jEdit 3.2 User’s Guide

jEdit 3.2 User’s Guide

Table of Contents

I. Using jEdit 11
- 1. Starting jEdit 12
  - 1.1. Conventions 12
  - 1.2. Platform-Independent Instructions 12
  - 1.3. Starting jEdit on Windows 13
  - 1.4. Command Line Usage 14
- 2. jEdit Basics 17
  - 2.1. Buffers 17
  - 2.2. Views 17
    - 2.2.1. Window Docking 18
    - 2.2.2. The Status Bar 18
  - 2.3. The Text Area 19
  - 2.4. Command Repetition 20
- 3. Working With Files 21
  - 3.1. Creating New Files 21
  - 3.2. Opening Files 21
  - 3.3. Saving Files 21
    - 3.3.1. Autosave and Crash Recovery 22
    - 3.3.2. Backups 22
  - 3.4. Line Separators 23
  - 3.5. Character Encodings 23
    - 3.5.1. Commonly Used Encodings 24
  - 3.6. The File System Browser 25
  - 3.7. Reloading Files 28
  - 3.8. Multi-Threaded I/O 28
  - 3.9. Printing Files 29
  - 3.10. Closing Files and Exiting jEdit 29
- 4. Editing Text 31
  - 4.1. Moving The Caret 31
  - 4.2. Selecting Text 31
    - 4.2.1. Rectangular Selection 31

4.2.2. Multiple Selection....32
4.3. Inserting and Deleting Text....33
4.4. Undo and Redo....34
4.5. Working With Words....35
4.6. Working With Lines....35
4.7. Working With Paragraphs....36
4.8. Scrolling....37
4.9. Transferring Text....37
4.9.1. Quick Copy....38
4.9.2. The System Clipboard....38
4.9.3. General Register Commands....39
4.10. Markers....40
4.11. Search and Replace....41
4.11.1. Searching For Text....41
4.11.2. Replacing Text....42
4.11.3. HyperSearch....43
4.11.4. Multiple File Search....44
4.11.5. The Search Bar....44
5. Editing Source Code....46
5.1. Edit Modes....46
5.1.1. Mode Selection....46
5.1.2. Syntax Highlighting....46
5.2. Abbreviations....47
5.2.1. Positional Parameters....47
5.3. Bracket Matching....48
5.4. Tabbing and Indentation....49
5.4.1. Soft Tabs....49
5.4.2. Automatic Indent....50
5.5. Commenting Out Code....51
5.6. Folding....52
5.6.1. Narrowing....53
6. Customizing jEdit....54
6.1. The Buffer Options Dialog Box....54
6.2. Buffer-Local Properties....54

6.3. The Global Options Dialog Box 56
6.4. The jEdit Settings Directory 59
7. Using Macros 61
7.1. Recording Macros 61
7.2. Running Macros 62
7.3. How jEdit Organizes Macros 62
8. Installing and Using Plugins 65
8.1. The Plugin Manager 65
8.2. Installing Plugins 65
8.3. Updating Plugins 66
A. Keyboard Shortcuts 68
B. The Activity Log 75
C. History Text Fields 76
D. Glob Patterns 77
E. Regular Expressions 79
F. Macros Included With jEdit 82
F.1. File Management Macros 82
F.2. Text Macros 82
F.3. Java Code Macros 84
F.4. Search Macros 86
F.4.1. The Find_Occurrence Macro Group 87
F.5. Macros for Listing Properties 88
F.6. Miscellaneous Macros 89
G. jEditLauncher for Windows 93
G.1. Introduction 93
G.2. Starting jEdit 93
G.3. The Context Menu Handler 94
G.4. Uninstalling jEdit and jEditLauncher 95
G.5. The jEditLauncher Interface 96
G.6. Scripting Examples 97
G.7. Legal Notice 99
II. Writing Edit Modes 101
9. Writing Edit Modes 102

9.1. An XML Primer....102
9.2. The Preamble and MODE tag....103
9.3. The PROPS Tag....103
9.4. The RULES Tag....105
9.4.1. The TERMINATE Rule....106
9.4.2. The WHITESPACE Rule....108
9.4.3. The SPAN Rule....108
9.4.4. The EOL_SPAN Rule....110
9.4.5. The MARK_PREVIOUS Rule....110
9.4.6. The MARK_FOLLOWING Rule....111
9.4.7. The SEQ Rule....111
9.4.8. The KEYWORDS Rule....112
9.4.9. Token Types....112
10. Installing Edit Modes....114
III. Writing Macros....116
11. Introducing BeanShell....117
11.1. Single Execution Macros....117
12. A Few Simple Macros....119
12.1. The Mandatory First Example....119
12.2. Helpful Methods in the Macros Class....121
12.3. Now For Something Useful....124
13. A Dialog-Based Macro....128
13.1. Use of the Macro....128
13.2. Listing of the Macro....128
13.3. Analysis of the Macro....131
13.3.1. Import Statements....132
13.3.2. Create the Dialog....132
13.3.3. Create the Text Fields....133
13.3.4. Create the Buttons....134
13.3.5. Register the Action Listeners....135
13.3.6. Make the Dialog Visible....136
13.3.7. The Action Listener....136
13.3.8. Get the User’s Input....137

13.3.9. Call jEdit Methods to Manipulate Text 137
13.3.10. The Main Routine 139
14. Macro Tips and Techniques 140
14.1. Getting Input for a Macro 140
14.1.1. Getting a Single Line of Text 140
14.1.2. Getting Multiple Data Items 141
14.1.3. Selecting Input From a List 144
14.1.4. Using a Single Keypress as Input 145
14.2. Startup Scripts 147
14.3. Running Scripts from the Command Line 148
14.4. Advanced BeanShell Techniques 150
14.4.1. BeanShell's Convenience Syntax 150
14.4.2. Special BeanShell Keywords 151
14.4.3. Implementing Interfaces 151
14.4.4. BeanShell Commands 152
14.5. Debugging Macros 152
14.5.1. Identifying Exceptions 152
14.5.2. Using the Activity Log as a Tracing Tool 153
IV. Writing Plugins 155
15. Introducing the Plugin API 156
16. jEdit as a Plugin Host 158
16.1. Loading Plugins 158
16.1.1. The JARClassLoader 158
16.1.2. Starting the Plugin 159
16.2. The User Interface of a Plugin 161
16.2.1. The Role of the View Object 161
16.2.2. The DockableWindowManager and the EditBus 161
16.2.3. Message Routing and Dockable Window Creation 162
17. The jEdit Plugin API 164
17.1. Plugin Core Classes 164
17.1.1. Class EditPlugin 164
17.1.2. Class EBPlugin 165
17.2. Interface DockableWindow 167

17.3. Plugin Option Pane Classes 168
17.3.1. Class AbstractOptionPane 168
17.3.2. Class OptionGroup..169
17.4. Other Plugin Resources..170
17.4.1. The Action Catalog..170
17.4.2. Plugin Properties..172
17.4.3. Plugin Documentation 174
18. Writing a Plugin..176
18.1. QuickNotepad: An Example Plugin..176
18.2. Writing a Plugin Core Class 177
18.2.1. Choosing a Base Class..177
18.2.2. Implementing Base Class Methods..178
18.2.2.1. General Considerations..178
18.2.2.2. Example Plugin Core Class 179
18.2.3. Resources for the Plugin Core Class..181
18.2.3.1. Actions..181
18.2.3.2. Action Labels and Menu Items..183
18.3. Implementing a Dockable Window Class..184
18.3.1. Using a Single Window Class..184
18.3.2. An Action Interface..185
18.3.3. A Lightweight Dockable Window Class..186
18.4. The Plugin's Visible Window 187
18.4.1. Class QuickNotepad..187
18.4.2. Class QuickNotepadToolBar 191
18.5. Designing an Option Pane 192
18.6. Creating Other Plugin Resources..195
18.7. Compiling the Plugin..197
V. jEdit API Reference..200
19. BeanShell Commands..201
19.1. Output Commands 201
19.2. File Management Commands 201
19.3. Component Commands..202
19.4. Resource Management Commands..203

19.5. Script Execution Commands....203
19.6. BeanShell Object Management Commands....204
19.7. Other Commands....205
20. General jEdit Classes....207
20.1. Class jEdit....207
20.2. Class View....210
20.3. Class Registers....211
20.4. Interface Registers.Register....212
20.5. Class DockableWindowManager....213
20.6. Class JEditTextArea....213
20.6.1. Class Selection....214
20.6.2. Selection methods in JEditTextArea....215
20.6.2.1. Adding and removing selections....215
20.6.2.2. Getting and setting selected text....217
20.6.2.3. Other selection methods....218
20.6.3. Other methods in JEditTextArea....218
20.6.3.1. Editing caret methods....218
20.6.3.2. Methods for scrolling the text area....220
20.6.3.3. Methods for calculating editing positions....221
20.6.3.4. Other methods for retrieving text....221
20.6.3.5. Methods for deleting text....221
20.6.3.6. Methods for modifying text....222
20.6.3.7. Methods for creating comments....223
20.6.3.8. Methods for getting buffer statistics....223
20.7. Class Buffer....224
20.7.1. File attribute methods....224
20.7.2. Editing attribute methods....224
20.7.3. Editing action methods....225
20.7.3.1. General editing methods....226
20.7.3.2. Marker methods....227
20.7.3.3. Folding methods....228
20.7.3.4. Virtual and physical line indices....229
20.8. Class Macros....229
20.9. Class SearchAndReplace....230

20.10. Class GUIUtilities..233
20.11. Class TextUtilities..234
20.12. Class MiscUtilities...235
20.13. Class BeanShell 236
21. EditBus Classes..238
21.1. Class EditBus...238
21.2. Interface EBComponent...238
21.3. Class EBMessage..239
21.4. Class BufferUpdate..239
21.5. Class CreateDockableWindow...240
21.6. Class EditorExiting...241
21.7. Class EditorExitRequested..241
21.8. Class EditorStarted...241
21.9. Class EditPaneUpdate...241
21.10. Class MacrosChanged..242
21.11. Class PropertiesChanged 242
21.12. Class SearchSettingsChanged...242
21.13. Class VFSUpdate..242
21.14. Class ViewUpdate...243

I. Using jEdit

This part of the user's guide covers jEdit's text editing commands, along with basic usage of macros and plugins.

This part of the user's guide was written by Slava Pestov <`slava@jedit.org`>.

Chapter 1. Starting jEdit

1.1. Conventions

Several conventions are used throughout the manual. They will be described here.

When a menu item selection is being described, the top level menu is listed first, followed by successive levels of submenus, finally followed by the menu item itself. All menu components are separated by greater-than symbols (“>”). For example, View>Scrolling>Scroll to Current Line refers to the Scroll to Current Line command contained in the Scrolling submenu of the View menu.

As with many other applications, menu items that end with ellipsis (...) display dialog boxes or windows when invoked.

Many jEdit commands can be also be invoked using keystrokes. This speeds up editing by letting you keep your hands on the keyboard. Not all commands with keyboard shortcuts are accessible with one key stroke; for example, the keyboard shortcut for Scroll to Current Line is **Control-E Control-J**. That is, you must first press **Control-E**, followed by **Control-J**.

MacOS

When running on MacOS, the primary modifier key is **Command**, not **Control**. If you are a Mac user, mentally substitute **Command** whenever you see **Control** in this guide.

1.2. Platform-Independent Instructions

Exactly how jEdit is started depends on the operating system; on Unix systems, usually you would run the “jedit” command at the command line, or select jEdit from a menu; on Windows, you might use the jEditLauncher package, which is documented in Section 1.3.

If jEdit is started while another copy is already running, control is transferred to the running copy, and a second instance is not loaded. This saves time and memory if jEdit is started multiple times. Communication between instances of jEdit is implemented using TCP/IP sockets; the initial instance is known as the *server*, and subsequent invocations are *clients*.

If the **-background** command line switch is specified, jEdit will continue running and waiting for client requests even after all editor windows are closed. The advantage of background mode is that you can open and close jEdit any number of times, only having to wait for it to start the first time. The downside of background mode is that jEdit will continue to consume memory when no windows are open.

For more information about command line switches that control the server feature, see Section 1.4.

Unlike other applications, jEdit automatically loads any files that were open last time in was used, so you can get back to work immediately, without having to find the files you are working on first. This feature can be disabled in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.

The edit server and security

Not only does the server pick a random TCP port number on startup, it also requires that clients provide an *authorization key*; a randomly-generated number only accessable to processes running on the local machine. So not only will “bad guys” have to guess a 64-bit integer, they will need to get it right on the first try; the edit server shuts itself off upon receiving an invalid packet.

In environments that demand absolute security, the edit server can be disabled by specifying the **-noserver** command line switch.

1.3. Starting jEdit on Windows

On Windows, jEdit comes with *jEditLauncher* - an optional package of components that make it easy to start jEdit, manage its command line settings, and launch files and macro scripts.

The jEditLauncher package provides three shortcuts for running jEdit: one in the desktop's Start menu, a entry in the Programs menu, and a second shortcut on your desktop. Any of these may be deleted or moved without affecting jEdit's operation. To launch jEdit, simply select one of these shortcuts as you would for any Windows application.

The jEditLauncher package includes a utility for changing the command line parameters that are stored with jEditLauncher and used everytime it runs jEdit. You can change the Java interpreter used to launch jEdit, the amount of heap memory, the working directory and other command line parameters. To make these changes, select Set jEdit Parameters from the jEdit group in the Programs menu, or run **`jedit /p`** from a command line that has jEdit's installation directory in its search path. A dialog will appear that allows you to change and save a new set of command line parameters.

The package also add menu items to the context or "right-click" menu displayed by the Windows shell when you click on a file item in the desktop window, a Windows Explorer window or a standard file selection dialog. The menu entries allow you to open selected files in jEdit, starting the aplication if necessary. It will also allow you to open all files in a directory with a given extension with a single menu selection. If a BeanShell macro script with a `.bsh` extension is selected, the menu includes the option of running that script within jEdit. If you have the JDiff plugin installed with jEdit, you can also select two files and have jEdit compare them in a side-by-side graphical display.

For a more detailed description of all features found in the jEditLauncher package, see Appendix G.

1.4. Command Line Usage

On operating systems that support a command line, jEdit can be passed various arguments to control its behavior.

When opening files from the command line, a line number or marker to position the caret on can be specified like so:

```
$ jedit MyApplet.java +line:10
$ jedit thesis.tex +marker:c
```

A number of options can also be specified to control several obscure features. They are listed in the following table.

If you are using jEditLauncher to start jEdit on Windows, only file names and marker and line number specifications can be specified on the command line; other parameters must be set as described in Section G.2.

Option	**Description**
-background	Runs jEdit in background mode. In background mode, the edit server will continue listening for client connections even after all views are closed. See Chapter 1.
-norestore	jEdit will not attempt to restore previously open files on startup. This feature can also be set permanently in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.
-run=*`script`*	Runs the specified BeanShell script. There can only be one of these parameters on the command line. See Section 14.3 for details.
-server	Stores the server port info in the file named `server inside the settings directory.`
-server=*`name`*	Stores the server port info in the file named *`name. File names for this parameter are relative to the settings directory.`*
-noserver	Does not attempt to connect to a running edit server, and does not start one either. For information about the edit server, see Chapter 1.
-settings=*`dir`*	Loads and saves the user-specific settings from the directory named *`dir, instead of the default user.home/.jedit. dir will be created if it does not exist. Has no effect when connecting to another instance via the edit server.`*

Option	**Description**
-nosettings	Starts jEdit without loading user-specific settings. See Section 6.4.
-nostartupscripts	Causes jEdit to not run any startup scripts. See Section 14.2. Has no effect when connecting to another instance via the edit server.
-usage	Shows a brief command line usage message without starting jEdit. This message is also shown if an invalid switch was specified.
-version	Shows the version number without starting jEdit.
- -	Specifies the end of the command line switches. Further parameters are treated as file names, even if they begin with a dash. Can be used to open files whose names start with a dash, and so on.

Chapter 2. jEdit Basics

2.1. Buffers

Several files can be opened and edited at once. Each open file is referred to as a *buffer*. The combo box above the text area selects the buffer to edit. Different emblems are displayed next to buffer names in the list, depending the buffer's state; a red disk is shown for buffers with unsaved changes, a lock is shown for read-only buffers, and a spark is shown for new buffers which don't yet exist on disk.

In addition to the buffer combo box, various commands can also be used to select the buffer to edit.

View>Go to Previous Buffer (keyboard shortcut: **Control-Page Up**) switches to the previous buffer in the list.

View>Go to Next Buffer (keyboard shortcut: **Control-Page Down**) switches to the next buffer in the list.

View>Go to Recent Buffer (keyboard shortcut: **Control-‘**) switches to the buffer that was being edited prior to the current one.

2.2. Views

Each editor window is known as a *view*. It is possible to have multiple views open at once, and each view can be split into multiple panes.

View>New View creates a new view.

View>Close View closes the current view. If only one view is open, closing it will exit jEdit, unless background mode is on; see Chapter 1 for information about starting jEdit in background mode.

View>Splitting>Split Horizontally (shortcut: **Control-2**) splits the view into two text areas, above each other.

View>Splitting>Split Vertically (shortcut: **Control**-**3**) splits the view into two text areas, next to each other.

View>Splitting>Unsplit (shortcut: **Control**-**1**) removes all but the current text area from the view.

When a view is split, editing commands operate on the text area that has keyboard focus. To give a text area keyboard focus, click in it with the mouse, or use the following commands.

View>Splitting>Go to Previous Text Area (shortcut: **Alt**-**Page Up**) shifts keyboard focus to the previous text area.

View>Splitting>Go to Next Text Area (shortcut: **Alt**-**Page Down**) shifts keyboard focus to the next text area.

Clicking the text area with the right mouse button displays a popup menu. Both this menu and the tool bar at the top of the view offer quick mouse-based access to frequently-used commands. The contents of the tool bar and right-click menu can be changed in the Utilities>Global Options dialog box.

2.2.1. Window Docking

The file system browser, HyperSearch results window, and many plugin windows can optionally be docked into the view. This can be configured in the Docking pane of the Utilities>Global Options dialog box; see Section 6.3. Note that changes made in this option pane will not take effect immediately; you must restart jEdit or open a new view first.

When windows are docked into the view, the commands in the View>Docking menu (shortcuts: **Control**-**E** **1**, **2**, **3**, **4**) can be used to show or hide the top, bottom, left and right docking areas, respectively. Double-clicking on the borders of docking areas has the same effect.

2.2.2. The Status Bar

A *status bar* at the bottom of the view displays the following information, from left to right:

- The line number containing the caret

- The column position of the caret, with the leftmost column being 1.

  If the line contains tabs, the *file* position (where a hard tab is counted as one column) is shown first, followed by the *screen* position (where each tab counts for the number of columns until the next tab stop).

- Prompts displayed by commands such as those dealing with registers and markers (see Section 4.9 and Section 4.10)
- The current buffer's edit mode. Clicking this displays the Utilities>Buffer Options dialog box. See Section 5.1 and Section 6.1.
- The current buffer's character encoding. Clicking this displays the Utilities>Buffer Options dialog box. See Section 3.5 and Section 6.1.
- If multiple selection is enabled, the text multi is shown in black, otherwise it will be grayed out. Clicking her or pressing **Control-\** turns multiple selection on and off. See Section 4.2.2.
- If overwrite mode is enabled, the text over is shown in black, otherwise it will be grayed out. Clicking here or pressing **Insert** turns overwrite mode on and off. See Section 4.3.
- If portions of the buffer are invisible due to folding, the text fold is shown in black, otherwise it will be grayed out. See Section 5.6.
- If input/output operations are in progress, a small disk icon and progress bars for each running operation are displayed. Clicking here will display the Utilities>I/O Progress Monitor dialog box. See Section 3.8.

2.3. The Text Area

Text editing takes place in the text area. It behaves in a similar manner to many Windows and MacOS editors; the few unique features will be described in this section.

The text area will automatically scroll up or down if the caret is moved closer than three lines to the first or last visible line. This feature is called *electric scrolling* and can be disabled in the Text Area pane of the Utilities>Global Options dialog box; see Section 6.3.

To aid in locating the caret, the current line is drawn with a different background color. To make it clear which lines end with white space, end of line markers are drawn at the end of each line. Both these features can be disabled in the Text Area pane of the Utilities>Global Options dialog box.

The strip on the left of the text area is called a *gutter*. The gutter displays marker and register locations; it will also display line numbers if the View>Line Numbers (shortcut: **Control-E Control-T**) command is invoked.

2.4. Command Repetition

To repeat a command any number of times, invoke Utilities>Repeat Next Command (shortcut: **Control-Enter**) and enter the desired repeat count, followed by the command to repeat (either a keystroke or menu item selection). For example, "**Control-Enter 1 4 Control-D**" will delete 14 lines, and "**Control-Enter 8 #**" will insert "########" in the buffer.

If you specify a repeat count greater than 20, a confirmation dialog box will be displayed, asking if you really want to perform the action. This prevents you from hanging jEdit by executing a command too many times.

Chapter 3. Working With Files

3.1. Creating New Files

File>New File (shortcut: **Control-N**) opens a new untitled buffer. When it is saved, a file will be created on disk. Another way to create a new file is to specify a non-existent file name when starting jEdit from your operating system's command line.

3.2. Opening Files

File>Open File (shortcut: **Control-O**) displays a file selector dialog box and loads the specified file into a new buffer. Multiple files can be opened at once by holding down **Control** while clicking on them in the file system browser.

File>Insert File displays a file selector dialog box and inserts the specified file into the current buffer.

The File>Current Directory menu lists all files in the current buffer's directory.

The File>Recent Files menu lists recent files. When a recent file is opened, the caret is automatically moved to its previous location in that file. The number of recent files to remember can be changed and caret position saving can be disabled in the General pane of the Utilities>Global Options dialog box; see Section 6.3.

Files that you do not have write access to are opened in read-only mode, and editing will not be permitted.

GZipped Files

jEdit supports transparent editing of GZipped files; files with the `.gz` extension are automatically decompressed before loading, and compressed when saving.

3.3. Saving Files

Changed made to a buffer do not affect the file on disk until the buffer is *saved*.

File>Save (shortcut: **Control-S**) saves the current buffer to disk.

