Auto Import" Dialog
(dialog.autoimport.enabled = true)

Enable the "Auto Import" Dialog
(dialog.autoimport.enabled = true)
When this option is enabled, DrJava will display a dialog to automatically import classes when a class name is interpreted but not known. After the class has been selected, DrJava will execute the appropriate "import" statement and re-execute the line that caused the dialog to appear.

Classes to Auto-Import
(interactions.auto.import.classes = [])

Classes to Auto-Import
(interactions.auto.import.classes = [])
This option allows you to select classes and packages that should be imported automatically whenever the Interactions Pane is reset. List fully-qualified class names (e.g. java.util.ArrayList, or packages ending with a *, e.g. java.util.*).

Restore last working directory of the Interactions Pane on start up (`sticky.interactions.dir = true`)	Restore last working directory of the Interactions Pane on start up (`sticky.interactions.dir = true`) If this option is enabled, DrJava will restore the directory that was last used in the Interactions Pane. If it is disabled, DrJava will always use the value of the "user.home" property.
Smart Run Command (`smart.run.for.applets.and.programs = true`)	Smart Run Command (`smart.run.for.applets.and.programs = true`) If this option is enabled, DrJava will run applets and ACM Java Task Force Programs using the "Run" and "Run Project" buttons as well. These applets and ACM Java Task Force Programs do not need to have a main method to be run, as long as they are proper applets or ACM Java Task Force Programs.
Enforce access control (`dynamicjava.access.control = disabled`)	Enforce access control (`dynamicjava.access.control = disabled`) This option controls the access control DrJava performs when class members are accessed. If the option is set to 'private and package only', then access control is used for all class members that are private or package private. If it is set to 'private only', then access control is used only for private members, and the other access levels can always be accessed. If it is set to 'disabled', all class members can be accessed, regardless of their access level. (Note: Currently, access control in DrJava's Interactions Pane has not been fully implemented; at most, access is checked for private and package private members; protected members can always be accessed.)
Require semicolon (`dynamicjava.require.semicolon = false`)	Require semicolon (`dynamicjava.require.semicolon = false`) If this option is enabled, then DrJava will require a semicolon at the end of statements in the Interactions Pane.
Require variable type (`dynamicjava.require.variable.type = true`)	Require variable type (`dynamicjava.require.variable.type = true`) If this option is enabled, then DrJava will a variable type for variable declarations in the Interactions Pane (e.g. `int i = 5`). If it is disabled, DrJava will attempt to assign a

variable type automatically (e.g. `i = 5` to declare an `int i`).

Debugger

All configurable options relating to the debugger.

Sourcepath (`debug.sourcepath = ""`)

Sourcepath (`debug.sourcepath = ""`)
A list of directories on which to search for source files when stepping through code. The debugger will attempt to open files from these directories automatically when stepping.

Step Into Java Classes (`debug.step.java = false`)

Step Into Java Classes (`debug.step.java = false`)
Whether to step into Java source files when stepping through a suspended method call. It is recommended to put the Java source (usually distributed with the JDK) on the Sourcepath if this option is selected.

Step into Interpreter Classes (`debug.step.djava = false`)

Step into Interpreter Classes (`debug.step.djava = false`)
Whether to step into DynamicJava source files when stepping through a suspended method call. DynamicJava is the Java interpreter used in the Interactions pane, and the source can be obtained from http://koala.ilog.fr/djava. Useful primarily when debugging DrJava itself.

Step into DrJava Classes (`debug.step.drjava = false`)

Step into DrJava Classes (`debug.step.drjava = false`)
Whether to step into DrJava source files when stepping through a suspended method call. Useful primarily when debugging DrJava itself.

Classes/Packages To Exclude (`debug.step.exclude = ""`)

Classes/Packages To Exclude (`debug.step.exclude = ""`)
Allows you to specify other classes/packages to step over. This should be a list with fully qualified names. To exclude a whole package, add `packagename.*` to the list. You might use this box to exclude instructor provided libraries, for example `java.util.*`.

Auto-Import after Breakpoint/Step (`debug.auto.import = true`)

Auto-Import after Breakpoint/Step (`debug.auto.import = true`)
Automatically imports all classes and packages again that had been imported when the program was last suspended, i.e. before the breakpoint was hit or before the last step was taken.

Auto-Step Rate in ms (`auto.step.rate = 1000`)

Auto-Step Rate in ms (`auto.step.rate = 1000`)
The delay interval at which Automatic Trace steps into every single line of code of the program.

Javadoc

All configurable options relating to generating Javadoc.