File>Save All Buffers (shortcut: **Control-E Control-S**) saves all open buffers to disk, asking for confirmation first.

File>Save As saves the buffer to a different specified file on disk. The buffer is then renamed, and subsequent saves also save to the specified file.

File>Save a Copy As saves the buffer to a different specified file on disk, but doesn't rename the buffer, and doesn't clear the "modified" flag.

3.3.1. Autosave and Crash Recovery

The autosave feature protects your work from computer crashes and such. Every 30 seconds, all buffers with unsaved changes are written out to their respective file names, enclosed in hash ("#") characters. For example, `program.c` will be autosaved to `#program.c#`.

Saving a buffer using one of the commands in the previous section automatically deletes the autosave file, so they will only ever be visible in the unlikely event of a jEdit (or operating system) crash.

If an autosave file is found while a buffer is being loaded, jEdit will offer to recover the autosaved data.

The autosave feature can be configured in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.

3.3.2. Backups

The backup feature can be used to roll back to the previous version of a file after changes were made. When a buffer is saved for the first time after being opened, its original contents are "backed up" under a different file name.

The default behavior is to back up the original contents to the buffer's file name suffixed with a tilde ("~"). For example, `paper.tex` will be backed up to `paper.tex~`.

The backup feature can also be configured to do any of the following:

- Save numbered backups, named *`filename~number~`*
- Add a prefix to the backed-up file name
- Adds a suffix, other than "~", to the backed-up file name
- Backups can optionally be saved in a specified backup directory, instead of the directory of the original file. This can reduce clutter

The above features can be configured in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.

3.4. Line Separators

The three major operating systems use different conventions to mark line endings in text files. The MacOS uses Carriage-Return characters (`\r` in Java-speak) for that purpose. Unix uses Newline characters (`\n`). Windows uses both (`\r\n`). jEdit can read and write files in all three formats.

When loading a file, the line separator used within is automatically detected, and will be used when saving a file back to disk. The line separator used when saving the current buffer can be changed in the Utilities>Buffer Options dialog box; see Section 6.1.

By default, new files are saved with your operating system's native line separator. This can be changed in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3. Note that changing this setting has no effect on existing files.

3.5. Character Encodings

Internally, Java programs like jEdit store text as a stream of 16-bit numerical values, with each value a character in the Unicode character set. Unicode is a standardized character set that can represent characters in almost all human languages.

Unfortunately, most other computer programs use far less flexible methods of storing text; therefore, if jEdit loaded and saved all files as raw Unicode, it would be useless.

To get around this, jEdit converts Unicode text to other character encodings and vice versa when loading and saving files. jEdit can use any encoding supported by the Java platform.

The default encoding, used to load and save files for which no other encoding is specified, can be set in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3. The setting is presented as an editable combo box; the combo box contains a few of the more frequently used encodings, but the Java platform defines practically hundreds more you can use.

Unfortunately, there is no programmical way to obtain a list of all supported encodings, and the set is constantly changing with each Java version. So to play it safe, jEdit has a few pre-defined defaults, but allows you to use any other supported encoding, assuming you know its name.

Unless you change the default encoding, jEdit will use your operating system's native encoding; `MacRoman` on the MacOS, `Cp1252` on Windows, and `8859_1` on Unix.

The File>Open With Encoding lets you open a file with an encoding other than the default. The menu contains a set of items, one for each common encoding, along with System Default and jEdit Default commands. Invoking a menu item displays the usual file dialog box, and opens the selected file with the chosen encoding.

The Open With Other Encoding command in the same menu lets you enter an arbitriary encoding name, assuming it is supported by your Java implementation.

Once a file has been opened, the encoding to use when saving it can be set in the Utilities>Buffer Options dialog box.

The current buffer's encoding is shown in the status bar. If a file is opened without an explicit encoding specified, jEdit will use the encoding last used when working with that file, if the file is in the recent file list. Otherwise, the default encoding will be used.

3.5.1. Commonly Used Encodings

The most frequently-used character encoding is ASCII, or “American Standard Code for Information Interchange”. ASCII encodes Latin letters used in English, in addition to numbers and a range of punctuation characters. The ASCII character set consists of 127 characters only, and it is unsuitable for anything but English text (and other file types which only use English characters, like most program source). jEdit will load and save files as 7-bit ASCII if the `ASCII` encoding is used.

Because ASCII is unsuitable for international use, most operating systems use an 8-bit extension of ASCII, with the first 127 characters remaining the same, and the rest used to encode accents, ulmauts, and various less frequently used typographical marks. Unfortunately, the three major operating systems all extend ASCII in a different way. Files written by Macintosh programs can be read using the `MacRoman` encoding; Windows text files are usually stored as `Cp1252`. In the Unix world, the `8859_1` (otherwise known as Latin1) character encoding has found widespread usage.

Windows users are accustomized to dealing with files in a wide range of character sets, known as *code pages*. Java supports a large number of code pages; the encoding name consists of the text “Cp”, followed by a number.

Raw Unicode files are quite rare, but can be read and written with the `Unicode` encoding. One reason raw Unicode has not found widespread usage for storing files on disk is that each character takes up 16 bits. Most other character sets devote 8 bits per character, which saves space. The `UTF8` encoding encodes frequently-used Unicode characters as 8 bits, with less-frequent ones stretching up to 24 bits. This saves space but allows the full range of Unicode characters to be represented.

Many common cross-platform international character sets are supported; `KOI8_R` for Russian text, `Big5` and `GBK` for Chinese, and `SJIS` for Japanese.

Java even supports a few downright obscure encodings, such as the `EBCDIC` character encoding used on IBM mainframes.

3.6. The File System Browser

Utilities>File System Browser displays a file system browser. By default, the file system browser is shown in a floating window; it can be set to dock into the view in the Docking pane of the Utilities>Global Options dialog box; see Section 2.2.1.

The directory to browse is specified in the Path text field. A subset of the current directory to display can be specified in the form of a glob pattern in the Filter text field. See Appendix D for information about glob patterns. Both text fields remember previously entered strings; see Appendix C for details.

You can view an entire directory hierarchy at once by clicking the expander controls next to directories in the tree.

The toolbar buttons perform the following actions, from left to right:

- Up - displays the current directory's parent in the file system view. The popup arrow next to this button displays a menu listing all the parent directories of the current directory, up to the filesystem root
- Reload - reloads the file list
- Local Drives - displays all local drives. On Windows, this will be a list of drive letters; on Unix, the list will only contain one entry, the root directory
- Home Directory - displays your home directory in the file system browser
- Parent Directory of Current Buffer - displays the directory containing the current buffer in the file system browser

Clicking the More button displays a menu containing several less frequently-used commands:

- Show Hidden Files - a check box menu item that controls if hidden files will be shown in the file list
- New Directory - creates a new directory, prompting for the desired name
- Search in Directory - displays the search and replace dialog box for searching in all files in the current directory that match the current filename filter. See Section 4.11 for

information about the search and replace feature

- Add to Favorites - adds the currently selected (or the currently displayed, if there is nothing selected) directory to the favorites list
- Go to Favorites - displays the favorites list. To remove a directory from the list, right-click on it and select Delete from the resulting popup menu

Right-clicking on a file in the file system browser displays a popup menu, containing commands for manipulating that file, in addition to all the commands from the More menu. If the file is already open, the popup will have commands to display it or close it. Unopened file popups have commands for opening, opening with a different encoding, deleting and renaming. Note that attempting to delete a directory containing files will give an error; only empty directories may be deleted.

The file system browser can be navigated from the keyboard:

- **Enter** - opens the currently selected file or directory
- **Left** - goes to the current directory's parent
- **Up** - selects previous file in list
- **Down** - selects next file in list
- Typing the first few characters of a file's name will select that file

The file system view must have keyboard focus for these keys to work. In the Open File dialog box, it is given keyboard focus by default. In other instances, it can be given keyboard focus by clicking with the mouse.

The file system browser can be customized in the File System Browser pane of the Utilities>Global Options dialog box. The following settings can be changed:

- The directory to display initially (either the directory containing the current buffer, your home directory, the favorites list, or the most recently visited directory)
- If icons should be shown (disabling icons can save a lot of screen space)
- If hidden files should be shown by default

- If the file list should be sorted
- If the sort should be case-insensitive
- If files are directories should be sorted together, as opposed to directories always being at the top of the list
- If double-clicking an open file should close it
- If the file name filter in Open and Save dialog boxes should be based on the current buffer's name

3.7. Reloading Files

If an open buffer is modified on disk by another application, a warning dialog box is displayed, offering to either continue editing (and lose changes made by the other application) or reload the buffer from disk (and lose any usaved changes). This feature can be disabled in the General pane of the Utilities>Global Options dialog box; see Section 6.3.

File>Reload can be used to discard unsaved changes and reload the current buffer from disk at any other time; a confirmation dialog box will be displayed first if the buffer has unsaved changes.

File>Reload All Buffers discards unsaved changes in all open buffers and reload them from disk, asking for confirmation first.

3.8. Multi-Threaded I/O

To improve responsiveness and perceived performance, jEdit executes all input/output operations asynchronously. While I/O is in progress, a small disk icon and progress meters for each running operation are shown in the status bar. The Utilities>I/O Progress Monitor command displays a window with more detailed status information. Requests can also be aborted in this window. Note that aborting a buffer save can result in data loss.

By default, four I/O threads are created, which means that up to four buffers can be loaded or saved simultaneously. The number of threads can be changed in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3. Setting the number to zero disables multi-threaded I/O completely; doing this is not recommended.

3.9. Printing Files

File>Print (shortcut: **Control**-**P**) will print the current buffer. By default, the printed output will have syntax highlighting, and each page will have a header with the file name, and a footer with the current date/time and page number. The appearance of printed output can be customized in the Printing pane of the Utilities>Global Options dialog box. The following settings can be changed:

- The font to use when printing
- If a header with the file name should be printed on each page
- If a footer with the page number and current date should be printed on each page
- If line numbers should be printed
- If the output should be styled according to the current mode's syntax highlighting rules
- If the output should be colored according to the current mode's syntax highlighting rules (might look bad on grayscale printers)
- The page margins

3.10. Closing Files and Exiting jEdit

File>Close Buffer (shortcut: **Control**-**W**) closes the current buffer. If it has unsaved changes, jEdit will ask if they should be saved first.

File>Close All Buffers (shortcut: **Control**-**E** **Control**-**W**) closes all buffers. If any buffers have unsaved changes, they will be listed in a dialog box where they can be saved or discarded. In the dialog box, multiple buffers to operate on at once can be selected by clicking on them in the list while holding down **Control**.

File>Exit (shortcut: **Control**-**Q**) will completely exit jEdit.

Chapter 4. Editing Text

4.1. Moving The Caret

The most direct way to move the caret is to click the mouse at the desired location in the text area. It can also be moved using the keyboard.

The **Left**, **Right**, **Up** and **Down** keys move the caret in the respective direction, and the **Page Up** and **Page Down** keys move the caret up and down one screenful, respectively.

When pressed once, the **Home** key moves the caret to the first non-whitespace character of the current line. Pressing it a second time moves the caret to the beginning of the line. Pressing it a third time moves the caret to the first visible line.

The **End** key behaves in a similar manner, going to the last non-whitespace character, the end of the line, and finally to the last visible line.

Control-Home and **Control-End** move the caret to the beginning and end of the buffer, respectively.

More advanced caret movement is covered in Section 4.5, Section 4.6 and Section 4.7.

4.2. Selecting Text

A *selection* is a a block of text marked for further manipulation. Unlike many other applications, jEdit supports both range and rectangular selections, and several chunks of text can be selected simultaneously.

Dragging the mouse creates a range selection from where the mouse was pressed to where it was released. Holding down **Shift** while clicking a location in the buffer will create a selection from the caret position to the clicked location.

Holding down **Shift** in addition to a caret movement key (**Left**, **Up**, **Home**, etc) will extend the selection in the specified direction. If no selection exists, one will be created.

Edit>Select All (shortcut: **Control-A**) selects the entire buffer.

Edit>Select None (shortcut: **Escape**) deactivates the selection.

4.2.1. Rectangular Selection

Holding down **Control** and dragging will create a rectangular selection. Holding down **Shift** and **Control** while clicking a location in the buffer will create a rectangular selection from the caret position to the clicked location.

It is possible to select a rectangle with zero width but non-zero height. This can be used to insert a new column between two existing columns, for example. Such zero-width selections are shown as a thin vertical line.

Rectangles can be deleted, copied, pasted, and operated on using ordinary editing commands.

> **Note:** Rectangular selections are implemented using character offsets, not absolute screen positions, so they might not behave as you might expect if a proportional-width font is being used, or hard tabs are enabled. For information about changing the font used in the text area, see Section 6.3. For more information about hard and soft tabs, see Section 5.4.1.

4.2.2. Multiple Selection

Pressing **Control**-\ turns multiple selection mode on and off. In multiple selection mode, multiple fragments of text can be selected and operated on simultaneously, and the text `multi` is shown in black in the status bar.

While multiple selection mode is active, you can click and drag the mouse to reposition the caret and create new selections.

Various jEdit commands behave differently with multiple selections:

- Commands that copy text place the contents of each selection, separated by line breaks, in the specified register. Note that the selections are appended in the order they were created, not the order they appear in the buffer. For example, if you select a chunk of text near the end of the buffer, another near the beginning, then invoke Copy, the clipboard will contain the later chunk, followed by a line break, followed by the earlier chunk.

- Commands that insert (or paste) text replace each selection with the entire text that is being inserted.
- Commands that filter text (such as Spaces to Tabs, Range Comment, and even Replace in Selection) operate on each selection, in turn.
- Line-based commands (such as Shift Indent Left, Shift Indent Right, and Line Comment) operate on each line that contains at least one selection, in addition to the line containing the caret.
- Caret movement commands that would normally deactivate the selection (such as the arrow keys, while **Shift** is not being held down), move the caret, leaving the selection as-is.
- Some older plugins may not support multiple selection at all.

Tip: Deactivating multiple selection mode while a fragmented selection exists will leave the selection in place, but it will prevent you from making further fragmented selections. If a fragmented selection exists but multiple selection mode is not active, the text `multi` will be shown in dark blue in the status bar.

4.3. Inserting and Deleting Text

Text entered at the keyboard is inserted into the buffer. If overwrite mode is on, one character is deleted from in front of the caret position for every character that is inserted. To activate overwrite mode, press **Insert**. The caret is drawn as horizontal line while in overwrite mode; the text `over` also appears in the status bar.

Inserting text while there is a selection will replace the selection with the inserted text.

Inserting text at the end of a line beyond the wrap column will automatically break the line at the appropriate word boundary. The wrap column is indicated in the text area as a faint blue line and its location (specified in number of character positions from the left margin) can be changed in one of several ways:

- On a global or mode-specific basis in the Editing and Mode-Specific panes of the Utilities>Global Options dialog box; see Section 6.3.
- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box; see Section 6.1.
- In the current buffer for future editing sessions by placing the following in one of the first 10 lines of the buffer, where `column` is the desired wrap column position:

```
:maxLineLen=column:
```

To disable word wrap completely, set the wrap column to 0 using any of the above means.

> **Note:** Word wrap is implemented using character offsets, not screen positions, so it might not behave like you expect if a proportional-width font is being used. For information about changing the font used in the text area, see Section 6.3.

When inserting text, keep in mind that the **Tab** and **Enter** keys might not behave entirely like you expect because of various indentation features; see Section 5.4 for details.

The simplest way to delete text is with the **Backspace** and **Delete** keys. If nothing is selected, they delete the character before or after the caret, respectively. If a selection exists, both delete the selection.

More advanced deletion commands are described in Section 4.5, Section 4.6 and Section 4.7.

4.4. Undo and Redo

Edit>Undo (shortcut: **Control-Z**) undoes the effects of the most recent text editing command. For example, this can be used to restore unintentionally deleted text. More complicated operations, such as a search and replace, can also be undone. By default, the undo queue remembers the last 100 edits; older edits are discarded. The undo queue size can be changed in the Editing pane of the Utilities>Global Options dialog box.

Edit>Redo (shortcut: **Control-R**) goes forward in the undo queue, redoing changes which were undone. For example, if some text was inserted, Undo will remove it from the buffer. Redo will insert it again.

4.5. Working With Words

Holding down **Control** in addition to **Left** or **Right** moves the caret a word at a time. Holding down **Shift** and **Control** in addition to **Left** or **Right** extends the selection a word at a time.

A single word can be selected by double-clicking with the mouse, or using the Edit>Text>Select Word command (shortcut: **Control-E W**). A selection that begins and ends on word boundaries can be created by double-clicking and dragging.

Pressing **Control** in addition to **Backspace** or **Delete** deletes the word before or after the caret, respectively.

Edit>Word Count displays a dialog box with the number of characters, words and lines in the current buffer.

Edit>Complete Word (shortcut: **Control-B**) searches the current buffer for possible completions of the current word. This feature be used to avoid retyping previously entered identifiers in program source, for example.

If there is only one completion, it will be inserted into the buffer immediately. If multiple completions were found, they will be listed in a popup below the caret position. To insert a completion from the list, either click it with the mouse, or select it using the **Up** and **Down** keys and press **Enter**. To close the popup without inserting a completion, press **Escape**.

4.6. Working With Lines

An entire line can be selected by triple-clicking with the mouse, or using the Edit>Text>Select Line command (shortcut: **Control-E L**). A selection that begins and ends on line boundaries can be created by triple-clicking and dragging.

Edit>Go to Line (shortcut: **Control-L**) displays an input dialog box and moves the caret to the specified line number.

Edit>Select Line Range (shortcut **Control-E Control-L**) selects all text between between two specified line numbers, inclusive.

Edit>Text>Join Lines (shortcut: **Control-J**) removes any whitespace from the start of the next line and joins it with the current line. For example, invoking Join Lines on the first line of the following Java code:

```
new Widget(Foo
       .createDefaultFoo());
```

Will change it to:

```
new Widget(Foo.createDefaultFoo());
```

Edit>Text>Delete Line (shortcut: **Control-D**) deletes the current line.

Edit>Text>Delete to Start Of Line (shortcut: **Shift-Backspace**) deletes all text from the start of the current line to the caret.

Edit>Text>Delete to End Of Line (shortcut: **Shift-Delete**) deletes all text from the caret to the end of the current line.

Edit>Text>Remove Trailing Whitespace (shortcut: **Control-E R**) removes all whitespace from the end of each selected line, or the current line if there is no selection.

4.7. Working With Paragraphs

As far as jEdit is concerned, “paragraphs” are delimited by double newlines. This is also how TeX defines a paragraph. Note that jEdit doesn’t parse HTML files for “<P>” tags, nor does it support paragraphs delimited only by a leading indent.

Holding down **Control** in addition to **Up** or **Down** moves the caret to the previous and next paragraph, respectively. As with other caret movement commands, holding down **Shift** in addition to the above extends the selection, a paragraph at a time.

Edit>Text>Select Paragraph (shortcut: **Control-E P**) selects the paragraph containing the caret.

Edit>Text>Delete Paragraph (shortcut: **Control-E D**) deletes the paragraph containing the caret.

Edit>Text>Format Paragraph (shortcut: **Control-E F**) splits and joins lines in the current paragraph to make them fit within the wrap column position. See Section 4.3 for information and word wrap and changing the wrap column.

4.8. Scrolling

View>Scrolling>Scroll to Current Line (shortcut: **Control-E Control-J**) centers the line containing the caret on the screen.

View>Scrolling>Center Caret on Screen (shortcut: **Control-E Control-I**) moves the caret to the line in the middle of the screen.

View>Scrolling>Line Scroll Up (shortcut: **Control-’**) scrolls the text area up by one line.

View>Scrolling>Line Scroll Down (shortcut: **Control-/**) scrolls the text area down by one line.

View>Scrolling>Page Scroll Up (shortcut: **Alt-’**) scrolls the text area up by one screenful.

View>Scrolling>Page Scroll Down (shortcut: **Alt-/**) scrolls the text area down by one screenful.

The above scrolling commands differ from the caret movement commands in that they don’t actually move the caret; they just change the scroll bar position.

View>Scrolling>Synchronized Scrolling is a check box menu item, that if selected, forces scrolling performed in one text area to be propagated to all other text areas in the current view. Invoking the command a second time disables the feature.

4.9. Transferring Text

jEdit provides a rich set of commands for moving and copying text using *registers*. A register is a holding area with a single-character name that can hold once piece of text at a time. Registers are global to the editor; all buffers share the same set. With the exception of the clipboard, register contents are only accessable inside jEdit.

jEdit has three sets of commands for working with registers. The “quick copy” and system clipboard features allow easy access to two specific registers. A third set of commands allows access to all other registers.

4.9.1. Quick Copy

Holding down the **Alt** key and clicking the left mouse button in the text area inserts the most recently selected text at the clicked location. If you have a three-button mouse, you can simply click the middle mouse button, without holding down **Alt**.

Internally, this is implemented by storing the most recently selected text in the `%` register (recall that registers have single-character names).

X Windows primary selection

If jEdit is being run under Java 2 version 1.4 on Unix, you will be able to transfer text with other X Windows applications using the quick copy feature. On other platforms and Java versions, the contents of the quick copy register are only accessable from within jEdit.

4.9.2. The System Clipboard

Quick copy is very useful in many situations, but it has a number of drawbacks; it requires the use of the mouse, it cannot be used to replace an existing selection, and it cannot be used to transfer text between different applications (unless you are using Java 2 version 1.4 on Unix).

The system clipboard, internally known as the `$` register, does not have these limitations, but can be slightly less convinient to use.

Edit>Cut (shortcut: **Control-X**) places the selected text in the clipboard and removes it from the buffer.

Edit>Copy (shortcut: **Control-C**) places the selected text in the clipboard and leaves it in the buffer.

Edit>Cut Append (shortcut: **Control-E Control-U**) appends the selected text to the clipboard, then removes it from the buffer. After this command has been invoked, the clipboard will consist of the former clipboard contents, followed by a newline, followed by the selected text.

Edit>Copy Append (shortcut: **Control-E Control-A**) appends the selected text to the clipboard, and leaves it in the buffer. After this command has been invoked, the clipboard will consist of the former clipboard contents, followed by a newline, followed by the selected text.

Edit>Paste (shortcut: **Control-V**) inserts the clipboard contents in place of the selection (or at the caret position, if there is no selection).

4.9.3. General Register Commands

These commands are slightly less convinient to use than the two methods of transferring text described above, but have the advantage that they allow any number of strings to be copied simultaneously.

These commands all expect a single-character register name to be typed at the keyboard after the command is invoked, and subsequently operate on that register. Pressing **Escape** instead of specifying a register name will cancel the operation.

Edit>Registers>Cut to Register (shortcut: **Control-R Control-X** ***key***) stores the selected text in the specified register, removing it from the buffer.

Edit>Registers>Copy to Register (shortcut: **Control-R Control-C** ***key***) stores the selected text in the specified register, leaving it in the buffer.

Edit>Registers>Cut Append to Register (shortcut: **Control-R Control-U** ***`key`***) adds the selected text to the existing contents of the specified register, and removes it from the buffer.

Edit>Registers>Copy Append to Register (shortcut: **Control-R Control-A** ***`key`***) adds the selected text to the existing contents of the specified register, without removing it from the buffer.

Edit>Registers>Paste from Register (shortcut: **Control-R Control-V** ***`key`***) replaces the selection with the contents of the specified register.

Edit>Paste Previous (shortcut: **Control-E Control-V**) displays a dialog box listing recently copied and pasted text. By default, the last 20 strings are remembered; this can be changed in the General pane of the Utilities>Global Options dialog box; see Section 6.3.

Edit>Registers>View Registers displays a dialog box for viewing the contents of registers (including the clipboard).

4.10. Markers

Each buffer can have any number of *markers* defined, pointing to specific locations within that buffer. Each line in a buffer can have at most one marker set pointing to it. Markers are persistent; they are saved to `.`*`filename`*`.marks`, where *`filename`* is the file name. (The dot prefix makes the markers file hidden on Unix systems.) Marker saving can be disabled in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.

Markers are listed in the Markers menu; selecting a marker from this menu is the simplest way to return to its location. Each marker can optionally have a single-character shortcut; markers without a shortcut can only be returned to using the menu, markers with a shortcut can be accessed more quickly from the keyboard.

Lines which contain markers are indicated in the gutter with a highlight. Moving the mouse over the highlight displays a tool tip showing the marker's shortcut, if it has one. See Section 2.3 for information about the gutter.

Markers>Add/Remove Marker (shortcut: **Control-E Control-M**) adds a marker without a shortcut pointing to the current line. If a marker is already set on the current line, the marker is removed instead. If text is selected, markers are added to the first and last line of each selection.

Markers>Add Marker With Shortcut (shortcut: **Control-T** ***key***) reads the next character entered at the keyboard, and adds a marker with that shortcut pointing to the current line. If a previously-defined marker already has that shortcut, it will no longer have that shortcut, but will remain in the buffer. Pressing **Escape** instead of specifying a marker shortcut after invoking the command will cancel the operation.

Markers>Go to Marker (shortcut: **Control-Y** ***key***) reads the next character entered at the keyboard, and moves the caret to the location of the marker with that shortcut. Pressing **Escape** instead of specifying a marker shortcut after invoking the command will cancel the operation.

Markers>Select to Marker (shortcut: **Control-U** ***key***) reads the next character entered at the keyboard, and extends the selection to the location of the marker with that shortcut. Pressing **Escape** instead of specifying a marker shortcut after invoking the command will cancel the operation.

Markers>Swap Caret and Marker (shortcut: **Control-U** ***key***) reads the next character entered at the keyboard, and swaps the position of the caret with the location of the marker with that shortcut. Pressing **Escape** instead of specifying a marker shortcut after invoking the command will cancel the operation.

Markers>Go to Previous Marker (shortcut: **Alt-Up**) goes to the nearest marker before the caret position.

Markers>Go to Next Marker (shortcut: **Alt-Down**) goes to the nearest marker after the caret position.

Markers>Remove All Markers removes all markers set in the current buffer.

4.11. Search and Replace

4.11.1. Searching For Text

Search>Find (shortcut: **Control-F**) displays the search and replace dialog box.

The search string can be entered in the Search for text field. This text field remembers previously entered strings; see Appendix C for details.

The search can be made case insensitive (for example, searching for “Hello” will match “hello”, “HELLO” and “HeLlO”) by selecting the Ignore case check box. Regular expressions may be used to match inexact sequences of text if the Regular expressions check box is selected; see Appendix E for more information about regular expressions. Note that regular expressions can only be used when searching forwards.

Clicking Find will locate the next (or previous, if searching backwards) occurrence of the search string after the caret position. If the Keep dialog check box is selected, the dialog box will remain open; otherwise, it will be closed after the search string is located.

If no occurrences could be found and the Auto wrap check box is selected, the search will automatically be restarted and a message will be shown in the status bar to indicate that. If the check box is not selected, a dialog box will be displayed, offering to restart the search.

Search>Find Next (shortcut: **Control-G**) locates the next occurrence of the most recent search string without displaying the search and replace dialog box.

Search>Find Previous (shortcut: **Control-H**) locates the previous occurrence of the most recent search string without displaying the search and replace dialog box.

Search>Find Selection (shortcut: **Control-E Control-F**) displays the search and replace dialog box with the currently selected text entered in the Search for text field.

4.11.2. Replacing Text

Occurrences of the search string can be replaced with either a replacement string or the result of a BeanShell script snippet. Two radio buttons in the search and replace dialog box can be used to choose between these two behaviors.

The replace string text field remembers previously entered strings; see Appendix C for details.

Clicking Replace will perform a replacement in each text selection. Clicking Replace & Find will perform a replacement in each text selection and locate the next occurrence of the search string. Clicking Replace All will perform a replacement in each buffer to be searched.

In text replacement mode, the search string is replaced with the replacement string. If regular expressions are enabled, positional parameters (`$0`, `$1`, `$2`, and so on) can be used to insert the contents of matched subexpressions in the replacement text; see Appendix E for more information.

In BeanShell replacement mode, the search string is replaced with the return value of a BeanShell snippet. The following predefined variables can be referenced in the snippet:

- `_0` – the text to be replaced
- `_1` - `_9` – if regular expressions are enabled, these contain the values of matched subexpressions.

BeanShell syntax and features are covered in great detail in Part III in *jEdit 3.2 User's Guide*, but here are some examples:

To convert all HTML tags to lower case, search for the following regular expression:

```
<(.*?)>
```

Replacing it with the following BeanShell snippet:

```
"<" + _1.toLowerCase() + ">"
```

To replace arithmetic expressions between curly braces with their result, search for the following regular expression:

```
\{(.+)\}
```

Replacing it with the following BeanShell snippet:

```
eval(_1)
```

These two examples only scratch the surface; the possibilities are endless.

4.11.3. HyperSearch

If the HyperSearch check box in the search and replace dialog box is selected, clicking Find will list all occurrences of the search string, rather than locating them one by one.

By default, HyperSearch results are shown in a floating window; the window can be set to dock into the view in the Docking pane of the Utilities>Global Options dialog box; see Section 2.2.1.

Running searches can be stopped in the Utilities>I/O Progress Monitor dialog box.

4.11.4. Multiple File Search

Searching, replacement and HyperSearch can also be performed in all open buffers or all files in a directory.

If the All buffers radio button in the search and replace dialog box is selected, all open buffers whose names match the glob pattern entered in the Filter text field will be searched. See Appendix D for more information about glob patterns.

If the Directory radio button is selected, all files in the directory whose names match the glob will be searched. The directory to search in can either be entered in the Directory text field, or chosen in a file selector dialog box by clicking Choose. If the Search subdirectories check box is selected, all subdirectories of the specified directory will also be searched. Keep in mind that searching through directories with many files can take a long time and consume a large amount of memory.

The Directory and Filter text fields remember previously entered strings; see Appendix C for details.

Two convinience commands are provided for performing multiple file searches.

Search>Search in Open Buffers (shortcut: **Control-E Control-B**) displays the search and replace dialog box, and selects the All buffers radio button.

Search>Search in Directory (shortcut: **Control-E Control-D**) displays the search and replace dialog box, and selects the Directory radio button.

4.11.5. The Search Bar

The search bar at the top of the view provides a convenient way to perform simple searches without opening the search and replace dialog box first. Neither multiple file search nor replacement can be done from the search bar.

Unless the HyperSearch check box is selected, the search bar will perform an *incremental search*. In incremental search mode, the first occurrence of the search string is located in the current buffer as is it is being typed. Pressing **Enter** and **Shift-Enter** searches for the next and previous occurrence, respectively. Once the desired occurrence has been found, press **Escape** to return keyboard focus to the text area.

If the HyperSearch check box is selected, entering a search string and pressing **Enter** will perform a HyperSearch. When in HyperSearch mode, the search bar remembers previously entered strings; see Appendix C for details.

The search bar can be accessed from the keyboard using the Search>Quick Incremental Search (shortcut: **Control-,**) and Search>Quick HyoerSearch (shortcut: **Control-.**) commands.

The search bar can be disabled in the General pane of the Utilities>Global Options dialog box.

Note that incremental searches cannot be not recorded in macros. Use the search and replace dialog box instead.

Chapter 5. Editing Source Code

5.1. Edit Modes

An *edit mode* specifies syntax highlighting rules, auto indent behavior, and various other customizations for editing a certain file type. This section only covers using and selecting edit modes; information about writing your own can be found in Part II in *jEdit 3.2 User's Guide*.

5.1.1. Mode Selection

When a file is opened, jEdit first checks the file name against a list of known patterns. For example, files whose names end with ".c" are edited in C mode, and files named `Makefile` are edited in Makefile mode. If a suitable match based on file name cannot be found, jEdit checks the first line of the file. For example, files whose first line is "#!/bin/sh" are edited in shell script mode.

If automatic mode selection is not appropriate, the edit mode can be specified manually. The current buffer's edit mode can be set on a one-time basis in the Utilities>Buffer Options dialog box; see Section 6.1. To set a buffer's edit mode for future editing sessions, place the following in one of the first 10 lines of the buffer, where *`edit mode`* is the name of the desired edit mode:

```
:mode=edit mode:
```

5.1.2. Syntax Highlighting

Syntax highlighting is the display of programming language tokens using different fonts and colors. This makes code easier to follow and errors such as misplaced quotes easier to spot. All edit modes except for the plain text mode perform syntax highlighting.

The colors and styles used to highlight syntax tokens can be changed in the Styles pane of the Utilities>Global Options dialog box; see Section 6.3.

Syntax highlighting can be enabled or disabled in one of several ways:

- On a global or mode-specific basis in the Editing and Mode-Specific panes of the Utilities>Global Options dialog box.
- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box; see Section 6.1.
- In the current buffer for future editing sessions, by placing the following in one of the first 10 lines of the buffer, where *`flag`* is either “true” or “false”:

  ```
  :syntax=flag:
  ```

5.2. Abbreviations

Using abbreviations reduces the time spent typing long but commonly used strings. For example, in Java mode, the abbreviation “sout” is defined to expand to “System.out.println()”, so to insert “System.out.println()” in a Java buffer, you only need to type “sout” followed by **Control-;**. Each abbreviation can either be global, in which case it will expand in all edit modes, or mode-specific. Abbreviations can be edited in the Abbreviations pane of the Utilities>Global Options dialog box; see Section 6.3. The Java, SGML and VHDL edit modes include some pre-defined abbreviations you might find useful.

Edit>Expand Abbreviation (keyboard shortcut: **Control-;**) attempts to expand the word before the caret. If no expansion could be found, it will offer to define one.

Automatic abbreviation expansion can be enabled in the Abbreviations pane of the Utilities>Global Options dialog box; see Section 6.3. If enabled, pressing the space bar after entering an abbreviation will automatically expand it.

If automatic expansion is enabled, a space can be inserted without expanding the word before the caret by pressing **Control-E V Space**.

5.2.1. Positional Parameters

Positional parameters are an advanced feature that make abbreviations much more useful. The best way to describe them is with an example.

Suppose you have an abbreviation “F” that is set to expand to the following:

```
for(int $1 = 0; $1 < $2; $1++)
```

Now, simply entering “F” in the buffer and expanding it will insert the above text as-is. However, if you expand `F#j#array.length#`, the following will be inserted:

```
for(int j = 0; j < array.length; j++)
```

Expansions can contain up to nine positional parameters. Note that a trailing hash character (“#”) must be entered when expanding an abbreviation with parameters.

5.3. Bracket Matching

Misplaced and unmatched brackets are one of the most common syntax errors encountered when writing code. jEdit has several features to make brackets easier to deal with.

If the character immediately before the caret position is a bracket, the matching one will be highlighted (assuming it is visible on the screen). Bracket highlighting can be disabled in the Text Area pane of the Utilities>Global Options dialog box; see Section 6.3.

Edit>Source Code>Go to Matching Bracket (shortcut: **Control-]**) goes to the bracket matching the one before the caret.

Double-clicking on a bracket in the text area will select all text between the bracket and the one matching it.

Edit>Source Code>Select Code Block (shortcut: **Control-[**) selects all text between the two brackets nearest to the caret.

Edit>Source Code>Go to Previous Bracket (shortcut: **Control-E [**) moves the caret to the previous opening bracket.

Edit>Source Code>Go to Next Bracket (shortcut: **Control-E]**) moves the caret to the next closing bracket.

> **Note:** jEdit's bracket matching algorithm only checks syntax tokens with the same type as the original bracket for matches. So brackets inside string literals and comments will not cause problems, as they will be skipped.

5.4. Tabbing and Indentation

jEdit makes a distinction between the *tab width*, which is is used when displaying tab characters, and the *indent width*, which is used when a level of indent is to be added or removed, for example by mode-specific smart indent routines. Both can be changed in one of several ways:

- On a global or mode-specific basis in Editing and Mode-Specific panes of the the Utilities>Global Options dialog box.
- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box; see Section 6.1.
- In the current buffer for future editing sessions by placing the following in one of the first 10 lines of the buffer, where *n* is the desired tab width, and *m* is the desired indent width:

  ```
  :tabSize=n:indentSize=m:
  ```

Edit>Source Code>Shift Indent Left (shortcut: **Alt-Left**) adds one level of indent to each selected line, or the current line if there is no selection.

Edit>Source Code>Shift Indent Right (shortcut: **Alt-Right**) removes one level of indent from each selected line, or the current line if there is no selection.

5.4.1. Soft Tabs

Because files indented using tab characters may look less than ideal when viewed on a system with a different default tab size, it is sometimes desirable to use multiple spaces, known as *soft tabs*, instead of real tab characters, to indent code.

Soft tabs can be enabled or disabled in one of several ways:

- On a global or edit mode-specific basis in the Editing and Mode-Specific panes of the Utilities>Global Options dialog box.
- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box; see Section 6.1.
- In the current buffer for future editing sessions by placing the following in one of the first 10 lines of the buffer, where *flag* is either “true” or “false”:

  ```
  :noTabs=flag:
  ```

Changing the soft tabs setting has no effect on existing tab characters; it only affects subsequently-inserted tabs.

Edit>Source Code>Spaces to Tabs converts soft tabs to hard tabs in the current selection.

Edit>Source Code>Tabs to Spaces converts hard tabs to soft tabs in the current selection.

5.4.2. Automatic Indent

The auto indent feature inserts the appropriate number of tabs or spaces at the beginning of a line.

If indent on enter is enabled, pressing **Enter** will create a new line with the appropriate amount of indent automatically. If indent on tab is enabled, pressing **Tab** on an unindented line will insert the appropriate amount of indentation. Pressing it again will insert a tab character.

By default, indent on enter is enabled and indent on tab is disabled. This can be changed in one of several ways:

- On a global or mode-specific basis in the Editing and Mode-Specific panes of the Utilities>Global Options dialog box.

- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box; see Section 6.1.
- In the current buffer for future editing sessions by placing the following in the first 10 lines of a buffer, where *`flag`* is either "true" or "false":

  ```
  :indentOnEnter=flag:indentOnTab=flag:
  ```

Auto indent behavior is mode-specific. In most edit modes, the indent of the previous line is simply copied over. However, in C-like languages (C, C++, Java, JavaScript), curly brackets and language statements are taken into account and indent is added and removed as necessary.

Edit>Source Code>Indent Selected Lines (shortcut: **Control-I**) indents all selected lines, or the current line if there is no selection.

To insert a literal tab or newline without performing indentation, prefix the tab or newline with **Control-E V**. For example, to create a new line without any indentation, type **Control-E V Enter**.

5.5. Commenting Out Code

Most programming and markup languages support "comments", or regions of code which are ignored by the compiler/interpreter. jEdit has commands which make inserting comments more convenient.

Edit>Source Code>Range Comment (shortcut: **Control-E Control-C**) encloses the selection with comment start and end strings, for example `/*` and `*/` in Java mode.

Comment start and end strings can be changed on a mode-specific basis in the Mode-Specific pane of the Utilities>Global Options dialog box, or on a buffer-specific basis using buffer-local properties. For example, placing the following in one of the first 10 lines of a buffer will change the range comment strings to `(*` and `*)`:

```
:commentStart=(*:commentEnd=*):
```

Edit>Source Code>Line Comment (shortcut: **Control-E Control-K**) inserts the line comment string, for example `//` in Java mode, at the start of each selected line.

The line comment string can be changed on a mode-specific basis in the Mode-Specific pane of the Utilities>Global Options dialog box, or on a buffer-specific basis using buffer-local properties. For example, placing the following in one of the first 10 lines of a buffer will change the line comment string to `#`:

```
:lineComment=#:
```

5.6. Folding

The folding feature allows lines to be hidden or shown depending on their indent level. Since most programming languages use indentation to nest code, folding away lines with a lot of indent has the effect of displaying an “overview” of the file only, while displaying higher indent levels “zooms in” on the contents and shows more “detail”.

A set of consecutive lines with the same leading indent is referred to as a *fold*. The visibility of each fold can be set independently. A fold that is hidden is said to be “collapsed”; a visible fold is “expanded”. Text hidden by folding is still present in the buffer, and can be made visible again using the appropriate commands. Cursor movement commands skip over the hidden text, but text manipulation commands act on it.

The initial fold visibility level, in multiples of the indent width, can be specified on a mode-specific or global basis in the Utilities>Global Options dialog box; see Section 6.3. Folds with a level higher than this will be automatically collapsed after a buffer is loaded. Setting this value to zero makes all folds expanded initially (this is the default).

The simplest way to expand and collapse folds is to click the fold markers in the gutter to the left of the text area; a fold marker is drawn next to the first line of each fold. An empty square is drawn next to an expanded fold; a filled square next to a collapsed fold. Unless the **Shift** key is held down, clicking a filled square will expand the fold by one level only; nested folds will remain collapsed. Holding down **Shift** while clicking will fully expand the fold and all nested folds.

View>Folding>Collapse Fold (keyboard shortcut: **Alt-Backspace**) collapses the fold containing the caret.

View>Folding>Expand Fold One Level (keyboard shortcut: **Alt**-**Enter**) expands the fold containing the caret. Nested folds will remain collapsed.

View>Folding>Expand Fold Fully (keyboard shortcut: **Alt**-**Shift**-**Enter**) expands the fold containing the caret, also expanding any nested folds.

View>Folding>Expand All Folds (keyboard shortcut: **Control**-**E Enter** ***key***) reads the next character entered at the keyboard, and expands all folds in the buffer with a fold level less than that specified, and collapsed all others.

View>Folding>Expand All Folds (keyboard shortcut: **Control**-**E X**) expands all folds in the buffer.

View>Folding>Select Fold (keyboard shortcut: **Control**-**E S**) selects all lines in the fold containing the caret. Control-clicking on a fold marker in the gutter on the left of the text area has the same effect.

Because folding is based on indent levels, changing the leading indent of a line while folds are collapsed may result in portions of the buffer becoming temporarily inaccessable. In such a case, simply invoke Expand All Folds to restore the visibility of the hidden lines.

The text `fold` is shown in black the status bar if portions of the buffer are invisible due to folding. Otherwise, it is grayed out.

5.6.1. Narrowing

The narrowing feature hides all parts of the buffer except for one specified region. While that region appears to be all there is, the rest of the text is still in the buffer; just not visible. While it may seem unrelated to folding, both folding and narrowing are implemented using the same code internally.

View>Folding>Narrow Buffer to Selection (keyboard shortcut: **Control**-**E N**) hides all lines the buffer except those in the selection.

View>Folding>Expand All Folds (keyboard shortcut: **Control**-**E X**) will make visible any lines hidden by narrowing.

Chapter 6. Customizing jEdit

6.1. The Buffer Options Dialog Box

Utilities>Buffer Options displays a dialog box for changing editor settings on a per-buffer basis. Any changes made in this dialog box are lost after the buffer is closed.

The following settings may be changed here:

- The edit mode (see Section 5.1)
- The tab width (see Section 5.4)
- The indent width
- The wrap column (see Section 4.3)
- The line separator (see Section 3.4)
- If syntax highlighting should be enabled (see Section 5.1.2)
- If auto indent and soft tabs should be enabled (see Section 5.4)

The “Corresponding buffer-local properties” text field displays buffer-local properties that duplicate the current settings in the dialog box.

6.2. Buffer-Local Properties

Buffer-local properties provide an alternate way to change editor settings on a per-buffer basis. While changes made in the Buffer Options dialog box are lost after the buffer is closed, buffer-local properties take effect each time the file is opened, because they are embedded in the file itself.

When jEdit loads a file, it checks the first 10 lines for colon-enclosed name/value pairs. The following example changes the indent width to 4 characters, enables soft tabs, and sets the buffer’s edit mode to Perl:

```
:indentSize=4:noTabs=true:mode=perl:
```

Note that adding buffer-local properties to a buffer only takes effect after the next time the buffer is loaded.

The following table describes each buffer-local property in detail.