Access Level (`javadoc.access.level = "package"`)

Access Level (`javadoc.access.level = "package"`)
Specifies the lowest access level for fields and methods to include in the generated documentation. Legal values are "public", "protected", "package", and "private".

Java Version for Javadoc Links (`javadoc.link.version = (JDK Version)`)

Java Version for Javadoc Links (`javadoc.link.version = (JDK Version)`)
Specifies which URL to use when generating links to Java library classes. Legal values are "1.3", "1.4", and "none" if no links to Java library classes are desired. (This option defaults to the version of the user's JDK.)

This setting also controls which of the URLs below is used for the "Open Java API Javadoc" feature.

Javadoc 1.3 URL (`javadoc.1.3.link = "http://java.sun.com/j2se/1.3/docs/api"`)

Javadoc 1.3 URL (`javadoc.1.3.link = "http://java.sun.com/j2se/1.3/docs/api"`)
The URL to use when generating links to JDK 1.3 library classes or opening the Javadoc pages for the Java API.

Javadoc 1.4 URL (`javadoc.1.4.link = "http://java.sun.com/j2se/1.4/docs/api"`)

Javadoc 1.4 URL (`javadoc.1.4.link = "http://java.sun.com/j2se/1.4/docs/api"`)
The URL to use when generating links to JDK 1.4 library classes or opening the Javadoc pages for the Java API.

Javadoc 1.5 URL (`javadoc.1.5.link = "http:/java.sun.com/j2se/1.5/docs/api"`)

Javadoc 1.5 URL (`javadoc.1.5.link = "http:/java.sun.com/j2se/1.5/docs/api"`)
The URL to use when generating links to JDK 1.5 library classes or opening the Javadoc pages for the Java API.

Javadoc 1.6 URL (`javadoc.1.6.link = "http://java.sun.com/javase/6/docs/api"`)

Javadoc 1.6 URL (`javadoc.1.6.link = "http://java.sun.com/javase/6/docs/api"`)

The URL to use when generating links to JDK 1.6 library classes or opening the Javadoc pages for the Java API.

JUnit 3.8.2 URL (`junit.3.8.2.link = "http://www.cs.rice.edu/~javaplt/javadoc/junit3.8.2"`)

JUnit 3.8.2 URL (`junit.3.8.2.link = "http://www.cs.rice.edu/~javaplt/javadoc/junit3.8.2"`)
The URL to use when generating links to JUnit 3.8.2 library classes or opening the Javadoc pages for the Java API.

Additional Javadoc URLs (`javadoc.additional.link = []`)

Additional Javadoc URLs (`javadoc.additional.link = []`)
A list of URLs used to open Javadoc pages for user-specified libraries. Please enter the URL to the directory that contains the `allclasses-frame.html` file. For example, to open DrJava's Javadoc, enter `http://drjava.org/javadoc/drjava`.

Default Destination Directory (`javadoc.destination = ""`)

Default Destination Directory (`javadoc.destination = ""`)
If a directory is specified, it will be used as the default when generating new documentation.

Custom Javadoc Parameters (`javadoc.custom.params = ""`)

Custom Javadoc Parameters (`javadoc.custom.params = ""`)
Any custom parameters to pass to the Javadoc tool, separated by spaces. Use "javadoc -help" at a command line to view the available parameters.

Generate Javadoc From Source Roots (`javadoc.from.roots = false`)

Generate Javadoc From Source Roots (`javadoc.from.roots = false`)
If this option is enabled, then Javadoc will not only search the current package and all subpackages for files, it will also search all "enclosing" packages (those at a higher level).

Notifications

Configures how often DrJava notifies you for certain events. The notifications in this section can all be suppressed by clicking on a "Do not show this message again" checkbox (or similar) on the notification itself.

Prompt Before Quit (`quit.prompt = true`)

Prompt Before Quit (`quit.prompt = true`)
Whether to display a confirmation message before DrJava quits.

Prompt Before Resetting Interactions Pane (`interactions.reset.prompt = true`)

Prompt Before Resetting Interactions Pane (`interactions.reset.prompt = true`)

Whether to display a confirmation message before resetting the Interactions Pane.