Property name	**Description**
collapseFolds	Folds with a level of this or higher will be collapsed when the buffer is opened. If set to zero, all folds will be expanded initially. See Section 5.6.
commentEnd	The range comment end string. For example, in Java mode the default value is “*/”. See Section 5.5.
commentStart	The range comment start string. For example, in Java mode the default value is “/*”. See Section 5.5.
indentOnEnter	If set to “true”, pressing **Enter will insert a line break and automatically indent the new line. See Section 5.4.**
indentOnTab	If set to “true”, indentation will be performed when the **Tab key is pressed. See Section 5.4.**
indentSize	The width, in characters, of one indent. Must be an integer greater than 0. See Section 5.4.
lineComment	The line comment string. For example, in Java mode the default value is “//”. See Section 5.5.
maxLineLen	The maximum line length and wrap column position. Inserting text beyond this column will automatically insert a line break at the appropriate position. See Section 4.3.
mode	The default edit mode for the buffer. See Section 5.1.
noTabs	If set to “true”, soft tabs (multiple space characters) will be used instead of “real” tabs. See Section 5.4.
noWordSep	A list of non-alphanumeric characters that are *not to be treated as word separators.*

Property name	**Description**
syntax	If set to “false”, syntax highlighting will be not be performed. See Section 5.1.2.
tabSize	The tab width. Must be an integer greater than 0. See Section 5.4.
wordBreakChars	Characters, in addition to spaces and tabs, at which lines may be split when word wrapping. See Section 4.3.

6.3. The Global Options Dialog Box

Utilities>Global Options displays the global options dialog box. The dialog box is divided into several panes, each pane containing a set of related options. Use the list on the left of the dialog box to switch between panes. Only panes created by jEdit are described here; some plugins add their own option panes, and information about them can be found in the documentation for the plugins in question.

The General Pane

The General option pane lets you change various miscellaneous settings, such as the number of recent files to remember, the Swing look & feel, and such.

The Loading and Saving Pane

The Loading and Saving option pane lets you change settings such as the autosave frequency, backup settings, file encoding, and so on.

The Editing Pane

The Editing option pane lets you change settings such as the tab size, syntax highlighting and soft tabs on a global basis.

Due to the design of jEdit’s properties implementation, changes to some settings in this option pane only take effect in subsequently opened files.

The Mode-Specific Pane

The Mode-Specific option pane lets you change settings such as the tab size, syntax highlighting and soft tabs on a mode-specific basis.

The File name glob and First line glob text fields let you specify a glob pattern that names and first lines of buffers will be matched against to determine the edit mode.

This option pane does not change XML mode definition files on disk; it merely writes values to the user properties file which override those in mode files. To find out how to edit mode files directly, see Part II in *jEdit 3.2 User's Guide*.

The Text Area Pane

The Text Area option pane lets you customize the appearance of the text area.

The Gutter Pane

The Gutter option pane lets you customize the appearance of the gutter.

The Colors Pane

The Colors option pane lets you change the text area's color scheme.

The Styles Pane

The Styles option pane lets you change the text styles and colors used for syntax highlighting.

The Docking Pane

The Docking option pane lets you specify which dockable windows should be floating, and which should be docked in the view.

The Context Menu Pane

The Context Menu option pane lets you edit the text area's right-click context menu.

The Tool Bar Pane

The Tool Bar option pane lets you edit the tool bar, or disable it completely.

The Shortcuts Pane

The Shortcuts option pane let you change keyboard shortcuts. Each command can have up to two shortcuts associated with it.

The combo box at the top of the option pane selects the shortcut set to edit (command, plugin or macro shortcuts).

To change a shortcut, click the appropriate table entry and press the keys you want associated with that command in the resulting dialog box. The dialog box will warn you if the shortcut is already assigned.

The Abbreviations Pane

The Abbreviations option pane lets you enable or disable automatic abbreviation expansion, and edit currently defined abbreviations.

The combo box labelled “Abbrev set” selects the abbreviation set to edit. The first entry, “global”, contains abbreviations available in all edit modes. The subsequent entries contain mode-specific abbreviations.

To change an abbreviation expansion, click the appropriate table entry, which will display a dialog box for doing so.

To add an abbreviation, enter it in the last line of the list, which is always blank. When the last line is changed, a new, blank, line is added.

See Section 5.2.1 for information about positional parameters in abbreviations.

The Printing Pane

The Printing option pane lets you customize the appearance of printed output.

The File System Browser Pane

The File System Browser option pane lets you customize jEdit’s file system browser.

6.4. The jEdit Settings Directory

jEdit stores all settings, macros, and so on as files inside its *settings directory*. In most cases, editing these files is not necessary, since graphical tools and commands can do the job. However, being familiar with the structure of the settings directory still comes in handy in certain situations, for example when you want to copy jEdit settings between computers.

The location of the settings directory is system-specific; it is printed to the activity log (Utilities>Activity Log). For example:

```
[message] jEdit: Settings directory is /home/slava/.jedit
```

Specifying the **-settings** switch on the command line instructs jEdit to store settings in a different directory. For example, the following command will instruct jEdit to store all settings in the `jedit` subdirectory of the `C:` drive:

```
C:\jedit> jedit -settings=C:\jedit
```

The **-nosettings** switch will force jEdit to not look for or create a settings directory. Default settings will be used instead.

If you are using jEditLauncher to start jEdit on Windows, these parameters cannot be specified on the MS-DOS prompt command line when starting jedit; they must be set as described in Section G.2.

jEdit creates the following files and directories inside the settings directory; plugins may add more:

- `jars` - this directory contains plugins. See Chapter 8.
- `macros` - this directory contains macros. See Chapter 7.
- `modes` - this directory contains custom edit modes. See Part II in *jEdit 3.2 User's Guide*.
- `PluginManager.download` - this directory is usually empty. It only contains files while the plugin manager is downloading a plugin. For information about the plugin manager, see Chapter 8.
- `session` - a list of files, used when restoring previously open files on startup.

- `abbrevs` - a plain text file which stores all defined abbreviations. See Section 5.2.
- `activity.log` - a plain text file which contains the full activity log. See Appendix B.
- `history` - a plain text file which stores history lists, used by history text fields and the Edit>Paste Previous command. See Section 4.9 and Appendix C.
- `properties` - a plain text file which stores the majority of jEdit's settings.
- `recent.xml` - an XML file which stores the list of recently opened files. jEdit remembers the caret position, selection state and character encoding of each recent file, and automatically restores those values when one of the files in the list is opened.
- `server` - a plain text file that only exists while jEdit is running. The edit server's port number and authorization key is stored here. See Chapter 1.

Chapter 7. Using Macros

Macros in jEdit are short scripts written in a scripting language called *BeanShell*. They provide an wasy way to automate repetitive keyboard and menu procedures, as well as access to the objects and methods created by jEdit. Macros also provide a powerful facility for customizing jEdit and automating complex text processing and programming tasks. In this section we describe how to record and run macros. A detailed guide on writing macros appears later in a separate part of the user's guide; see Part III in *jEdit 3.2 User's Guide*.

7.1. Recording Macros

The simplest use of macros is to record a series of key strokes and menu commands as a BeanShell script, and play them back at a later time. While this doesn't let you take advantage of the full power of BeanShell, it is still a great time saver and can even be used to "prototype" more complicated macros.

Macros>Record Macro (shortcut: **Control**-**M** **Control**-**R**) prompts for a macro name and begins recording.

While recording is in progress, the string "Macro recording" is displayed in the status bar. jEdit records the following:

- Key strokes
- Menu item commands
- Tool bar clicks
- All search and replace operations except incremental search

Mouse clicks in the text area are *not* recorded; to record the equivalent of mouse operations, use the text selection commands or arrow keys.

Macros>Stop Recording (shortcut: **Control**-**M** **Control**-**S**) stops recording. It also switches to the buffer containing the recorded macro, giving you a chance to check over the recorded commands and make any necessary changes. The file name extension `.bsh` is automatically appended to the macro name, and all spaces are converted to underscore characters, in order to make the macro name a valid file name. These two operations are reversed when macros are displayed in the Macros menu. See Section 7.3 for details. When you are happy with the macro, save the buffer and it will appear in the Macros menu. To discard the macro, close the buffer without saving.

If a complicated operation only needs to be repeated a few of times, using the temporary macro feature is quicker than saving a new macro file.

Macros>Record Temporary Macro (shortcut: **Control-M Control-M**) begins recording to a buffer named `Temporary_Macro.bsh`. Once recording is complete, you don't need to save the `Temporary_Macro.bsh`buffer before playing it back.

Macros>Run Temporary Macro (shortcut: **Control-M Control-P**) plays the macro recorded to the `Temporary_Macro.bsh` buffer.

If you do not save the temporary macro, you must keep the buffer containing the macro script open during your jEdit session. To have the macro available for your next jEdit session, save the buffer `Temporary_Macro.bsh` as an ordinary macro with a descriptive name of your choice. The new name will then be displayed in the Macros menu.

7.2. Running Macros

Macros supplied with jEdit, as well as macros that you record or write, are displayed under the Macros menu in a hierarchical structure. The jEdit installation includes about 50 macros divided into several major categories. Each category corresponds to a nested submenu under the Macros menu. An index of these macros containing short descriptions and usage notes is found in Appendix F.

To run a macro, choose the Macros menu, navigate through the hierarchy of submenus, and select the name of the macro to execute. You can also assign execution of a particular macro to a keyboard shortcut, toolbar button or context menu using the Macro Shortcuts, Tool Bar or Context Menu panes of the Utilities>Global Options dialog; see Section 6.3.

Macros>Run Last Macro (shortcut: **Control-M Control-L**) runs the last macro run by jEdit again.

XInsert plugin

The XInsert plugin has a feature that lists the title of macros, organized by subdirectories, as part of its tree list display. Clicking on the leaf of the tree corresponding to a macro name causes jEdit to execute the macro immediately. The plugin allows you to keep a list of macros and cut-and-paste text fragments available while editing without opening menus. For information about installing plugins, see Chapter 8.

7.3. How jEdit Organizes Macros

Every macro, whether or not you originally recorded it, is stored on disk and can be edited as a text file. The file names of macros must have a `.bsh` extension. By default, jEdit associates a `.bsh` file with the BeanShell edit mode for purposes of syntax highlighting, indentation and other formatting. However, BeanShell syntax does not impose any indentation or line break requirements.

The Macros menu lists all macros stored in two places: the `macros` subdirectory of the jEdit install directory, and the `macros` subdirectory of the user-specific settings directory (see Section 6.4 for information about the settings directory). Any macros you record will be stored in the user-specific directory.

Macros stored elsewhere can be run using the Macros>Run Other Macro command, which displays a file chooser dialog box, and runs the specified file.

The listing of individual macros in the Macros menu can be organized in a hierarchy using subdirectories in the general or user-specific macro directories; each subdirectory appears as a submenu. You will find such a hierarchy in the default macro set included with jEdit.

When jEdit first loads, it scans the designated macro directories and assembles a listing of individual macros in the Macros menu. When scanning the names, jEdit will delete underscore characters and the `.bsh` extension for menu labels, so that `List_Useful_Information.bsh`, for example, will be displayed in the Macros menu as List Useful Information.

Macros>Browse System Macros displays the `macros` subdirectory of the directory in which jEdit is installed in the file system browser.

Macros>Browse User Macros displays the `macros` subdirectory of the user settings directory in the file system browser.

Macros can be opened and edited much like ordinary files from the file system browser. Editing macros from within jEdit will automatically update the macros menu; however, if you modify macros from another program, the Macros>Rescan Macros will need to be invoked to update the macro list.

Chapter 8. Installing and Using Plugins

A *plugin* is an application which is loaded and runs as part of another, host application. Plugins respond to user commands and perform tasks that supplement the host application's features.

This chapter covers installing, updating and removing plugins. Documentation for the plugins themselves can be found in Help>jEdit Help, and information about writing plugins can be found in Part IV in *jEdit 3.2 User's Guide*.

8.1. The Plugin Manager

Plugins>Plugin Manager displays the plugin manager window. The plugin manager lists all installed plugins; clicking on a plugin in the list will display information about it.

To remove plugins, select them (multiple plugins can be selected by holding down **Control**) and click Remove Plugins. This will display a confirmation dialog box first.

8.2. Installing Plugins

Plugins can be installed in two ways; manually, and from the plugin manager. In most cases, plugins should be installed from the plugin manager. It is easier and more convinient.

To install plugins manually, go to http://plugins.jedit.org in a web browser and follow the directions on that page.

To install plugins from the plugin manager, make sure you are connected to the Internet and click the Install Plugins button in the plugin manager window. The plugin manager will then download information about available plugins from the jEdit web site[1] and list those not already installed in the Install Plugins dialog box. Only plugins compatible with your jEdit release will be shown, and installing a plugin will also automatically install any other plugins it depends on. As a result of this, if you use the plugin manager to install plugins, it is very hard to end up with a non-working set of plugins.

Click on a plugin in the list to see some information about it. To select plugins to be installed, click the check box next to their names in the list.

The Install source code check box controls if source code for the plugins should be downloaded and installed. Unless you are a developer, you probably don't need the source.

The two radio buttons select the location where the plugins are to be installed. Plugins can be installed in either the `jars` subdirectory of the jEdit installation directory, or the `jars` subdirectory of the user-specific settings directory. For information about the settings directory, Section 6.4.

Once you have specified plugins to install, click Install Plugins to begin the download process. Once all plugins have been downloaded and installed, a dialog box is shown advising that jEdit must be restarted before plugins can be used.

Firewalls

If you are connected to the Internet through a firewall or proxy, you will need to configure firewall settings in the Plugin Options>Firewall pane of the Utilities>Global Options dialog box, otherwise the plugin manager might not be able to connect to the jEdit web site.

This assumes you chose to install the Firewall plugin when installing jEdit. This plugin requires Java 2.

8.3. Updating Plugins

Clicking Update Plugins in the plugin manager will show a dialog box very similar to the one for installing plugins. It will list plugins for which updated versions are available. It will also offer to delete any obsolete plugins.

Notes

1. The list of plugins is downloaded from http://plugins.jedit.org/export/new_plugin_manager.php, in XML format.

Appendix A. Keyboard Shortcuts

This appendix documents the default set of keyboard shortcuts. They can be customized to suit your taste in the Utilities>Global Options dialog box; see Section 6.3.

Files

For details, see Section 2.1, Section 2.2 and Chapter 3.

Control-N	New file.
Control-O	Open file.
Control-W	Close buffer.
Control-E Control-W	Close all buffers.
Control-S	Save buffer.
Control-E Control-S	Save all buffers.
Control-P	Print buffer.
Control-Page Up	Go to previous buffer.
Control-Page Down	Go to next buffer.
Control-‘	Go to recent buffer.
Control-Q	Exit jEdit.

Views

For details, see Section 2.2.

Control-E Control-T	Turn gutter (line numbering) on and off.
Control-2	Split view horizontally.
Control-3	Split view vertically.
Control-1	Unsplit.

Alt-Page Up	Go to previous text area.
Alt-Page Down	Go to next text area.
Control-E 1; 2; 3; 4	Collapse/expand top; bottom; left; right docking area.

Repeating

For details, see Section 2.4.

Control-Enter *`number command`*	Repeat the command (it can be a keystroke, menu item selection or tool bar click) the specified number of times.

Moving the Caret

For details, see Section 4.1, Section 4.5, Section 4.6, Section 4.7 and Section 5.3.

`Arrow`	Move caret one character or line.
Control-*`Arrow`*	Move caret one word or paragraph.
Page Up; Page Down	Move caret one screenful.
Home	First non-whitespace character of line, beginning of line, first visible line (repeated presses).
End	Last non-whitespace character of line, end of line, last visible line (repeated presses).
Control-Home	Beginning of buffer.
Control-End	End of buffer.
Control-]	Go to matching bracket.
Control-E [;]	Go to previous; next bracket.
Control-L	Go to line.

Selecting Text

For details, see Section 4.2, Section 4.5, Section 4.6, Section 4.7 and Section 5.3.

Shift-*`Arrow`*	Extend selection by one character or line.
Control-Shift-*`Arrow`*	Extend selection by one word or paragraph.
Shift-Page Up; Shift-Page Down	Extend selection by one screenful.
Shift-Home	Extend selection to first non-whitespace character of line, beginning of line, first visible line (repeated presses).
Shift-End	Extend selection to last non-whitespace character of line, end of line, last visible line (repeated presses).
Control-Shift-Home	Extend selection to beginning of buffer.
Control-Shift-End	Extend selection to end of buffer.
Control-[	Select code block.
Control-E W; L; P	Select word; line; paragraph.
Control-E Control-L	Select line range.
Control-\	Switch between single and multiple selection mode.

Scrolling

For details, see Section 2.2.

Control-E Control-J	Center current line on screen.
Control-E Control-I	Center caret on screen.
Control-’; Control-/	Scroll up; down one line.
Alt-’; Alt-/	Scroll up; down one page.

Text Editing

For details, see Section 4.4, Section 4.3, Section 4.5, Section 4.6 and Section 4.7.

Control-Z	Undo.
Control-E Control-Z	Redo.
Backspace; Delete	Delete character before; after caret.
Control-Backspace; Control-Delete	Delete word before; after caret.
Control-D; Control-E D	Delete line; paragraph.
Shift-Backspace; Shift-Delete	Delete from caret to beginning; end of line.
Control-E R	Remove trailing whitespace from the current line (or all selected lines).
Control-J	Join lines.
Control-B	Complete word.
Control-E F	Format paragraph (or selection).

Clipboard and Registers

For details, see Section 4.9.

Control-X	Cut selected text to clipboard.
Control-C	Copy selected text to clipboard.
Control-E Control-U	Append selected text to clipboard, removing it from the buffer.
Control-E Control-A	Append selected text to clipboard, leaving it in the buffer.
Control-V	Paste clipboard contents.
Control-R Control-X *key*	Cut selected text to register *key*.
Control-R Control-C *key*	Copy selected text to register *key*.

Control-R Control-U *key*	Append selected text to register *key, removing it from the buffer.*
Control-R Control-A *key*	Append selected text to register *key, leaving it in the buffer.*
Control-R Control-V *key*	Paste contents of register *key.*
Control-E Control-V	Paste previous.

Markers

For details, see Section 4.10.

Control-E Control-M	If current line doesn't contain a marker, one will be added. Otherwise, the existing marker will be removed. Use the Markers menu to return to markers added in this manner.
Control-T *key*	Add marker with shortcut *key.*
Control-Y *key*	Go to marker with shortcut *key.*
Control-U *key*	Select to marker with shortcut *key.*
Control-K *key*	Go to marker with shortcut *key, and move the marker to the previous caret position.*
Alt-Up; Alt-Down	Move caret to previous; next marker.

Search and Replace

For details, see Section 4.11.

Control-F	Open search and replace dialog box.
Control-G	Find next.

Control-H	Find previous.
Control-E Control-F	Find selection.
Control-E Control-B	Search in open buffers.
Control-E Control-D	Search in directory.
Control-E Control-R	Replace in selection.
Control-E Control-G	Replace in selection and find next.
Control-,	Quick incremental search.
Control-.	Quick HyperSearch.

Source Code Editing

For details, see Section 5.2, Section 5.4 and Section 5.5.

Control-;	Expand abbreviation.
Alt-Left; Alt-Right	Shift current line (or all selected lines) left; right.
Control-I	Indent current line (or all selected lines).
Control-E Control-C	Wing comment selection.
Control-E Control-B	Box comment selection.
Control-E Control-K	Block comment selection.

Folding and Narrowing

For details, see Section 5.6 and Section 5.6.1.

Alt-Backspace	Collapse fold containing caret.
Alt-Enter	Expand fold containing caret one level only.
Alt-Shift-Enter	Expand fold containing caret fully.

Control-E Enter `key`	Expand folds with level less than `key, collapse all others.`
Control-E X	Expand all folds.
Control-E S	Select fold.
Control-E N	Narrow to selection.

Macros

For details, see Chapter 7.

Control-M Control-R	Record macro.
Control-M Control-M	Record temporary macro.
Control-M Control-S	Stop recording.
Control-M Control-P	Run temporary macro.
Control-M Control-L	Run most recently played or recorded macro.

Appendix B. The Activity Log

The *activity log* is very useful for troubleshooting problems, and helps when developing plugins.

Utilities>Activity Log displays the last 500 lines of the activity log. By default, the activity is shown in a floating window. It can be set to dock into the view in the Docking pane of the Utilities>Global Options dialog box; see Section 2.2.1. The complete log can be found in the `activity.log` file inside the jEdit settings directory, the path of which is shown inside the activity log window.

jEdit writes the following information to the activity log:

- Information about your Java implementation (version, operating system, architecture, etc)
- All error messages and runtime exceptions (most errors are shown in dialog boxes as well; but the activity log usually contains more detailed and technical information)
- All sorts of debugging information that can be helpful when tracking down bugs
- Information about loaded plugins

While jEdit is running, the log file on disk may not always accurately reflect what has been logged, due to buffering being done for performance reasons. To ensure the file on disk is up to date, invoke the Utilities>Update Activity Log on Disk command. The log file is also automatically updated on disk when jEdit exits.

Appendix C. History Text Fields

The text fields in the search and replace dialog box and file system browser remember the last 20 entered strings by default. The number of strings to remember can be changed in the General pane of the Utilities>Global Options dialog box; see Section 6.3.

Pressing **Up** recalls previous strings. Pressing **Down** after recalling previous strings recalls later strings.

Pressing **Shift-Up** or **Shift-Down** will search backwards or forwards, respectively, for strings beginning with the text already entered in the text field.

Clicking the triangle to the right of the text field, or clicking with the right-mouse button anywhere else will display a pop-up menu of all previously entered strings; selecting one will input it into the text field. Holding down **Shift** while clicking will display a menu of all previously entered strings that begin with the text already entered.

Appendix D. Glob Patterns

jEdit uses glob patterns similar to those in the various Unix shells to implement file name filters in the file system browser. Glob patterns resemble regular expressions somewhat, but have a much simpler syntax. The following character sequences have special meaning within a glob pattern:

- `?` matches any one character
- `*` matches any number of characters
- `{a,b,c}` matches any one of *a*, *b* or *c*
- `[abc]` matches any character in the set *a*, *b* or *c*
- `[^abc]` matches any character not in the set *a*, *b* or *c*
- `[a-z]` matches any character in the range *a* to *z*, inclusive. A leading or trailing dash will be interpreted literally

Within a character class expression, the following sequences have special meaning:

- `[:alnum:]` Any alphanumeric character
- `[:alpha:]` Any alphabetical character
- `[:blank:]` A space or horizontal tab
- `[:cntrl:]` A control character
- `[:digit:]` A decimal digit
- `[:graph:]` A non-space, non-control character
- `[:lower:]` A lowercase letter
- `[:print:]` Same as `[:graph:]`, but also space and tab
- `[:punct:]` A punctuation character
- `[:space:]` Any whitespace character, including newlines

- `[:upper:]` An uppercase letter
- `[:xdigit:]` A valid hexadecimal digit

Here are some example glob patterns:

- **`*`** - all files
- **`*.java`** - all files whose names end with “.java”
- **`*.{c,h}`** - all files whose names end with either “.c” or “.h”
- **`*[^~]`** - all files whose names do not end with “~”

Appendix E. Regular Expressions

jEdit uses regular expressions to implement inexact search and replace. A regular expression consists of a string where some characters are given special meaning with regard to pattern matching.

Within a regular expression, the following characters have special meaning:

Positional Operators

- `^` matches at the beginning of a line
- `$` matches at the end of a line
- `\b` matches at a word break
- `\B` matches at a non-word break
- `\<` matches at the start of a word
- `\>` matches at the end of a word

One-Character Operators

- `.` matches any single character
- `\d` matches any decimal digit
- `\D` matches any non-digit
- `\n` matches the newline character
- `\s` matches any whitespace character
- `\S` matches any non-whitespace character
- `\t` matches a horizontal tab character
- `\w` matches any word (alphanumeric) character
- `\W` matches any non-word (alphanumeric) character

- `\\` matches the backslash ("\") character

Character Class Operator

- `[abc]` matches any character in the set *a*, *b* or *c*
- `[^abc]` matches any character not in the set *a*, *b* or *c*
- `[a-z]` matches any character in the range *a* to *z*, inclusive. A leading or trailing dash will be interpreted literally

Within a character class expression, the following sequences have special meaning:

- `[:alnum:]` Any alphanumeric character
- `[:alpha:]` Any alphabetical character
- `[:blank:]` A space or horizontal tab
- `[:cntrl:]` A control character
- `[:digit:]` A decimal digit
- `[:graph:]` A non-space, non-control character
- `[:lower:]` A lowercase letter
- `[:print:]` Same as `[:graph:]`, but also space and tab
- `[:punct:]` A punctuation character
- `[:space:]` Any whitespace character, including newlines
- `[:upper:]` An uppercase letter
- `[:xdigit:]` A valid hexadecimal digit

Subexpressions and Backreferences

- `(abc)` matches whatever the expression *abc* would match, and saves it as a subexpression. Also used for grouping

- `(?:...)` pure grouping operator, does not save contents
- `(?#...)` embedded comment, ignored by engine
- `(?=...)` positive lookahead; the regular expression will match if the text in the brackets matches, but that text will not be considered part of the match
- `(?!...)` negative lookahead; the regular expression will match if the text in the brackets does not match, and that text will not be considered part of the match
- `\`*n* where 0 < *n* < 10, matches the same thing the *n*th subexpression matched. Can only be used in the search string
- `$`*n* where 0 < *n* < 10, substituted with the text matched by the *n*th subexpression. Can only be used in the replacement string

Branching (Alternation) Operator

- *a*`|`*b* matches whatever the expression *a* would match, or whatever the expression *b* would match.

Repeating Operators

These symbols operate on the previous atomic expression.

- `?` matches the preceding expression or the null string
- `*` matches the null string or any number of repetitions of the preceding expression
- `+` matches one or more repetitions of the preceding expression
- `{`*m*`}` matches exactly *m* repetitions of the one-character expression
- `{`*m*`,`*n*`}` matches between *m* and *n* repetitions of the preceding expression, inclusive
- `{`*m*`,}` matches *m* or more repetitions of the preceding expression

Stingy (Minimal) Matching

If a repeating operator (above) is immediately followed by a `?`, the repeating operator will stop at the smallest number of repetitions that can complete the rest of the match.

Appendix F. Macros Included With jEdit

jEdit comes with a large number of sample macros that perform a variety of tasks. The following index provides short descriptions of each macro, in some cases accompanied by usage notes.

F.1. File Management Macros

These macros automate the opening and closing of files.

- `Browse_Directory.bsh`

  Opens a directory supplied by the user in the file system browser.

- `Close_Except_Active.bsh`

  Closes all files except the current buffer.

  Prompts the user to save any buffer containing unsaved changes.

- `Go_to_File_System_Browser.bsh`

  Sets the input focus to the file system browser.

- `Open_Path.bsh`

  Opens the file supplied by the user in an input dialog.

- `Open_Selection.bsh`

  Opens the file named by the current buffer's selected text.

F.2. Text Macros

These macros generate various forms of formatted text.

- `Add_Prefix_and_Suffix.bsh`

  Adds user-supplied “prefix” and “suffix” text to each line in a group of selected lines.

  Text is added after leading whitespace and before trailing whitespace. A dialog window receives input and “remembers” past entries.

- `Color_Picker.bsh`

  Displays a color picker and inserts the selected color in hexadecimal format, prefixed with a “#”.

- `Duplicate_Line.bsh`

  Duplicates the line on which the caret lies immediately beneath it and moves the caret to the new line.

- `Insert_Date.bsh`

  Inserts the current date and time in the current buffer.

  The inserted text includes a representation of the time in the “Internet Time” format.

- `Insert_Tag.bsh`

  Inserts a balanced pair of markup tags as supplied in a input dialog.

- `Toggle_Line_Comment.bsh`

  Toggles line comments, alternately inserting and deleting them at the beginning of each selected line.

  If there is no selection, the macro operates on the current line.

- `Make_Double_Box_Comments.bsh`

Makes a individual wing style comment of equal width for each selected line in the current buffer.

```
/*  This is an example of the kind                */
/*  of comment (for Java or C/C++) produced       */
/*  by this macro. It has uniform width           */
/*  regardless of the width of the several lines. */

<!- HTML or SGML code                          ->
<!- will look like this when the macro is run ->
```

- `Reverse.bsh`

  Reverses the selected text in the current buffer.

- `Rot13.bsh`

  Replaces the selected text with the text encoded by the Rot13 “encryption” algorithm.

  Rot13 is a simple encoding scheme involving fixed character substitution. A second application of the algorithm restores the original text.

- `Write_File_Header.bsh`

  Writes a formatted file header in the current buffer based upon user input.

  This macro asks for the name of the file, the author and a brief description of its contents. It also asks whether the file should be saved immediately after the header is inserted. The header will be set off with block comments based upon the editing mode of the buffer; if the user has not set an editing mode, the macro will select one based upon the file extension.

  > **Note:** The notes accompanying the macro source code describe how the macro can be modified to produce a file header conforming to to personal taste or institutional requirements.

F.3. Java Code Macros

These macros handle text formatting and generation tasks that are particularly useful in writing Java code.

- `Get_Class_Name.bsh`

  Inserts a Java class name based upon the buffer's file name.

- `Get_Package_Name.bsh`

  Inserts a plausible Java package name for the current buffer.

  The macro compares the buffer's path name with the elements of the classpath being used by the jEdit session. An error message will be displayed if no suitable package name is found. This macro will not work if jEdit is being run as a JAR file without specifying a classpath. In that case the classpath seen by the macro consists solely of the JAR file.

- `Make_Get_and_Set_Methods.bsh`

  Creates `getXXX()` or `setXXX()` methods that can be pasted into the buffer text.

  This macro presents a dialog that will "grab" the names of instance variables from the caret line of the current buffer and paste a corresponding `getXXX()` or `setXXX()` method to one of two text areas in the dialog. The text can be edited in the dialog and then pasted into the current buffer using the Insert... buttons. If the caret is set to a line containing something other than an instance variable, the text grabbing routine is likely to generate nonsense.

  As explained in the notes accompanying the source code, the macro uses a global variable which can be set to configure the macro to work with either Java or C++ code. When set for use with C++ code, the macro will also write (in commented text) definitions of `getXXX()` or `setXXX()` suitable for inclusion in a header file.

- `Tidy_Block_Comments.bsh`

  Formats all end-of-line "block" comments to begin at a fixed column.

This macro uses jEdit's syntax parsing routines to identify block comments and place them in a column specified by the user. If uncommented text extends beyond the specified column, the block comment will be placed two columns after the end of the uncommented text with an intervening whitespace.

An input dialog allows the user to specify the display column for block comments or to accept a default value. The user can also select whether tabs will be substituted for spaces and whether comments at the beginning of a line will be ignored. The macro will complain if the current buffer's editing mode does not support block comments.

F.4. Search Macros

These macros provide various shortcuts to search methods. A group of macros in this category allow the user to search of other occurrences of the word that appear on or next to the editing caret.

- `Find_Matching_File.bsh`

  Switches between C++ header (`.h`) and source (`.cpp`) files with the same name in the same directory.

  > **Note:** This macro is easily adapted to work with any pair of file extensions.

- `Next_Char.bsh`

  Finds next occurence of character on current line.

  The macro takes the next character typed after macro execution as the character being searched. That character is not displayed. If the character does not appear in the balance of the current line, no action occurs.

  This macro illustrates the use of `InputHandler.readNextChar()` as a means of obtaining user input. See Section 14.1.4.

- `Write_HyperSearch_Results.bsh`

This macro writes the contents of the “HyperSearch Results” window to a new text buffer.

The macro employs a simple text report format. Since the HyperSearch window’s object does not maintain the search settings that produced the displayed results, the macro examines the current settings in the `SearchAndReplace` object. It confirms that the HyperSearch option is selected before writing the report. However, the only way to be sure that the report’s contents are completely accurate is to run the macro immediately after a HyperSearch.

F.4.1. The Find_Occurrence Macro Group

This is a group of macros that enable searches in a text buffer for another occurrence of the word situated at or immediately to the left of the editing caret. When these macros are linked to keyboard shortcuts, they give the user the ability to search for occurrences of a word without leaving the text buffer or interrupting use of the keyboard.

Because the searching routine for each procedure has common code, the set of macros consists of four macros that set a temporary jEdit property and then call the main search macro, `Find_Occurrence.bsh`. That macro reads the temporary property, executes the corresponding search procedure, and erases the property. If the property cannot be found, the search routine looks for the next succeeding occurrence of the search term.

The final macro retrieves the marker left by the searching macro for the file and caret position applicable just prior to the search.

- `Find_Occurrence.bsh`

  This macro runs the search routine corresponding to the property set by one of its companion macros.

  If the macro is called directly or if the search type property cannot be found, it will find the next occurrence of the word on or to the left of the editing caret. If the search succeeds, the macro sets a bookmark by creating temporary jEdit properties for the buffer name and caret location.

- `Find_First_Occurrence.bsh`

  Calls `Find_Occurrence` to find the first occurrence of the word on or to the left of the editing caret.

- `Find_Previous_Occurrence.bsh`

  Calls `Find_Occurrence` to find the immediately preceding occurrence of the word on or to the left of the editing caret.

- `Find_Next_Occurrence.bsh`

  Calls `Find_Occurrence` to find the next occurrence of the word on or to the left of the editing caret.

- `Find_Last_Occurrence.bsh`

  Calls `Find_Occurrence` to find the last occurrence of the word on or to the left of the editing caret.

- `Return_From_Find.bsh`

  Returns the user to the buffer and location specified in the bookmark created by `Find_Occurrence`, reopening a file if necessary.

  The file is reopened if necessary; an error message is displayed if the file no longer exists. If the file exists but the caret position index exceeds the size of the file (because of intervening deletions, for example), the file is displayed and an error message alerts the user that the bookmarked caret position is invalid. The bookmark is deleted immediately after it is used.

F.5. Macros for Listing Properties

These macros produce lists or tables containing properties used by the Java platform or jEdit itself.

- `jEdit_Properties.bsh`

  Writes an unsorted list of jEdit properties in a new buffer.

- `System_Properties.bsh`

  Writes an unsorted list of all Java system properties in a new buffer.

- `Look_and_Feel_Properties.bsh`

  Writes an unsorted list of the names of Java Look and Feel properties in a new buffer.

F.6. Miscellaneous Macros

While these macros do not fit easily into the other categories, they all provide interesting and useful functions.

- `Cascade_jEdit_Windows.bsh`

  Rearranges view and floating plugin windows.

  The windows are arranged in an overlapping “cascade” pattern beginning near the upper left corner of the display.

- `Copy_Mode_Abbrevs.bsh`

  Copies all abbreviations from one editing mode to another, overwriting any duplicate entries.

  A number of jEdit editing modes target languages that share keywords, tags or other features. Examples include “java” and “beanshell”, and “c” and “c++”. This macro saves the trouble of manually editing abbreviations sets to share abbreviations between editing modes. The macro will also permit copying of a mode’s abbreviations to the “global” abbreviation set that is available in all buffers regardless of editing mode.

  The macro will overwrite any existing abbreviations in the target editing mode using the same abbreviation as a member of the source set. Use caution in copying from one set to another, as any attempt to undo the copying must be done manually.

- `Display_Abbreviations.bsh`

  Displays the abbreviations registered for each of jEdit’s editing modes.

  The macro provides a read-only view of the abbreviations contained in the “Abbreviations” option pane. Pressing a letter key will scroll the table to the first entry beginning with that letter. A further option is provided to write a selected mode’s abbreviations or all abbreviations in a text buffer for printing as a reference. Notes in the source code listing point out some display options that are configured by modifying global variables.

- `Display_Shortcuts.bsh`

  Displays a sorted list of the keyboard shortcuts currently in effect.

  The macro provides a combined read-only view of command, macro and plugin shortcuts. Pressing a letter key will scroll the table to the first entry beginning with that letter. A further option is provided to write the shortcut assignments in a text buffer for printing as a reference. Notes in the source code listing point out some display options that are configured by modifying global variables.

- `Evaluate_Buffer_in_BeanShell.bsh`

  Evaluates contents of current buffer as a BeanShell script, and opens a new buffer to receive any text output.

  This is a quick way to test a macro script even before its text is saved to a file. Opening a new buffer for output is a precaution to prevent the macro from inadvertently erasing or overwriting itself. BeanShell scripts that operate on the contents of the current buffer will not work meaningfully when tested using this macro.

- `Go_to_Text_Area.bsh`

  Sets the input focus to the text editing area.

  Linked to a keyboard shortcut, this macro can quickly return input focus to the text area after invoking a command that shifts focus to a docked plugin window.

- `Include_Guard.bsh`

Intended for C/C++ header files, this macro inserts a preprocessor directive in the current buffer to ensure that the header is included only once per compilation unit.

To use the macro, first place the caret at the beginning of the header file before any uncommented text. The macro will return to this position upon completion. The defined term that triggers the “include guard” is taken from the buffer’s name.

- `List_Plugin_Internal_Names.bsh`

  Writes a sorted list of installed plugins to the current buffer.

  The form of each name is that used by `jEdit.getPlugin()`.

  Tip: The name can be used in a macro to test for the presence of a particular plugin.

- `Make_Bug_Report.bsh`

  Creates a new buffer with installation and error information extracted from the activity log.

  The macro extracts initial messages written to the activity log describing the user’s operating system, JDK, jEdit version and installed plugins. It then appends the last set of error messages written to the activity log. The new text buffer can be saved and attached to an email message or a bug report made on SourceForge.

- `Run_Macro_at_Caret.bsh`

  Executes the macro whose name appears at the editing caret.

  When used with abbreviations for macro name, this macro allows the user to execute any macro script from the keyboard by typing its name, without the `.bsh` extension. It will search for the requested script in both the system and user macro directories, in each case using the caret text as a relative path.

  The full utility of this macro can be acheived when it is combined with abbreviations for commonly used macros. To try it out, follow these steps:

1. In the “Macro Shortcuts” option pane, Associate `Run_Macro_at_Caret` with the shortcut **Control-Space**.
2. In the “global” abbreviation group, associate the abbreviation “dtt” with the text “/Text/Insert_Date”. The leading forward slash character is necessary and should be used regardless of one’s operating system. Make sure that the abbreviation option pane has the checkbox Space bar expands abbrevs selected.
3. To activate the macro from the keyboard, type **`dtt`** in a text buffer.
4. Press the space bar to expand **`ddt`** to **`/Text/Insert_Date`**
5. Press **Control-Space** to run the macro. The text **`/Text/Insert_Date`** will be replaced by the output of the `Insert_Date` macro.

Repeating this procedure allows the user to execute macros from the keyboard using shortcut names instead of keystrokes.

- `Show_Free_Memory.bsh`

  Runs the Java garbage collection routine to free unneeded memory.

  After running garbage collection, the macro displays a message box with text and graphic displays of jEdit’s memory usage after garbage collection.

Appendix G. jEditLauncher for Windows

G.1. Introduction

The jEditLauncher package is a set of lightweight components for running jEdit under the Windows group of operating systems. The package is designed to run on Windows 95, Windows 98, Windows Me, Windows NT (versions 4.0 and greater) and Windows 2000.

While jEdit does not make available a component-type interface, it does contains an “EditServer” that listens on a socket for requests to load scripts written in the BeanShell scripting language. When the server activates, it writes the server port number and a pseudo-random, numeric authorization key to a text file. By default, the file is named `server` and is located in the settings directory (see Section 6.4).

The jEditLauncher component locates and reads this file, opens a socket and attempts to connect to the indicated port. If successful, it transmits the appropriate BeanShell script to the server. If unsuccessful, it attempts to start jEdit and repeats the socket transmission once it can obtain the port and key information. The component will abandon the effort to connect roughly twenty seconds after it launches the application.

G.2. Starting jEdit

The main component of the jEditLauncher package is a client application entitled **jedit.exe**. It may be executed either from Windows Explorer, or the command line. It uses the jEditLauncher COM component to open files in jEdit that are listed as command line parameters. It supports Windows and UNC file specifications as well as wild cards. If called without parameters, it will launch jEdit. If jEdit is already running, it will simply open a new, empty buffer.

jedit.exe supports four command-line options. If any of these options are invoked correctly, the application will not load files or execute jEdit.

- The option **`/h`** causes a window to be displayed with a brief description of the application and its various options.
- The option **`/p`** will activate a dialog window displaying the command-line parameters to be used when calling jEdit. This option can also be triggered by selecting Set jEdit Parameters from the jEdit section of the Windows Programs menu.

  Using the dialog, you can change parameters specifying the executable for the Java interpreter, the JAR archive file or class name used as the target of the interpreter, and command line options for both. If the **`-jar`** option is not used for the Java executable, the principal jEdit class of `org.gjt.sp.jedit.jEdit` is set as fixed data. The working directory for the Java interpreter's process can also be specified. A read-only window at the bottom of the dialog displays the full command-line that jEditLauncher will invoke.

  Before committing changes to the command line parameters, **jedit.exe** validates the paths for the Java and jEdit targets as well as the working directory. It will complain if the paths are invalid. It will not validate command-line options, but it will warn you if it finds the **`-noserver`** option used for jEdit, since this will deactivate the edit server and make it impossible for jEditLauncher to open files.

  Note that due to the design of jEditLauncher, platform-independent command line options handled by jEdit itself (such as **-background** and **-norestore**) must be entered in the "Set jEdit Parameters" dialog box, and cannot be specified on the **jedit.exe** command line directly. For information about platform-independent command line options, see Section 1.4.
- The option **`/i`** is not mentioned in the help window for `jedit.exe`. It is intended primarily to be used in conjunction with jEdit's Java installer, but it can also be used to install or reinstall jEditLauncher manually. When accompanied by a second parameter specifying the directory where your preferred Java interpreter is located, jEditLauncher will install itself and set a reasonable initial set of command line parameters for executing jEdit. You can change these parameters later by running `jedit.exe` with the**`/p`** option.
- The option **`/u`** will cause jEditLauncher to be uninstalled by removing its registry entries. This option does not delete any jEditLauncher or jEdit files.

G.3. The Context Menu Handler

The jEditLauncher package also implements a context menu handler for the Windows shell. It is intended to be be installed as a handler available for any file. When you right-click on a file or shortcut icon, the context menu that appears will include an item displaying the jEdit icon and captioned Open with jEdit. If the file has an extension, another item will appear captioned Open *.XXX with jEdit, where XXX is the extension of the selected file. Clicking this item will cause jEdit to load all files with the same extension in the same directory as the selected file. Multiple file selections are also supported; in this circumstance only the Open with jEdit item appears.

If a single file with a `.bsh` extension is selected, the menu will also contain an item captioned Run script in jEdit. Selecting this item will cause jEditLauncher to run the selected file as a BeanShell script.

If exactly two files are selected, the menu will contain an entry for Show diff in jEdit. Selecting this item will load the two files in jEdit and have them displayed side-by-side with their differences highlighted by the JDiff plugin. The file selected first will be treated as the base for comparison purposes. If the plugin is not installed, an error message will be displayed in jEdit. See Chapter 8 for more information and installing plugins.

G.4. Uninstalling jEdit and jEditLauncher

There are three ways to uninstall jEdit and jEditLauncher.

- First, you can run `unlaunch.exe` in the jEdit installation directory.
- Second, you can choose Uninstall jEdit from the jEdit section of the Programs menu.
- Third, you can choose the uninstall option for jEdit in the Control Panel's Add/Remove Programs applet.

Each of these options will deactivate jEditLauncher and delete all files in jEdit's installation directory. As a safeguard, jEditLauncher displays a warning window and requires the user to confirm an uninstall operation. Because the user's settings directory can be set and changed from one jEdit session to another, user settings files must be deleted manually.

To deactivate jEditLauncher without deleting any files, run `jedit /u` from any command prompt where the jEdit installation directory is in the search path. This will remove the entries for jEditLauncher from the Windows registry, and disable the context menu handler, and the automatic launching and scripting capabilities. The package can reactivated by executing **jedit.exe** again, and jEdit can be started in the same manner as any other Java application on your system.

G.5. The jEditLauncher Interface

The core of the jEditLauncher package is a COM component that can communicate with the EditServer, or open jEdit if it is not running or is otherwise refusing a connection. The component supports both Windows and UNC file specifications, including wild cards. It will resolve shortcut links to identify and transmit the name of the underlying file.

Because it is implemented with a “dual interface”, the functions of jEditLauncher are available to scripting languages in the Windows environment such as VBScript, JScript, Perl (using the Win32::OLE package) and Python (using the win32com.client package).

The scriptable interface consists of two read-only properties and six functions:

Properties

- `ServerPort` - a read-only property that returns the port number found in jEdit’s server file; the value is not tested for authenticity. It returns zero if the server information file cannot be located.
- `ServerKey` - a read-only property that returns the numeric authorization key found in jEdit’s server file; the value is not tested for authenticity. It returns zero if the server information file cannot be located.