Prompt if Interactions Pane Exits Unexpectedly (`interactions.exit.prompt = true`)	Prompt if Interactions Pane Exits Unexpectedly (`interactions.exit.prompt = true`) Whether to display a message if the Interactions Pane is exited without the Reset button being clicked.
Prompt for Javadoc Destination (`javadoc.prompt.for.destination = true`)	Prompt for Javadoc Destination (`javadoc.prompt.for.destination = true`) Whether to always display the destination selection dialog when starting Javadoc.
Prompt Before Cleaning Build Directory (`prompt.before.clean = true`)	Prompt Before Cleaning Build Directory (`prompt.before.clean = true`) Whether to display a confirmation message before cleaning the build directory of a project.
Automatically Save Before Compiling (`save.before.compile = false`)	Automatically Save Before Compiling (`save.before.compile = false`) Whether to automatically save all files each time a Compile command is chosen.
Automatically Compile Before Testing (`compile.before.junit = false`)	Automatically Compile Before Testing (`compile.before.junit = false`) Whether to automatically compile before running JUnit tests.
Automatically Save Before Generating Javadoc (`save.before.javadoc = false`)	Automatically Save Before Generating Javadoc (`save.before.javadoc = false`) Whether to automatically save all files each time a Javadoc command is chosen.
Warn on Breakpoint Out of Sync (`warn.breakpoint.out.of.sync = true`)	Warn on Breakpoint Out of Sync (`warn.breakpoint.out.of.sync = true`) Whether to warn if setting a breakpoint in a source file that is not in sync with its class file.
Warn if Debugging Modified File (`warn.debug.modified.file = true`)	Warn if Debugging Modified File (`warn.debug.modified.file = true`) Whether to warn if using the debugger on a file which has been modified since its last save.
Warn to Restart to Change Look and Feel (`warn.change.laf = true`)	Warn to Restart to Change Look and Feel (`warn.change.laf = true`) Whether to warn that changes to the Look and Feel do not take effect until after a restart.

Warn if File's Path Contains a '#' Symbol
(`warn.path.contains.pound = true`)

Warn if File's Path Contains a '#' Symbol
(`warn.path.contains.pound = true`)
Whether DrJava should warn the user if the file being saved has a path that contains a '#' symbol. Users cannot use such files in the Interactions Pane because of a bug in Java.

Show a notification window when the first DrJava error occurs (`dialog.drjava.error.popup.enabled = true`)

Show a notification window when the first DrJava error occurs (`dialog.drjava.error.popup.enabled = true`)
Whether to show a popup dialog when the first internal DrJava error occurs. If this option is not selected, DrJava will try to continue quietly and only display the "DrJava Errors" button (See DrJava Errors).

Warn if Compiz Detected (`warn.if.compiz = true`)

Warn if Compiz Detected (`warn.if.compiz = true`)
Whether DrJava should warn the user if Compiz is detected (see Compiz Detected dialog). DrJava suffers from an incompatibility between the Linux window manager Compiz and Oracle's Swing Java GUI library. We, the developers of DrJava, cannot do anything to fix this problem. We hope that future versions of Java and Compiz will address the incompatibility. In the meantime, we recommend that you disable Compiz if you experience problems. We also suggest that you use the latest versions of Compiz and Java, so you can benefit from possible bug fixes made by Oracle and the Compiz developers. For more information, see http://drjava.org/compiz/.

Delete language level class files
(`delete.ll.class.files = always`)

Delete language level class files
(`delete.ll.class.files = always`)
Whether Drjava should delete class files generated from Language Level source files (.dj0, .dj1, and .dj2) before compilation. This may be necessary to avoid problems with the Language Level facility. Selecting "always" will do this automatically without involving the user. Selecting "ask me" will display a notification window and let the user decide. Selecting "never" will not delete class files nor ask the user to do so.

Check for new versions
(`new.version.notification = stable and beta versions only`)

Check for new versions
(`new.version.notification = stable and beta versions only`)

Whether Drjava should check if a newer version of DrJava exists. Setting this to "none (disabled)" will not check for newer versions. The other setting select what kind of versions will be considered, ranging from "stable versions only" to "weekly experimental builds".

Days between new version check
(`new.version.notification.days = 7`)

Days between new version check (`new.version.notification.days = 7`)
How many days have to pass before DrJava will look for a new version again. By default, DrJava will check every seven days..

Miscellaneous

These are the remaining configurable options in DrJava.

Indent Level (`indent.level = 2`)

Indent Level (`indent.level = 2`)
Sets how many spaces to use for each level of indenting. Note that tab characters are not allowed in DrJava.

Recent Files List Size (`recent.files.max.size = 5`)

Recent Files List Size (`recent.files.max.size = 5`)
Specifies how many recently used files to display in the File menu.

Size of Browser History
(`browser.history.max.size = 50`)

Size of Browser History (`browser.history.max.size = 50`)
Specifies how many source code locations are stored in the browser history.

Automatically Close Block Comments
(`auto.close.comments = false`)

Automatically Close Block Comments (`auto.close.comments = false`)
Whether to automatically insert the string designating the end of a multi-line comment after beginning one.