Functions

- `OpenFile` - a method that takes a single file name (with or without wild card characters) as a parameter.

- `OpenFiles` - this method takes a array of file names (with or without wild card characters) as a parameter. The form of the array is that which is used for arrays in the scripting environment. In JScript, for example the data type of the VARIANT holding the array is VT_DISPATCH; in VBScript, it is VT_ARRAY | VT_VARIANT, with array members having data type VT_BSTR.
- `Launch` - this method with no parameters attempts to open jEdit without loading additional files.
- `RunScript` - this method takes a file name or full file path as a parameter and runs the referenced file as a BeanShell script in jEdit. The predefined variables `view`, `editPane`, `textArea` and `buffer` are available to the script. If more than one view is open, the variable are initialized with reference to the earliest opened view. If no path is given for the file it will use the working directory of the calling process.
- `EvalScript` - this method takes a string as a parameter containing one or more BeanShell statements and runs the script in jEdit's BeanShell interpreter. The predefined variables are available on the same basis as in `RunScript`.
- `RunDiff` - this method takes two strings representing file names as parameters. If the JDiff plugin is installed, this method will activate the JDiff plugin and display the two files in the plugin's graphical "dual diff" format. The first parameter is treated as the base for display purposes. If the JDiff plugin is not installed, a error message box will be displayed in jEdit.

G.6. Scripting Examples

Here are some brief examples of scripts using jEditLauncher. The first two will run under Windows Script Host, which is either installed or available for download for 32-bit Windows operating systems. The next example is written in Perl and requires the Win32::OLE package. The last is written in Python and requires the win32gui and win32com.client extensions.

If Windows Script Host is installed, you can run the first two scripts by typing the name of the file containing the script at a command prompt. In jEdit's Console plugin, you can type **`cmd /c`** ***`script_path`*** or **`wscript`** ***`script_path`***.

```
' Example VBSscript using jEditLauncher interface
dim launcher
set launcher = CreateObject("JEdit.JEditLauncher")
a = Array("I:\Source Code Files\shellext\jeditshell\*.h", _
"I:\Source Code Files\shellext\jeditshell\*.cpp")
MsgBox "The server authorization code is " + _
FormatNumber(launcher.ServerKey, 0, 0, 0, 0) + ".", _
vbOKOnly + vbInformation, "jEditLauncher"
launcher.openFiles(a)
myScript = "jEdit.newFile(view); textArea.setSelectedText(" _
  & CHR(34) _
  & "Welcome to jEditLauncher." _
  & CHR(34) & ");"
launcher.evalScript(myScript)
```

```
/* Example JScript using jEditLauncher interface
 * Note: in contrast to VBScript, JScript does not
 * directly support message boxes outside a browser window
 */
var launcher = WScript.createObject("JEdit.JEditLauncher");
var a = new Array("I:\\weather.html", "I:\\test.txt");
b = "I:\\*.pl";
launcher.openFiles(a);
launcher.openFile(b);
c = "G:\\Program Files\\jEdit\\macros\\Misc"
  + "\\Properties\\System_properties.bsh";
launcher.runScript(c);
```

```
# Example Perl script using jEditLauncher interface
use Win32::OLE;
$launcher = Win32::OLE->new('JEdit.JEditLauncher') ||
   die "JEditLauncher: not found !\n";
@files = ();
foreach $entry (@ARGV) {
    @new = glob($entry);
    push(@files,@new);
}
$launcher->openFiles(\@files);
```

```
my($script) = "Macros.message(view, \"I found "
    .(scalar @files)." files.\");";
$launcher->evalScript($script);
```

```
# Example Python script using jEditLauncher interface
import win32gui
import win32com.client
o = win32com.client.Dispatch("JEdit.JEditLauncher")
port = o.ServerPort
if port == 0:
  port = "inactive. We will now launch jEdit"
win32gui.MessageBox(0, "The server port is %s." % port,
    "jEditLauncher", 0)
path = "C:\\WINNT\\Profiles\\Administrator\\Desktop\\"
o.RunDiff(path + "Search.bsh", path + "Search2.bsh")
```

G.7. Legal Notice

All source code and software distributed as the jEditLauncher package in which the author holds the copyright is made available under the GNU General Public License (“GPL”). A copy of the GPL is included in the file `COPYING.txt` included with jEdit.

Notwithstanding the terms of the General Public License, the author grants permission to compile and link object code generated by the compilation of this program with object code and libraries that are not subject to the GNU General Public License, provided that the executable output of such compilation shall be distributed with source code on substantially the same basis as the jEditLauncher package of which this source code and software is a part. By way of example, a distribution would satisfy this condition if it included a working Makefile for any freely available make utility that runs on the Windows family of operating systems. This condition does not require a licensee of this software to distribute any proprietary software (including header files and libraries) that is licensed under terms prohibiting or limiting redistribution to third parties.

The purpose of this specific permission is to allow a user to link files contained or generated by the source code with library and other files licensed to the user by Microsoft or other parties, whether or not that license conforms to the requirements of the GPL. This permission should not be construed to expand the terms of any license for any source code or other materials used in the creation of jEditLauncher.

II. Writing Edit Modes

This part of the user's guide covers writing edit modes for jEdit.

Edit modes specify syntax highlighting rules, auto indent behavior, and various other customizations for editing different file types. For general information about edit modes, see Section 5.1.

This part of the user's guide was written by Slava Pestov <`slava@jedit.org`>.

Chapter 9. Writing Edit Modes

Edit modes are defined using XML, the *extensible markup language*; mode files have the extension `.xml`. XML is a very simple language, and as a result edit modes are easy to create and modify. This section will start with a short XML primer, followed by detailed information about each supported tag and highlighting rule.

Reload Edit Modes command

Utilities>Reload Edit Modes reloads all edit mode XML files from disk. It is very useful when writing edit modes because it lets you see changes take effect without having to restart jEdit.

9.1. An XML Primer

A very simple edit mode looks like so:

```
<?xml version="1.0"?>

<!DOCTYPE MODE SYSTEM "xmode.dtd">

<MODE>
    <PROPS>
        <PROPERTY NAME="commentStart" VALUE="/*" />
        <PROPERTY NAME="commentEnd" VALUE="*/" />
    </PROPS>

    <RULES>
        <SPAN TYPE="COMMENT1">
            <BEGIN>/*</BEGIN>
            <END>*/</END>
        </SPAN>
    </RULES>
</MODE>
```

Note that each opening tag must have a corresponding closing tag. If there is nothing between the opening and closing tags, for example `<TAG></TAG>`, the shorthand notation `<TAG />` may be used. An example of this shorthand can be seen in the `<PROPERTY>` tags above.

XML is case sensitive. `Span` or `span` is not the same as `SPAN`.

To insert a special character such as < or > literally in XML (for example, inside an attribute value), you must write it as an *entity*. An entity consists of the character's symbolic name enclosed with "&" and ";". A full list of entities is out of the scope of this section, but the most important are:

- `&lt;` - The less-than (<) character
- `&gt;` - The greater-than (>) character
- `&amp;` - The ampersand (&) character

For example, the following will cause a syntax error:

```
<SEQ TYPE="OPERATOR">&</SEQ>
```

Instead, you must write:

```
<SEQ TYPE="OPERATOR">&amp;</SEQ>
```

Now that the basics of XML have been covered, the rest of this section will cover each construct in detail.

9.2. The Preamble and MODE tag

Each mode definition must begin with the following:

```
<?xml version="1.0"?>
<!DOCTYPE MODE SYSTEM "xmode.dtd">
```

Each mode definition must also contain exactly one `MODE` tag. All other tags (`PROPS`, `RULES`) must be placed inside the `MODE` tag.

9.3. The PROPS Tag

The `PROPS` tag and the `PROPERTY` tags inside it are used to define mode-specific properties. Each `PROPERTY` tag must have a `NAME` attribute set to the property's name, and a `VALUE` attribute with the property's value.

All buffer-local properties listed in Section 6.2 may be given values in edit modes. In addition, the following mode properties have no buffer-local equivalent:

- `indentCloseBrackets` - A list of characters (usually brackets) that subtract indent from the *current* line. For example, in Java mode this property is set to "}".
- `indentOpenBrackets` - A list of characters (usually brackets) that add indent to the *next* line. For example, in Java mode this property is set to "{".
- `indentPrevLine` - When indenting a line, jEdit checks if the previous line matches the regular expression stored in this property. If it does, a level of indent is added. For example, in Java mode this regular expression matches language constructs such as "if", "else", "while", etc.
- `doubleBracketIndent` - If a line matches the `indentPrevLine` regular expression and the next line contains an opening bracket, a level of indent will not be added to the next line, unless this property is set to "true". For example, with this property set to "false", Java code will be indented like so:

  ```
  while(objects.hasMoreElements())
  {
          ((Drawable)objects.nextElement()).draw();
  }
  ```

  On the other hand, settings this property to "true" will give the following result:

  ```
  while(objects.hasMoreElements())
          {
                  ((Drawable)objects.nextElement()).draw();
          }
  ```

Here is the complete `<PROPS>` tag for Java mode:

```
<PROPS>
```

```
    <PROPERTY NAME="indentOpenBrackets" VALUE="{" />
    <PROPERTY NAME="indentCloseBrackets" VALUE="}" />
    <PROPERTY NAME="indentPrevLine" VALUE="\s*(((if|while)
        \s*\(|else|case|default)[^;]*|for\s*\(.*)" />
    <PROPERTY NAME="doubleBracketIndent" VALUE="false" />
    <PROPERTY NAME="commentStart" VALUE="/*" />
    <PROPERTY NAME="commentEnd" VALUE="*/" />
    <PROPERTY NAME="blockComment" VALUE="//" />
    <PROPERTY NAME="noWordSep" VALUE="_" />
    <PROPERTY NAME="wordBreakChars" VALUE=",+-=<>/?^&*" />
</PROPS>
```

9.4. The RULES Tag

`RULES` tags must be placed inside the `MODE` tag. Each `RULES` tag defines a *ruleset*. A ruleset consists of a number of *parser rules*, with each parser rule specifying how to highlight a specific syntax token. There must be at least one ruleset in each edit mode. There can also be more than one, with different rulesets being used to highlight different parts of a buffer (for example, in HTML mode, one rule set highlights HTML tags, and another highlights inline JavaScript). For information about using more than one ruleset, see Section 9.4.3.

The `RULES` tag supports the following attributes, all of which are optional:

- `SET` - the name of this ruleset. All rulesets other than the first must have a name.
- `HIGHLIGHT_DIGITS` - if set to `TRUE`, digits (0-9, as well as hexadecimal literals prefixed with “0x”) will be highlighted with the `DIGIT` token type. Default is `FALSE`.
- `IGNORE_CASE` - if set to `FALSE`, matches will be case sensitive. Otherwise, case will not matter. Default is `TRUE`.
- `DEFAULT` - the token type for text which doesn’t match any specific rule. Default is `NULL`. See Section 9.4.9 for a list of token types.

Here is an example `RULES` tag:

```
<RULES IGNORE_CASE="FALSE" HIGHLIGHT_DIGITS="TRUE">
```

```
        ... parser rules go here ...
    </RULES>
```

Rule Ordering Requirements

You might encounter this very common pitfall when writing your own modes.

Since jEdit checks buffer text against parser rules in the order they appear in the ruleset, more specific rules must be placed before generalized ones, otherwise the generalized rules will catch everything.

This is best demonstrated with an example. The following is incorrect rule ordering:

```
<SPAN TYPE="MARKUP">
    <BEGIN>[</BEGIN>
    <END>]</END>
</SPAN>

<SPAN TYPE="KEYWORD1">
    <BEGIN>[!</BEGIN>
    <END>]</END>
</SPAN>
```

If you write the above in a rule set, any occurrence of “[” (even things like “[!DEFINE”, etc) will be highlighted using the first rule, because it will be the first to match. This is most likely not the intended behavior.

The problem can be solved by placing the more specific rule before the general one:

```
<SPAN TYPE="KEYWORD1">
    <BEGIN>[!</BEGIN>
    <END>]</END>
</SPAN>

<SPAN TYPE="MARKUP">
    <BEGIN>[</BEGIN>
    <END>]</END>
</SPAN>
```

Now, if the buffer contains the text “[!SPECIAL]”, the rules will be checked in order, and the first rule will be the first to match. However, if you write “[FOO]”, it will be highlighted using the second rule, which is exactly what you would expect.

9.4.1. The TERMINATE Rule

The `TERMINATE` rule specifies that parsing should stop after the specified number of characters have been read from a line. The number of characters to terminate after should be specified with the `AT_CHAR` attribute. Here is an example:

```
<TERMINATE AT_CHAR="1" />
```

This rule is used in Patch mode, for example, because only the first character of each line affects highlighting.

9.4.2. The WHITESPACE Rule

The `WHITESPACE` rule specifies characters which are to be treated as keyword delimiters. Most rulesets will have `WHITESPACE` tags for spaces and tabs. Here is an example:

```
<WHITESPACE> </WHITESPACE>
<WHITESPACE>        </WHITESPACE>
```

9.4.3. The SPAN Rule

The `SPAN` rule highlights text between a start and end string. The start and end strings are specified inside child elements of the `SPAN` tag. The following attributes are supported:

- `TYPE` - The token type to highlight the span with. See Section 9.4.9 for a list of token types
- `AT_LINE_START` - If set to `TRUE`, the span will only be highlighted if the start sequence occurs at the beginning of a line
- `EXCLUDE_MATCH` - If set to `TRUE`, the start and end sequences will not be highlighted, only the text between them will

- `NO_LINE_BREAK` - If set to `TRUE`, the span will be highlighted with the `INVALID` token type if it spans more than one line
- `NO_WORD_BREAK` - If set to `TRUE`, the span will be highlighted with the `INVALID` token type if it includes whitespace
- `DELEGATE` - text inside the span will be highlighted with the specified ruleset. To delegate to a ruleset defined in the current mode, just specify its name. To delegate to a ruleset defined in another mode, specify a name of the form *`mode`*`::`*`ruleset`*. Note that the first (unnamed) ruleset in a mode is called “MAIN”.

  Note: Do not delegate to rulesets that define a `TERMINATE` rule (examples of such rulesets include `text::MAIN` and `patch::MAIN`). It won’t work.

Here is a `SPAN` that highlights Java string literals, which cannot include line breaks:

```
<SPAN TYPE="LITERAL1" NO_LINE_BREAK="TRUE">
   <BEGIN>"</BEGIN>
   <END>"</END>
</SPAN>
```

Here is a `SPAN` that highlights Java documentation comments by delegating to the “JAVADOC” ruleset defined elsewhere in the current mode:

```
<SPAN TYPE="COMMENT2" DELEGATE="JAVADOC">
   <BEGIN>/**</BEGIN>
   <END>*/</END>
</SPAN>
```

Here is a `SPAN` that highlights HTML cascading stylesheets inside `<STYLE>` tags by delegating to the main ruleset in the CSS edit mode:

```
<SPAN TYPE="MARKUP" DELEGATE="css::MAIN">
   <BEGIN>&lt;style&gt;</BEGIN>
   <END>&lt;/style&gt;</END>
</SPAN>
```

> **Tip:** The `<END>` tag is optional. If it is not specified, any occurrence of the start string will cause the remainder of the buffer to be highlighted with this rule.
>
> This can be very useful when combined with delegation.

9.4.4. The EOL_SPAN Rule

An `EOL_SPAN` is similar to a `SPAN` except that highlighting stops at the end of the line, not after the end sequence is found. The text to match is specified between the opening and closing `EOL_SPAN` tags. The following attributes are supported:

- `TYPE` - The token type to highlight the span with. See Section 9.4.9 for a list of token types
- `AT_LINE_START` - If set to `TRUE`, the span will only be highlighted if the start sequence occurs at the beginning of a line
- `EXCLUDE_MATCH` - If set to `TRUE`, the start sequence will not be highlighted, only the text after it will

Here is an `EOL_SPAN` that highlights C++ comments:

```
<EOL_SPAN TYPE="COMMENT1">//</EOL_SPAN>
```

9.4.5. The MARK_PREVIOUS Rule

The `MARK_PREVIOUS` rule highlights from the end of the previous syntax token to the matched text. The text to match is specified between opening and closing `MARK_PREVIOUS` tags. The following attributes are supported:

- `TYPE` - The token type to highlight the text with. See Section 9.4.9 for a list of token types
- `AT_LINE_START` - If set to `TRUE`, the text will only be highlighted if it occurs at the beginning of the line

- `EXCLUDE_MATCH` - If set to `TRUE`, the match will not be highlighted, only the text before it will

Here is a rule that highlights labels in Java mode (for example, “XXX:”):

```
<MARK_PREVIOUS AT_LINE_START="TRUE"
    EXCLUDE_MATCH="TRUE">:</MARK_PREVIOUS>
```

9.4.6. The MARK_FOLLOWING Rule

The `MARK_FOLLOWING` rule highlights from the start of the match to the next syntax token. The text to match is specified between opening and closing `MARK_FOLLOWING` tags. The following attributes are supported:

- `TYPE` - The token type to highlight the text with. See Section 9.4.9 for a list of token types
- `AT_LINE_START` - If set to `TRUE`, the text will only be highlighted if the start sequence occurs at the beginning of a line
- `EXCLUDE_MATCH` - If set to `TRUE`, the match will not be highlighted, only the text after it will

Here is a rule that highlights variables in Unix shell scripts (“$CLASSPATH”, “$IFS”, etc):

```
<MARK_FOLLOWING TYPE="KEYWORD2">$</MARK_FOLLOWING>
```

9.4.7. The SEQ Rule

The `SEQ` rule highlights fixed sequences of text. The text to highlight is specified between opening and closing `SEQ` tags. The following attributes are supported:

- `TYPE` - the token type to highlight the sequence with. See Section 9.4.9 for a list of token types

- `AT_LINE_START` - If set to `TRUE`, the sequence will only be highlighted if it occurs at the beginning of a line

The following rules highlight a few Java operators:

```
<SEQ TYPE="OPERATOR">+</SEQ>
<SEQ TYPE="OPERATOR">-</SEQ>
<SEQ TYPE="OPERATOR">*</SEQ>
<SEQ TYPE="OPERATOR">/</SEQ>
```

9.4.8. The KEYWORDS Rule

There can only be one `KEYWORDS` tag per ruleset. The `KEYWORDS` rule defines keywords to highlight. Keywords are similar to `SEQ`s, except that `SEQ`s match anywhere in the text, whereas keywords only match whole words.

The `KEYWORDS` tag supports only one attribute, `IGNORE_CASE`. If set to `FALSE`, keywords will be case sensitive. Otherwise, case will not matter. Default is `TRUE`.

Each child element of the `KEYWORDS` tag should be named after the desired token type, with the keyword text between the start and end tags. For example, the following rule highlights the most common Java keywords:

```
<KEYWORDS IGNORE_CASE="FALSE">
   <KEYWORD1>if</KEYWORD1>
   <KEYWORD1>else</KEYWORD1>
   <KEYWORD3>int</KEYWORD3>
   <KEYWORD3>void</KEYWORD3>
</KEYWORDS>
```

9.4.9. Token Types

Parser rules can highlight tokens using any of the following token types:

- `NULL` - no special highlighting is performed on tokens of type `NULL`

- `COMMENT1`
- `COMMENT2`
- `FUNCTION`
- `INVALID` - tokens of this type are automatically added if a `NO_WORD_BREAK` or `NO_LINE_BREAK` `SPAN` spans more than one word or line, respectively.
- `KEYWORD1`
- `KEYWORD2`
- `KEYWORD3`
- `LABEL`
- `LITERAL1`
- `LITERAL2`
- `MARKUP`
- `OPERATOR`

Chapter 10. Installing Edit Modes

jEdit looks for edit modes in two locations; the `modes` subdirectory of the jEdit settings directory, and the `modes` subdirectory of the jEdit install directory. The location of the settings directory is system-specific; see Section 6.4.

Each mode directory contains a `catalog` file. All edit modes contained in that directory must be listed in the catalog, otherwise they will not be available to jEdit.

Catalogs, like modes themselves, are written in XML. They consist of a single `MODES` tag, with a number of `MODE` tags inside. Each mode tag associates a mode name with an XML file, and specifies the file name and first line pattern for the mode. A sample mode catalog looks like follows:

```
<?xml version="1.0"?>
<!DOCTYPE CATALOG SYSTEM "catalog.dtd">

<MODES>
    <MODE NAME="shellscript" FILE="shellscript.xml"
        FILE_NAME_GLOB="*.sh"
        FIRST_LINE_GLOB="#!/*sh*" />
</MODES>
```

In the above example, a mode named “shellscript” is defined, and is used for files whose names end with `.sh`, or whose first line starts with “#!/” and contains “sh”.

The `MODE` tag supports the following attributes:

- `NAME` - the name of the edit mode, as it will appear in the Buffer Options dialog box, the status bar, and so on
- `FILE` - the name of the XML file containing the mode definition
- `FILE_NAME_GLOB` - files whose names match this glob pattern will be opened in this edit mode. See Appendix D for information about glob patterns
- `FIRST_LINE_GLOB` - files whose first line matches this glob pattern will be opened in this edit mode. See Appendix D for information about glob patterns

If an edit mode is defined in the user-specific catalog with the same name as an edit mode in the global catalog, the version in the user-specific catalog will be used instead of the other version.

III. Writing Macros

This part of the user's guide covers writing macros for jEdit.

First, we will tell you a little about BeanShell, jEdit's macro scripting language. Next, we will walk through a few simple macros. We then present and analyze a dialog-based macro to illustrate additional macro writing techniques. Finally, we discuss several tips and techniques for writing and debugging macros.

This part of the user's guide was written by John Gellene <`jgellene@nyc.rr.com`>.

Chapter 11. Introducing BeanShell

Here is how BeanShell's author, Pat Niemeyer, describes his creation:

> "BeanShell is a small, free, embeddable, Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions, in addition to obvious scripting commands and syntax. BeanShell supports scripted objects as simple method closures like those in Perl and JavaScript."

As you might gather from this short quote, BeanShell is very similar to Java and is designed to be easy for Java programmers to learn. If you know how to program in Java, you already know how to write BeanShell macros. Nonetheless, the premise of this guide is that you should not have to know anything about Java to begin writing your own macros for jEdit.