Allow Assert Keyword in Java 1.4
(`javac.allow.assert = false`)

Allow Assert Keyword in Java 1.4 (`javac.allow.assert = false`)
Whether to support the `assert` keyword when compiling with a JDK 1.4 or later compiler.

Keep Emacs-style Backup Files (`files.backup = true`)

Keep Emacs-style Backup Files (`files.backup = true`)
Whether DrJava should keep a backup copy of each file that the user modifies, saved with a "~" at the end of the filename.

Clear Console After Interactions Reset
(`reset.clear.console = true`)

Clear Console After Interactions Reset (`reset.clear.console = true`)

	Whether DrJava should clear the contents of the Console Tab each time the Interactions Pane is reset.
Require test classes in projects to end in "Test" (`force.test.suffix = false`)	Require test classes in projects to end in "Test" (`force.test.suffix = false`) Whether to require that JUnit test classes in projects end in "Test". If this is enabled, classes that do not end in "Test" will not be considered JUnit tests when in project mode.
Put the focus in the definitions pane after find/replace (`find.replace.focus.in.defpane = false`)	Put the focus in the definitions pane after find/replace (`find.replace.focus.in.defpane = false`) Whether to put the focus in the Definitions pane after using find/replace. If this option is not enabled, the focus will remain in the Find/Replace pane.
Forcefully Quit DrJava (`drjava.use.force.quit = false`)	Forcefully Quit DrJava (`drjava.use.force.quit = false`) On some systems (namely tablet PCs), DrJava does not shut down properly when quit. Select this option to remedy this problem.
Enable Remote Control (`remote.control.enabled = true`)	Enable Remote Control (`remote.control.enabled = true`) Java's "remote control" allows other applications to control certain aspects of DrJava, for example what file is displayed. This feature is also necessary if you want to double-click on a .java file to open the file in an existing instance of DrJava.
Remote Control Port (`remote.control.port = 4444`)	Remote Control Port (`remote.control.port = 4444`) Selects the port that Drjava uses for its remote control.
Follow File Delay (`follow.file.delay = 300`)	Follow File Delay (`follow.file.delay = 300`) Specifies the number of milliseconds that have to pass before DrJava will update a "Follow File" window again.
Maximum Lines in "Follow File" Window (`follow.file.lines = 1000`)	Maximum Lines in "Follow File" Window (`follow.file.lines = 1000`) Specifies the number of of lines that a "Follow File" window may contain. If a file has more than this many lines, only the end of the file will be displayed.

File Types

Configurable options for file types. Note that the options here are only available on Windows, and only if the .exe file is used.

.drjava files are DrJava project files. .djapp files are DrJava add-ons. .java files are Java source files.

Associate .drjava and .djapp Files with DrJava

Associate .drjava and .djapp Files with DrJava
Set the file type associations so that double-clicking on .drjava and .djapp files in the Windows Explorer will open them in DrJava.

Remove .drjava and .djapp File Associations

Remove .drjava and .djapp File Associations
Remove the association of .drjava and .djapp files with DrJava.

Associate .java Files with DrJava

Associate .java Files with DrJava
Set the file type associations so that double-clicking on .java files in the Windows Explorer will open them in DrJava.

Remove .java File Associations

Remove .java File Associations
Remove the association of .java files with DrJava.

Automatically assign .java, .drjava and .djapp Files to DrJava(`file.ext.registration = ask me at startup`)

Automatically assign .java, .drjava and .djapp Files to DrJava (`file.ext.registration = ask me at startup`)
Specifies whether Drjava should automatically associate .java, .drjava and .djapp files with DrJava so those files are opened with DrJava when double-clicked. When set to the default, "ask me at startup", DrJava will check at startup if those associations exist, and if not, present a dialog allowing the user to set the file associations (see Set File Associations dialog). Setting this option to "always" will set the file associations automatically at startup without user interaction. Setting it to "never" will not change file associations nor ask the user.

JVMs

Configurable options for the two Java Virtual Machines (JVMs) that DrJava uses.

Maximum Heap Size for Main JVM in MB (`master.jvm.xmx = default`)

Maximum Heap Size for Main JVM in MB (`master.jvm.xmx = default`)
Specifies how many megabytes of memory Java should use for the Main JVM (the main part of DrJava including the editor and compiler). The "default" setting leaves this up

to Java. If you experience "Out of memory" errors, set this to a value that is larger than 64 MB (default in Java 5) but smaller than the amount of physical memory you have. If it is still too small, choose the next bigger setting. Note: You have to restart DrJava for changes to become effective.