If you are not a Java programmer, you will have to learn a little about Java classes and syntax, but that's not a bad thing. You will also have to learn a little (but not too much) about some of the classes that are defined and used by the jEdit program itself. That is in fact the major strength of using BeanShell with a program written in Java: it allows the user to customize the program's behavior by employing the same interfaces designed and used by the program's developer. Thus, BeanShell can turn a well-designed application into a powerful toolkit.

This guide focuses on using BeanShell in macros. If you are interested in learning more about BeanShell generally, consult the BeanShell web site (http://www.beanshell.org). Information on how to run and organize macros, whether included with the jEdit installation or written by you, can be found in Chapter 7.

11.1. Single Execution Macros

There are two ways jEdit lets you use BeanShell quickly on a "one time only" basis. You will find both of them in the Utilities menu.

Utilities>Evaluate BeanShell Expression causes jEdit to display a text input dialog that asks you to type a single line of BeanShell commands. You can type more than one BeanShell statement so long as each of them ends with a semicolon. If BeanShell successfully interprets your input, a message box will appear with the return value of the last statement. You can do the same thing using the BeanShell interpreter provided with the Console plugin; the return value will appear in the output window.

Utilities>Evaluate Selection evaluates the selected text as a BeanShell script and replaces the selected text with the return value of the last BeanShell statement.

Using Evaluate Selection is an easy way to do arithmetic calculations inline while editing. BeanShell uses numbers and arithmetic operations in an ordinary, intuitive way.

Try typing an expression like **`(3745*856)+74`** in the buffer, select it, and choose Utilities>Evaluate Selection. The selected text will be replaced by the answer, **`3205794`**.

Chapter 12. A Few Simple Macros

12.1. The Mandatory First Example

```
Macros.message(view, "Hello world!");
```

Running this one line script causes jEdit to display a message box (more precisely, a `JOptionPane` object) with the traditional beginner's message and an OK button. Let's see what is happening here.

This statement calls a static method (or function) named `message` in jEdit's `Macros` class. If you don't know anything about classes or static methods or Java (or C++, which employs the same concept), you will need to gain some understanding of a few terms. Obviously this is not the place for academic precision, but if you are entirely new to object-oriented programming, here are a few skeleton ideas to help you with BeanShell.

- An *object* is a collection of data that can be initialized, accessed and manipulated in certain defined ways.
- A *class* is a specification of what data an object contains and what methods can be used to work with the data. A Java application consists of one or more classes (in the case of jEdit over 200 classes) written by the programmer that defines the application's behavior. A BeanShell macro uses these classes, along with built-in classes that are supplied with the Java platform, to define its own behavior.
- A *subclass* (or child class) is a class which uses (or "inherits") the data and methods of its parent class along with additions or modifications that alter the subclass's behavior. Classes are typically organized in hierarchies of parent and child classes to organize program code, to define common behavior in shared parent class code, and to specify the types of similar behavior that child classes will perform in their own specific ways.
- A *method* (or function) is a procedure that works with data in a particular object, other data (including other objects) supplied as *parameters*, or both. Methods typically are

applied to a particular object which is an *instance* of the class to which the method belongs.

- A *static method* differs from other methods in that it does not deal with the data in a particular object but is included within a class for the sake of convenience.

Java has a rich set of classes defined as part of the Java platform. Like all Java applications, jEdit is organized as a set of classes that are themselves derived from the Java platform's classes. We will refer to *Java classes* and *jEdit classes* to make this distinction. Some of jEdit's classes (such as those dealing with regular expressions and XML) are derived from or make use of classes in other open-source Java packages. Except for BeanShell itself, we won't be discussing them in this guide.

In our one line script, the static method `Macros.message()` has two parameters because that is the way the method is defined in the `Macros` class. You must specify both parameters when you call the function. The first parameter, *`view`*, is a a variable naming a `View` object - an instance of jEdit's `View` class. A `View` represents a "parent" or top-level frame window that contains the various visible components of the program, including the text area, menu bar, toolbar, and any docked windows. It is a subclass of Java's `JFrame` class. With jEdit, you can create and display multiple views simultaneously. The variable `view` is predefined for purposes of BeanShell as the current, active `View` object. This is in fact the variable you want to specify as the first parameter. Normally you would not want to associate a message box with anything other than the current `View`.

The second parameter, which appears to be quoted text, is a *string literal* - a sequence of characters of fixed length and content. Behind the scenes, BeanShell and Java take this string literal and use it to create a `String` object. Normally, if you want to create an object in Java or BeanShell, you must construct the object using the `new` keyword and a *constructor* method that is part of the object's class. We'll show an example of that later. However, both Java and BeanShell let you use a string literal anytime a method's parameter calls for a `String`.

If you are a Java programmer, you might wonder about a few things missing from this one line program. There is no class definition, for example. You can think of a BeanShell script as an implicit definition of a `main()` method in an anonymous class. That is in fact how BeanShell is implemented; the class is derived from a BeanShell class called `XThis`. If you don't find that helpful, just think of a script as one or more blocks of procedural statements conforming to Java syntax rules. You will also get along fine (for the most part) with C or C++ syntax if you leave out anything to do with pointers or memory management - Java and BeanShell do not have pointers and deal with memory management automatically.

Another missing item from a Java perspective is a `package` statement. In Java, such a statement is used to bundle together a number of files so that their classes become visible to one another. Packages are not part of BeanShell, and you don't need to know anything about them to write BeanShell macros.

Finally, there are no `import` statements in this script. In Java, an `import` statement makes public classes from other packages visible within the file in which the statement occurs without having to specify a fully qualified class name. Without an import statement or a fully qualified name, Java cannot identify most classes using a single name as an identifier.

jEdit automatically imports a number of commonly-used packages into the namespace of every BeanShell script. Because of this, the script output of a recorded macro does not contain `import` statements. For the same reason, most BeanShell scripts you write will not require `import` statements.

Java requires `import` statement at the beginning of a source file. BeanShell allows you to place `import` statements anywhere in a script, including inside of block of statements. The `import` statement will cover all names used following the statement in the enclosing block.

If you try to use a class that is not imported without its fully-qualified name, the BeanShell interpreter will complain with an error message relating to the offending line of code.

Here is the full list of packages automatically imported by jEdit:

```
java.awt
java.awt.event
java.net
java.util
java.io
java.lang
javax.swing
javax.swing.event
org.gjt.sp.jedit
org.gjt.sp.jedit.browser
org.gjt.sp.jedit.gui
org.gjt.sp.jedit.io
org.gjt.sp.jedit.msg
org.gjt.sp.jedit.options
org.gjt.sp.jedit.pluginmgr
org.gjt.sp.jedit.search
org.gjt.sp.jedit.syntax
org.gjt.sp.jedit.textarea
org.gjt.sp.util
```

12.2. Helpful Methods in the Macros Class

Including `message()`, there are four static methods in the `Macros` class that allow you to converse easily with your macros. They all encapsulate calls to methods of the Java platform's `JOptionPane` class.

- `public static void` **`message`**`(View` *`view`*`, String` *`message`*`);`
- `public static void` **`error`**`(View` *`view`*`, String` *`message`*`);`
- `public static String` **`input`**`(View` *`view`*`, String` *`prompt`*`);`
- `public static String` **`input`**`(View` *`view`*`, String` *`prompt`*`, String`

```
    defaultValue);
```

The format of these four *declarations* provides a concise reference to the way in which the methods may be used. The keyword `public` means that the method can be used outside the `Macros` class. The alternatives are `private` and `protected`. For purposes of BeanShell, you just have to know that BeanShell can only use public methods of other Java classes. The keyword `static` we have already discussed. It means that the method does not operate on a particular object. You call a static function using the name of the class (like `Macros`) rather than the name of a particular object (like `view`). The third word is the type of the value returned by the method. The keyword `void` is Java's way of saying the the method does not have a return value.

The `error()` method works just like `message()` but displays an error icon in the message box. The `input()` method furnishes a text field for input, an OK button and a Cancel button. If "Cancel" is pressed, the method returns `null`. If OK is pressed, a `String` containing the contents of the text field is returned. Note that there are two forms of the `input()` method; the first form with two parameters displays an empty input field, the other lets you specify an initial default value.

For those without Java experience, it is important to know that `null` is *not* the same as an empty, "zero-length" `String`. It is Java's way of saying that there is no object associated with this variable. Whenever you seek to use a return value from `input()` in your macro, you should test it to see if it is `null`. In most cases, you will want to exit gracefully from the script with a `return` statement, because the presence of a null value for an input variable usually means that the user intended to cancel macro execution. BeanShell will complain if you call any methods on a `null` object.

We've looked at using `Macros.message()`. To use the other methods, you would write something like the following:

```
Macros.error(view, "Goodbye, cruel world!");

String result = Macros.input(view, "Type something here.");

String result = Macros.input(view, "What is your name?",
    "John Gellene");
```

In the last two examples, placing the word `String` before the variable name `result` tells BeanShell that the variable refers to a `String` object, even before a particular `String` object is assigned to it. In BeanShell, this *declaration* of the *type* of `result` is not necessary; BeanShell can figure it out when the macro runs. This can be helpful if you are not comfortable with types and classes; just use your variables and let BeanShell worry about it.

Without an explicit *type declaration* like `String result`, BeanShell variables can change their type at runtime depending on the object or data assigned to it. This dynamic typing allows you to write code like this (if you really wanted to):

```
// note: no type declaration
result = Macros.input(view, “Type something here.”);

// this is our predefined, current View
result = view;

// this is an “int” (for integer);
// in Java and BeanShell, int is one of a small number
// of “primitive” data types which are not classes
result = 14;
```

However, if you first declared `result` to be type `String` and and then tried these reassignments, BeanShell would complain.

One last thing before we bury our first macro. The double slashes in the examples just above signify that everything following them on that line should be ignored by BeanShell as a comment. As in Java and C/C++, you can also embed comments in your BeanShell code by setting them off with pairs of **`/* */`**, as in the following example:

```
/* This is a long comment that covers several lines
and will be totally ignored by BeanShell regardless of how
many lines it covers */
```

12.3. Now For Something Useful

Here is a macro that inserts the path of the current buffer in the text:

```
String newText = buffer.getPath();
textArea.setSelectedText(newText);
```

Two of the new names we see here, `buffer` and `textArea`, are predefined variables like `view`. The variable `buffer` represents a jEdit `Buffer` object, and `textArea` represents a `JEditTextArea` object.

- A `Buffer` represents the contents of an open text file. It is derived from Java's `PlainDocument` class. The variable `buffer` is predefined as the current buffer.
- A `JEditTextArea` is the visible component that displays the file being edited. It is derived from the `JComponent` class. The variable `textArea` represents the current `JEditTextArea` object, which in turn displays the current buffer.

Unlike in our first macro example, here we are calling class methods on particular objects. First, we call `getPath()` on the current `Buffer` object to get the full path of the text file currently being edited. Next, we call `setSelectedText()` on the current text display component, specifying the text to be inserted as a parameter.

In precise terms, the `setSelectedText()` method substitutes the contents of the `String` parameter for a range of selected text that includes the current caret position. If no text is selected at the caret position, the effect of this operation is simply to insert the new text at that position.

Here's a few alternatives to the full file path that you could use to insert various useful things:

```
// the file name (without full path)
String newText = buffer.getName();

// today's date
import java.text.DateFormat;

String newText = DateFormat.getDateInstance()
    .format(new Date());

// a line count for the current buffer
String newText = "This file contains "
```

```
    + textArea.getLineCount() + " lines.";
```

Here are brief comments on each:

- In the first, the call to `getName()` invokes another method of the `Buffer` class.
- The syntax of the second example chains the results of several methods. You could write it this way:

  ```
  import java.text.DateFormat;
  Date d = new Date();
  DateFormat df = DateFormat.getDateInstance();
  String result = df.format(d);
  ```

  Taking the pieces in order:

  - A Java `Date` object is created using the `new` keyword. The empty parenthesis after `Date` signify a call on the *constructor method* of `Date` having no parameters; here, a `Date` is created representing the current date and time.
  - `DateFormat.getDateInstance()` is a static method that creates and returns a `DateFormat` object. As the name implies, `DateFormat` is a Java class that takes `Date` objects and produces readable text. The method `getDateInstance()` returns a `DateFormat` object that parses and formats dates. It will use the default *locale* or text format specified in the user's Java installation.
  - Finally, `DateFormat.format()` is called on the new `DateFormat` object using the `Date` object as a parameter. The result is a `String` containing the date in the default locale.
  - Note that the `Date` class is contained in the `java.util` package, so an explicit import statement is not required. However, `DateFormat` is part of the `java.text` package, which is not automatically imported, so an explicit `import` statement must be used.

- The third example shows three items of note:
  - `getLineCount()` is a method in jEdit's `JEditTextArea` class. It returns an int representing the number of lines in the current text buffer. We call it on `textArea`,

the pre-defined, current `JEditTextArea` object.

- The use of the `+` operator (which can be chained, as here) appends objects and string literals to return a single, concatenated `String`.

The other pre-defined variable

In addition to `view`, `buffer` and `textArea`, there is one more pre-defined variable available for use in macros – `editPane`. That variable is set to the current `EditPane` instance. An `EditPane` object contains a text area and buffer switcher. A view can be split to display multiple buffers, each in its own edit pane. Among other things, the `EditPane` class contains methods for selecting the buffer to edit.

Most of the time your macros will manipulate the `buffer` or the `textArea`. Sometimes you will need to use `view` as a parameter in a method call. You will probably only need to use `editPane` if your macros work with split views.

Chapter 13. A Dialog-Based Macro

Now we will look at a more complicated macro which will demonstrate some useful techniques and BeanShell features.

13.1. Use of the Macro

Our new example adds prefix and suffix text to a series of selected lines. This macro can be used to reduce typing for a series of text items that must be preceded and following by identical text. In Java, for example, if we are interested in making a series of calls to `StringBuffer.append()` to construct a lengthy, formatted string, we could type the parameter for each call on successive lines as follows:

```
profileString_1
secretThing.toString()
name
address
addressSupp
city
“state/province”
country
```

Our macro would ask for input for the common “prefix” and “suffix” to be applied to each line; in this case, the prefix is **`ourStringBuffer.append(`** and the suffix is **`);`**. After selecting these lines and running the macro, the the resulting text would look like this:

```
ourStringBuffer.append(profileString_1);
ourStringBuffer.append(secretThing.toString());
ourStringBuffer.append(name);
ourStringBuffer.append(address);
ourStringBuffer.append(addressSupp);
ourStringBuffer.append(city);
ourStringBuffer.append(“state/province”);
ourStringBuffer.append(country);
```

13.2. Listing of the Macro

The macro script follows. You can find it in the jEdit distribution in the `Text` subdirectory of the `macros` directory. You can also try it out by invoking Macros>Text>Add Prefix and Suffix.

```
// beginning of Add_Prefix_and_Suffix.bsh

// import statement (see Section 13.3.1)
import javax.swing.border.*;

// main routine
void prefixSuffixDialog()
{
    // create dialog object (see Section 13.3.2)
    title = “Add prefix and suffix to selected lines”;
    dialog = new JDialog(view, title, false);
    content = new JPanel(new BorderLayout());
    content.setBorder(new EmptyBorder(12, 12, 12, 12));
    content.setPreferredSize(new Dimension(320, 160));
    dialog.setContentPane(content);

    // add the text fields (see Section 13.3.3)
    fieldPanel = new JPanel(new GridLayout(4, 1, 0, 6));
    prefixField = new HistoryTextField(“macro.add-prefix”);
    prefixLabel = new JLabel(“Prefix to add:”);
    suffixField = new HistoryTextField(“macro.add-suffix”);
    suffixLabel = new JLabel(“Suffix to add:”);
    fieldPanel.add(prefixLabel);
    fieldPanel.add(prefixField);
    fieldPanel.add(suffixLabel);
    fieldPanel.add(suffixField);
    content.add(fieldPanel, “Center”);

    // add the buttons (see Section 13.3.4)
    buttonPanel = new JPanel();
    buttonPanel.setLayout(new BoxLayout(buttonPanel,
        BoxLayout.X_AXIS));
    buttonPanel.setBorder(new EmptyBorder(12, 50, 0, 50));
buttonPanel.add(Box.createGlue());
ok = new JButton(“OK”);
cancel = new JButton(“Cancel”);
ok.setPreferredSize(cancel.getPreferredSize());
dialog.getRootPane().setDefaultButton(ok);
buttonPanel.add(ok);
buttonPanel.add(Box.createHorizontalStrut(6));
buttonPanel.add(cancel);
buttonPanel.add(Box.createGlue());
content.add(buttonPanel, “South”);

// register this method as an ActionListener for
// the buttons and text fields (see Section 13.3.5)
ok.addActionListener(this);
cancel.addActionListener(this);
prefixField.addActionListener(this);
suffixField.addActionListener(this);

// locate the dialog in the center of the
// editing pane and make it visible (see Section 13.3.6)
dialog.pack();
dialog.setLocationRelativeTo(view);
dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
dialog.setVisible(true);

// this method will be called when a button is clicked
// or when ENTER is pressed (see Section 13.3.7)
void actionPerformed(e)
{
    if(e.getSource() != cancel)
    {
        processText();
    }
    dialog.dispose();
}

// this is where the work gets done to insert
// the prefix and suffix (see Section 13.3.8)
void processText()
{
        prefix = prefixField.getText();
        suffix = suffixField.getText();
        if(prefix.length() == 0 && suffix.length() == 0)
            return;
        if(prefix.length() != 0)
            prefixField.addCurrentToHistory();
        if(suffix.length() != 0)
            suffixField.addCurrentToHistory();

        // text manipulation begins here using calls
        // to jEdit methods  (see Section 13.3.9)
        selectedLines = textArea.getSelectedLines();
        for(i = 0; i < selectedLines.length; ++i)
        {
            offsetBOL = textArea.getLineStartOffset(
                selectedLines[i]);
            textArea.setCaretPosition(offsetBOL);
            textArea.goToStartOfWhiteSpace(false);
            textArea.goToEndOfWhiteSpace(true);
            text = textArea.getSelectedText();
            if(text == null) text = "";
            textArea.setSelectedText(prefix + text + suffix);
        }
    }
}

// this single line of code is the script’s main routine
// (see Section 13.3.10)
prefixSuffixDialog();

// end of Add_Prefix_and_Suffix.bsh
```

13.3. Analysis of the Macro

13.3.1. Import Statements

```
// import statement
import javax.swing.border.*;
```

This macro makes use of classes in the `javax.swing.border` package, which is not automatically imported. As we mentioned previously (see Section 12.1), jEdit's implementation of BeanShell causes a number of classes to be automatically imported. Classes that are not automatically imported must be named by a full qualified name or be the subject of an `import` statement.

13.3.2. Create the Dialog

```
// create dialog object
title = “Add prefix and suffix to selected lines”;
dialog = new JDialog(view, title, false);
content = new JPanel(new BorderLayout());
content.setBorder(new EmptyBorder(12, 12, 12, 12));
dialog.setContentPane(content);
```

To get input for the macro, we need a dialog that provides for input of the prefix and suffix strings, an OK button to perform text insertion, and a Cancel button in case we change our mind. We have decided to make the dialog window non-modal. This will allow us to move around in the text buffer to find things we may need (including text to cut and paste) while the macro is running and the dialog is visible.

The Java object we need is a `JDialog` object from the Swing package. To construct one, we use the `new` keyword and call a *constructor* function. The constructor we use takes three parameters: the owner of the new dialog, the title to be displayed in the dialog frame, and a boolean parameter (`true` or `false`) that specifies whether the dialog will be modal or non-modal. We define the variable `title` using a string literal, then use it immediately in the `JDialog` constructor.

A `JDialog` object is a window containing a single object called a *content pane*. The content pane in turn contains the various visible components of the dialog. A `JDialog` creates an empty content pane for itself as during its construction. However, to control the dialog's appearance as much as possible, we will separately create our own content pane and attach it to the `JDialog`. We do this by creating a `JPanel` object. A `JPanel` is a lightweight container for other components that can be set to a given size and color. It also contains a *layout* scheme for arranging the size and position of its components. Here we are constructing a `JPanel` as a content pane with a `BorderLayout`. We put a `EmptyBorder` inside it to serve as a margin between the edge of the window and the components inside. We then attach the `JPanel` as the dialog's content pane, replacing the dialog's home-grown version.

A `BorderLayout` is one of the simpler layout schemes available for Java Swing objects. A `BorderLayout` divides the container into five sections: "North", "South", ""East, "West" and "Center". Components are added to the layout using the container's `add` method, specifying the component to be added and the section to which it is assigned. Building a component like our dialog window involves building a set of nested containers and specifying the location of each of their member components. We have taken the first step by creating a `JPanel` as the dialog's content pane.

13.3.3. Create the Text Fields

```
// add the text fields
fieldPanel = new JPanel(new GridLayout(4, 1, 0, 6));
prefixField = new HistoryTextField("macro.add-prefix");
prefixLabel = new JLabel(“Prefix to add”:);
suffixField = new HistoryTextField(“macro.add-suffix”);
suffixLabel = new JLabel(“Suffix to add:”);
fieldPanel.add(prefixLabel);
fieldPanel.add(prefixField);
fieldPanel.add(suffixLabel);
fieldPanel.add(suffixField);
content.add(fieldPanel, “Center”);
```

Next we shall create a smaller panel containing two fields for entering the prefix and suffix text and two labels identfying the input fields.

For the text fields, we will use jEdit's `HistoryTextField` class. It is derived from the Java Swing class `JTextField`. This class offers the enhancement of a stored list of prior values used as text input. The up and down keys scroll through the prior values for the variable.

To create the `HistoryTextField` objects we use a constructor method that takes a single parameter: the name of the tag under which history values will be stored. Here we choose names that are not likely to conflict with existing jEdit history items.

The labels are `JLabel` objects from the Java Swing package. The constructor we use takes the label text as a single `String` parameter.