JVM Args for Main JVM (`master.jvm.args = ""`)

Specifies the JVM arguments that should be used for DrJava's Main JVM, other than the maximum heap size (-Xmx), which is controlled using the option above. Note: You have to restart DrJava for changes to become effective.

Maximum Heap Size for Interactions JVM in MB (`slave.jvm.xmx = default`)

Specifies how many megabytes of memory Java should use for the Interactions JVM (used to interpret code in the Interactions Pane and to run programs developed in DrJava). The "default" setting leaves this up to Java. If you experience "Out of memory" errors, set this to a value that is larger than 64 MB (default in Java 5) but smaller than the amount of physical memory you have. If it is still too small, choose the next bigger setting. Note: You have to reset the Interactions Pane for changes to become effective.

JVM Args for Interactions JVM (`slave.jvm.args = ""`)

Specifies the JVM arguments that should be used for DrJava's Interactions JVM, other than the maximum heap size (-Xmx), which is controlled using the option above. Note: You have to reset the Interactions Pane for changes to become effective.

Appendix B. DrJava Errors

Even though we try hard, it is an unfortunate reality that DrJava contains errors. This appendix describes how DrJava will most likely react in case of an error, and how you can help improve DrJava if you encounter an error.

How DrJava Reacts. When an internal DrJava error occurs, DrJava puts information about the error in a list and displays a "DrJava Errors" toolbar button that is normally hidden. You can view this list by clicking on that button or by selecting the "DrJava Errors" command in the Help menu.

In the "DrJava Errors" dialog, you can use the "Previous" and "Next" buttons to browse the list of errors, or press the "Copy This Error" button to copy the information about the currently displayed error to the clipboard. This information is useful to us in finding the cause of the error. The "Dismiss" button clears the list of errors, hides the "DrJava Errors" button, and closes the window. The "Close" button leaves the list as it is and only closes the window.

Optionally, DrJava may also display a popup notification window when the first error occurs. You may disable this popup under "Notifications" in the Preferences window (see Configuring DrJava). In most cases, an internal DrJava error will not be fatal, and most likely you will be able to continue working even without restarting DrJava. Therefore, DrJava does not display this popup for every error, but only when the list of errors is empty, and then an error occurs. This is to inform you that DrJava might be unstable now.

The popup window contains a "Close" button to close the window and continue working, and a "More Information" button that will open up the "DrJava Errors" dialog. It also provides a checkbox labeled "Keep showing this notification" that allows you to disable the popup; disabling this checkbox does the same as disabling the "Show a notification window when the first DrJava error occurs" option in the Preferences window (see Configuring DrJava).

Submitting a Bug Report. If you encounter an error, we would be thrilled if you helped us improve DrJava by submitting a bug report on our SourceForge site at http://sourceforge.net/projects/drjava/

To submit a bug report, please follow these instructions:

1. Point your favorite browser to the URL above.

2. Click on the link labeled "Bugs" (it's in the long row of links: "Summary | Admin | Home Page | Tracker | Bugs | ...").

3. Optional: If you have the time, please browse through the list of bugs to see if the error you encountered has already been reported.

4. Click on the link labeled "Submit New" (it's on the left, below the long row of links).

5. Optional: If you have a SourceForge account, please log in before submitting the bug report. This is not necessary, but it enables us to communicate with you if we need more information.

6. Please enter a short description in the "Summary" field, and a more detailed description that includes the information from the "DrJava Errors" window in the "Detailed Description" field. Try to be as specific as you can. Here's a list of questions you might want to consider in your description:

   • What operating system are you using?

   • What version of the Java JDK are you using (1.5, 6, etc.)? Do you have multiple versions of the JDK (or maybe the JRE) installed?

   • Did you have a project open, or where you using just a bunch of files not part of a project?

- What were you doing just before the error occurred? Were you compiling, saving a file, debugging, using find/replace, etc.?

- What happened after the error occurred? Could you continue with your work, or was it so bad that you had to quit DrJava?

7. Press the "Submit" button on the bottom of the page

That's it! Thank you for helping us make DrJava better!

Appendix C. Indenting Files from the Command Line

DrJava has a very useful indenting algorithm, but indenting several large files can be a time consuming process. Because of this, DrJava provides a command line interface to its indenter that can be run on a series of files.

Running the Command Line Indenter

Use the following command at a command prompt to run the indenter on a series of files.
`java -classpath drjava-DATE-TIME.jar edu.rice.cs.drjava.IndentFiles [-indent N] [filename.java...]`

Replace DATE-TIME with the appropriate value for your DrJava file. The "-indent" argument is optional, where N is the number of spaces to use for an indentation level.