We wish to arrange these four components from top to bottom, one after the other. To achieve that, we use a `JPanel` object named `fieldPanel` that will be nested inside the dialog's content pane that we have already created. In the constructor for `fieldPanel`, we assign a new `GridLayout` with the indicated parameters: four rows, one column, zero spacing between columns (a meaningless element of a grid with only one column, but nevertheless a required parameter) and spacing of six pixels between rows. The spacing between rows spreads out the four "grid" elements. After the components, the panel and the layout are specified, the components are added to `fieldPanel` top to bottom, one "grid cell" at a time. Finally, the complete `fieldPanel` is added to the dialog's content pane to occupy the "Center" section of the content pane.

13.3.4. Create the Buttons

```
// add the buttons
buttonPanel = new JPanel();
buttonPanel.setLayout(new BoxLayout(buttonPanel,
    BoxLayout.X_AXIS));
buttonPanel.setBorder(new EmptyBorder(12, 50, 0, 50));
buttonPanel.add(Box.createGlue());
ok = new JButton(“OK”);
cancel = new JButton(“Cancel”);
ok.setPreferredSize(cancel.getPreferredSize());
```

```
dialog.getRootPane().setDefaultButton(ok);
buttonPanel.add(ok);
buttonPanel.add(Box.createHorizontalStrut(6));
buttonPanel.add(cancel);
buttonPanel.add(Box.createGlue());
content.add(buttonPanel, “South”);
```

Creating the buttons repeats the pattern we used in creating the text fields. First, we create a new, nested panel with a `BoxLayout`. A `BoxLayout` places components either in a single row or column, depending on the parameter passed to its constructor. We put an `EmptyBorder` in the new panel to set margins for placing the buttons. Then we create the buttons, using a `JButton` constructor that specifies the button text. After setting the size of the OK button to equal the size of the Cancel button, we designate the OK button as the default button in the dialog. This causes the OK button to be outlined as the default button. Finally, we place the button side by side with a 6 pixel gap between them (for aesthetic reasons), and place the completed `buttonPanel` in the “South” section of the dialog’s content pane.

13.3.5. Register the Action Listeners

```
// register this method as an ActionListener for
// the buttons and text fields
ok.addActionListener(this);
cancel.addActionListener(this);
prefixField.addActionListener(this);
suffixField.addActionListener(this);
```

In order to specify the action to be taken upon clicking a button or pressing the **Enter** key, we must register an `ActionListener` for each of the four active components of the dialog - the two `HistoryTextField` components and the two buttons. In Java, an `ActionListener` is an *interface* - an abstract specification for a derived class to implement. The `ActionListener` interface contains a single method to be implemented:

```
public void actionPerformed(ActionEvent e);
```

BeanShell does not permit a script to create derived classes. However, BeanShell offers a useful substitute: a method can be used as a scripted object that can implement methods of a number of Java interfaces. The method `prefixSuffixDialog()` that we are writing can thus be treated as an `ActionListener`. To accomplish this, we call `addActionListener()` on each of the four components specifying `this` as the `ActionListener`. We still need to implement the interface. We will do that shortly.

13.3.6. Make the Dialog Visible

```
// locate the dialog in the center of the
// editing pane and make it visible
dialog.pack();
dialog.setLocationRelativeTo(view);
dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
dialog.setVisible(true);
```

Here we do three things. First, we activate all the layout routines we have established by calling the `pack()` method for the dialog as the top-level window. Next we center the dialog's position in the active jEdit `view` by calling `setLocationRelativeTo()` on the dialog. We also call the `setDefaultCloseOperation()` function to specify that the dialog box should be immediately disposed if the user clicks the close box. Finally, we activate the dialog by calling `setVisible()`with the state parameter set to `true`.

At this point we have a decent looking dialog window that doesn't do anything. Without more code, it will not respond to user input and will not accomplish any text manipulation. The remainder of the script deals with these two requirements.

13.3.7. The Action Listener

```
// this method will be called when a button is clicked
// or when ENTER is pressed
void actionPerformed(e)
{
    if(e.getSource() != cancel)
    {
        processText();
        }
        dialog.dispose();
    }
```

The method `actionPerformed()` nested inside `prefixSuffixDialog()` implements the implicit `ActionListener` interface. It looks at the source of the `ActionEvent`, determined by a call to `getSource()`. What we do with this return value is straighforward: if the source is not the Cancel button, we call the `processText()` method to insert the prefix and suffix text. Then the dialog is closed by calling its `dispose()` method.

The ability to implement interfaces like `ActionListener` inside a BeanShell script is one of the more powerful features of the BeanShell package. With an `ActionListener` interface, which has only a single method, implementation is simple. When using other interfaces with multiple methods, however, there are some details to deal with that will vary depending on the version of the Java platform that you are running. These techniques are discussed in the next chapter; see Section 14.4.3.

13.3.8. Get the User's Input

```
// this is where the work gets done to insert
// the prefix and suffix
void processText()
{
    prefix = prefixField.getText();
    suffix = suffixField.getText();
    if(prefix.length() == 0 && suffix.length() == 0)
        return;
    if(prefix.length() != 0)
        prefixField.addCurrentToHistory();
    if(suffix.length() != 0)
        suffixField.addCurrentToHistory();
```

The method `processText()` does the work of our macro. First we obtain the input from the two text fields with a call to their `getText()` methods. If they are both empty, there is nothing to do, so the method returns. If there is input, any text in the field is added to that field's stored history list by calling `addCurrentToHistory()`.

13.3.9. Call jEdit Methods to Manipulate Text

```
    // text manipulation begins here using calls
    // to jEdit methods
    selectedLines = textArea.getSelectedLines();
    for(i = 0; i < selectedLines.length; ++i)
    {
        offsetBOL = textArea.getLineStartOffset(
            selectedLines[i]);
        textArea.setCaretPosition(offsetBOL);
        textArea.goToStartOfWhiteSpace(false);
        textArea.goToEndOfWhiteSpace(true);
        text = textArea.getSelectedText();
        if(text == null) text = "";
        textArea.setSelectedText(prefix + text + suffix);
    }
}
```

The text manipulation routine loops through each selected line in the text buffer. We get the loop parameters by calling `textArea.getSelectedLines()`, which returns an array consisting of the line numbers of every selected line. The array includes the number of the current line, whether or not it is selected, and the line numbers are sorted in increasing order. We iterate through each member of the `selectedLines` array, which represents the number of a selected line, and apply the following routine:

- Get the buffer position of the start of the line (expressed as a zero-based index from the start of the buffer) by calling `textArea.getLineStartOffset(selectedLines[i]);`
- Move the caret to that position by calling `textArea.setCaretPosition();`
- Find the first and last non-whitespace characters on the line by calling `textArea.goToStartOfWhiteSpace()` and `textArea.goToEndOfWhiteSpace();`

  The `goTo...` methods in `JEditTextArea` take a single parameter which tells jEdit whether the text between the current caret position and the desired position should be selected. Here, we call `textArea.goToStartOfWhiteSpace(false)` so that no text is selected, then call `textArea.goToEndOfWhiteSpace(true)` so that all of the text between the beginning and ending whitespace is selected.

- Retrieve the selected text by storing the return value of `textArea.getSelectedText()` in a new variable `text`.

  If the line is empty, `getSelectedText()` will return `null`. In that case, we assign an empty string to `text` to avoid calling methods on a null object.

- Change the selected text to `prefix + text + suffix` by calling `textArea.setSelectedText()`. If there is no selected text (for example, if the line is empty), the prefix and suffix will be inserted without any intervening characters.

13.3.10. The Main Routine

```
// this single line of code is the script’s main routine
prefixSuffixDialog();
```

The call to `prefixSuffixDialog()`is the only line in the macro that is not inside an enclosing block. BeanShell treats such code as a top-level `main` method and begins execution with it.

Our analysis of `Add_Prefix_and_Suffix.bsh` is now complete. In the next section, we look at other ways in which a macro can obtain user input, as well as other macro writing techniques.

Chapter 14. Macro Tips and Techniques

14.1. Getting Input for a Macro

The dialog-based macro discussed in Chapter 13 reflects a conventional approach to obtaining input in a Java program. Nevertheless, it can be too lengthy or tedious for someone trying to write a macro quickly. Not every macro needs a user interface specified in such detail; some macros require only a single keystroke or no input at all. In this section we outline some other techniques for obtaining input that will help you write macros quickly.

14.1.1. Getting a Single Line of Text

As mentioned earlier in Section 12.2, the method `Macros.input()` offers a convenient way to obtain a single line of text input. Here is an example that inserts a pair of HTML markup tags specified by the user.

```
// Insert_Tag.bsh

void insertTag()
{
    caret = textArea.getCaretPosition();
    tag = Macros.input(view, “Enter name of tag:”);
    if( tag == null || tag.length() == 0) return;
    text = textArea.getSelectedText();
    if(text == null) text = “”;
    sb = new StringBuffer();
    sb.append(“<”).append(tag).append(“>”);
    sb.append(text);
    sb.append(“</”).append(tag).append(“>”);
    textArea.setSelectedText(sb.toString());
    if(text.length() == 0)
        textArea.setCaretPosition(caret + tag.length() + 2);
}
```

```
insertTag();

// end Insert_Tag.bsh
```

Here the call to `Macros.input()` seeks the name of the markup tag. This method sets the message box title to a fixed string, “Macro input”, but the specific message Enter name of tag provides all the information necessary. The return value `tag` must be tested to see if it is null. This would occur if the user presses the Cancel button or closes the dialog window displayed by `Macros.input()`.

14.1.2. Getting Multiple Data Items

If more than one item of input is needed, a succession of calls to `Macros.input()` is a possible, but awkward approach, because it would not be possible to correct early input after the corresponding message box is dismissed. Where more is required, but a full dialog layout is either unnecessary or too much work, the Java method `JOptionPane.showConfirmDialog()` is available. The version to use has the following prototype:

- `public static int` **`showConfirmDialog`**`(Component` *`parentComponent`*`, Object` *`message`*`, String` *`title`*`, int` *`optionType`*`, int` *`messageType`*`);`

The usefulness of this method arises from the fact that the `message` parameter can be an object of any Java class (since all classes are derived from `Object`), or any array of objects. The following example shows how this feature can be used.

```
// excerpt from Write_File_Header.bsh

title = “Write file header”;

currentName = buffer.getName();

nameField = new JTextField(currentName);
authorField = new JTextField(“Your name here”);
descField = new JTextField(“”, 25);
```

```
namePanel = new JPanel(new GridLayout(1, 2));
nameLabel = new JLabel(“Name of file:”, SwingConstants.LEFT);
nameLabel.setForeground(Color.black);
saveField = new JCheckBox(“Save file when done”,
    !buffer.isNewFile());
namePanel.add(nameLabel);
namePanel.add(saveField);


message = new Object[9];
message[0] = namePanel;
message[1] = nameField;
message[2] = Box.createVerticalStrut(10);
message[3] = “Author’s name:”;
message[4] = authorField;
message[5] = Box.createVerticalStrut(10);
message[6] = “Enter description:”;
message[7] = descField;
message[8] = Box.createVerticalStrut(5);

if( JOptionPane.OK_OPTION !=
    JOptionPane.showConfirmDialog(view, message, title,
        JOptionPane.OK_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE))
    return null;

// *****remainder of macro script omitted*****

// end excerpt from Write_File_Header.bsh
```

This macro takes several items of user input and produces a formatted file header at the begining of the buffer. The full macro is included in the set of macros installed by jEdit. There are a number of input features of this excerpt worth noting.

- The macro uses a total of seven visible components. Two of them are created behind the scenes by `showConfirmDialog()`, the rest are made by the macro. To arrange them, the script creates an array of `Object` objects and assigns components to each location in the array. This translates to a fixed, top-to-bottom arrangement in the message box created by `showConfirmDialog()`.

- The macro uses `JTextField` objects to obtain most of the input data. The fields `nameField` and `authorField` are created with constructors that take the initial, default text to be displayed in the field as a parameter. When the message box is displayed, the default text will appear and can be altered or deleted by the user.
- The text field `descField` uses an empty string for its initial value. The second parameter in its constructor sets the width of the text field component, expressed as the number of characters of "average" width. When `showConfirmDialog()` prepares the layout of the message box, it sets the width wide enough to accomodate the designated with of `descField`. This technique produces a message box and input text fields that are wide enough for your data with one line of code.
- The displayed message box includes a `JCheckBox` component that determines whether the buffer will be saved to disk immediately after the file header is written. To conserve space in the message box, we want to display the check box to the right of the label Name of file:. To do that, we create a `JPanel` object and populate it with the label and the checkbox in a left-to-right `GridLayout`. The `JPanel` containing the two components is then added to the beginning of `message` array.
- The two visible components created by `showConfirmDialog()` appear at positions 3 and 6 of the `message` array. Only the text is required; they are rendered as text labels. Note that the constructor sets the foreground color `nameLabel` to black. The default text color of `JLabel` objects is gray for Java's default look-and-feel, so the color was reset for consistency with the rest of the message box.
- There are three invisible components created by `showConfirmDialog()`. Each of them involves a call to `Box.createVerticalStrut()`. The `Box` class is a sophisticated layout class that gives the user great flexibility in sizing and positioning components. Here we use a `static` method of the `Box` class that produces a vertical *struct*. This is a transparent component whose width expands to fill its parent component (in this case, the message box). The single parameter indicates the fixed height of the spacing "strut" in pixels. The last call to `createVerticalStrut()` separates the description text field from the OK and Cancel buttons that are automatically added by `showConfirmDialog()`.

- Finally, the call to `showConfirmDialog()` uses defined constants for the option type and the message type. The option type signifies the use of OK and Cancel buttons. The `QUERY_MESSAGE` message type causes the message box to display a question mark icon.

  The return value of the method is tested against the value `OK_OPTION`. If the return value is something else (because the Cancel button was pressed or because the message box window was closed without a button press), a `null` value is returned to a calling function, signalling that the user cancelled macro execution. If the return value is `OK_OPTION`, each of the input components can yield their contents for further processing by calls to `JTextField.getText()` (or, in the case of the check box, `JCheckBox.isSelected()`).

14.1.3. Selecting Input From a List

Another useful way to get user input for a macro is to use a combo box containing a number of pre-set options. If this is the only input required, one of the versions of `showInputDialog()` in the `JOptionPane` class provides a shortcut. Here is its prototype:

- `public static Object` **`showInputDialog`**`(Component` *`parentComponent`*`, Object` *`message`*`, String` *`title`*`, int` *`messageType`*`, Icon` *`icon`*`, Object[]` *`selectionValues`*`, Object` *`initialSelectionValue`*`);`

This method creates a message box containing a drop-down list of the options specified in the method's parameters, along with OK and Cancel buttons. Compared to `showConfirmDialog()`, this method lacks an `optionType` parameter and has three additional parameters: an `icon` to display in the dialog (which can be set to `null`), an array of `selectionValues` objects, and a reference to one of the options as the `initialSelectionValue` to be displayed. In addition, instead of returning an int representing the user's action, `showInputDialog()` returns the `Object` corresponding to the user's selection, or `null` if the selection is cancelled.

The following macro fragment illustrates the use of this method.

```
// fragment illustrating use of showInputDialog()
options = new Object[5];
```

```
options[0] = "JLabel";
options[1] = "JTextField";
options[2] = "JCheckBox";
options[3] = "HistoryTextField";
options[4} = "- other -";

result = JOptionPane.showInputDialog(view,
    "Choose component class",
    "Select class for input component",
    JOptionPane.QUESTION_MESSAGE,
    null, options, options[0]);
```

The return value `result` will contain either the `String` object representing the selected text item or `null` representing no selection. Any further use of this fragment would have to test the value of `result` and likely exit from the macro if the value equalled `null`.

A set of options can be similarly placed in a `JComboBox` component created as part of a larger dialog or `showMessageDialog()` layout. Here are some code fragments showing this approach:

```
// fragments from Display_Abbreviations.bsh
// import statements and other code omitted

// from main routine, this method call returns an array
// of Strings representing the names of abbreviation sets

abbrevSets = getActiveSets();

...

// from showAbbrevs() method

combo = new JComboBox(abbrevSets);
// set width to uniform size regardless of combobox contents
Dimension dim = combo.getPreferredSize();
dim.width = Math.max(dim.width, 120);
combo.setPreferredSize(dim);
combo.setSelectedItem(STARTING_SET); // defined as "global"
```

```
// end fragments
```

14.1.4. Using a Single Keypress as Input

Some macros may choose to emulate the style of character-based text editors such as emacs or vi. They will require only a single keypress as input that would be handled by the macro but not displayed on the screen. If the keypress corresponds to a character value, jEdit can pass that value as a parameter to a BeanShell script.

The jEdit class `InputHandler` is an abstract class that that manages associations between keyboard input and editing actions, along with the recording of macros. Keyboard input in jEdit is normally managed by the derived class `DefaultInputHandler`. One of the methods in the `InputHandler` class handles input from a single keypress:

- `public void` **`readNextChar`**`(String` *`prompt`*`, String` *`code`*`);`

When this method is called, the contents of the `prompt` parameter is shown in the view's status bar. The method then waits for a key press, after which the contents of the `code` parameter will be run as a BeanShell script, with one important modification. Each time the string `__char__` appears in the parameter script, it will be substituted by the character pressed. The key press is "consumed" by `readNextChar()`. It will not be displayed on the screen or otherwise processed by jEdit.

Using `readNextChar()` requires a macro within the macro, formatted as a single, potentially lengthy string literal. The following macro illustrates this technique. It selects a line of text from the current caret position to the first occurrence of the character next typed by the user. If the character does not appear on the line, no new selection occurs and the display remains unchanged.

```
// Next_Char.bsh

script = new StringBuffer(512);
script.append( "start = textArea.getCaretPosition();"         );
script.append( "line = textArea.getCaretLine();"              );
script.append( "end = textArea.getLineEndOffset(line) + 1;"   );
script.append( "text = buffer.getText(start, end - start);"   );
script.append( "match = text.indexOf(__char__, 1);"           );
script.append( "if(match != -1) {"                            );
script.append(   "if(__char__ != '\\n') ++match;"             );
script.append(   "textArea.select(start, start + match - 1);" );
script.append( "}"                                            );

view.getInputHandler().readNextChar("Enter a character",
    script.toString());

// end Next_Char.bsh
```

Once again, here are a few comments on the macro's design.

- A `StringBuffer` object is used for efficiency; it obviates multiple creation of fixed-length `String` objects. The parameter to the constructor of `script` specifies the initial size of the buffer that will receive the contents of the child script.
- Besides the quoting of the script code, the formatting of the macro is entirely optional but (hopefully) makes it easier to read.
- It is important that the child script be self-contained. It does not run in the same namespace as the "parent" macro `Next_Char.bsh` and therefore does not share variables, methods, or scripted objects defined in the parent macro.
- Finally, access to the `InputHandler` object used by jEdit is available by calling `getInputHandler()` on the current view.

14.2. Startup Scripts

On startup, jEdit runs any BeanShell scripts located in the `startup` subdirectory of the jEdit installation and user settings directories (see Section 6.4). As with macros, the scripts must have a `.bsh` file name extension. Startup scripts are run near the end of the startup sequence, after plugins, properties and such have been initialized, but before the first view is opened.

Startup scripts can perform initialization tasks that cannot be handled by command line options or ordinary configuration options, such as customizing jEdit's user interface by changing entries in the Java platform's `UIManager` class.

Startup scripts have an additonal feature that can help you further customize jEdit. Unlike with macros, variables and methods defined in a startup script are available in all instances of the BeanShell interpreter created in jEdit. This allows you to create a personal library of methods and objects that can be accessed at any time during the editing session in another macro, the BeanShell shell of the Console plugin, or menu items such as Utilities>Evaluate BeanShell Expression.

The startup script routine will run script files in the installation directory first, followed by scripts in the user settings directory. In each case, scripts will be executed in alphabetical order, applied without regard to whether the file name contains upper or lower case characters.

If a startup script throws an exception (becuase, for example, it attempts to call a method on a `null` object). jEdit will show an error dialog box and move on to the next startup script. If script bugs are causing jEdit to crash or hang on startup, you can use the **-nostartupscripts** command line option to disable them for that editing session.

Another important difference between startup scripts and ordinary macros is that startup scripts cannot use the pre-defined variables `view`, `textArea`, `editPane` and `buffer`. This is because they are executed before the initial view is created.

If you are writing a method in a startup script and wish to use one of the above variables, pass parameters of the appropriate type to the method, so that a macro calling them after startup can supply the appropriate values. For example, a startup script could include a method

```
void doSomethingWithView(View v, String s)  {
    ...
}
```

so that during the editing session another macro can call the method using

```
doSomethingWithView(view, "something");
```

14.3. Running Scripts from the Command Line

The **-run** command line switch specifies a BeanShell script to run on startup:

```
$ jedit -run=test.bsh
```

Note that just like with startup scripts, the `view`, `textArea`, `editPane` and `buffer` variables are not defined.

If another instance is already running, the script will be run in that instance, and you will be able to use the `jEdit.getLastView()` method to obtain a view. However, if a new instance of jEdit is being started, the script will be run at the same time as all other startup scripts; that is, before the first view is opened.

If your script needs a view instance to operate on, you can use the following code snippet to obtain one, no matter how the script is being run:

```
void doSomethingUseful()
{
    void run()
    {
        view = jEdit.getLastView();

        // put actual script body here
    }

    if(jEdit.getLastView() == null)
        VFSManager.runInAWTThread(this);
    else
        run();
}

doSomethingUseful();
```

If the script is being run in a loaded instance, the `run()` method can be invoked directly. If the script is running on startup, a bit of magic has to be performed first. The method that does the script's work must be named `run()` so that the closure can implement the `Runnable` interface; this closure is then passed to the `runInAWTThread()` method.

When the `runInAWTThread()` method is invoked during startup, it schedules the specified `Runnable` to be run after startup is complete. If invoked when jEdit is fully loaded, the runnable will be run after all pending input/output is complete, or immediately if there are no pending I/O operations. Only the former behavior is useful in macros.

14.4. Advanced BeanShell Techniques

BeanShell has a few advanced features that we haven't mentioned yet. They will be discussed in this section.

14.4.1. BeanShell's Convenience Syntax

We noted earlier that BeanShell syntax does not require that variables be declared or defined with their type, and that variables that are not typed when first used can have values of differing types assigned to them. In addition to this "loose" syntax, BeanShell allows a "convenience" syntax for dealing with the properties of JavaBeans. They may be accessed or set as if they were data members. They may also be accessed using the name of the property enclosed in quotation marks and curly brackets. For example, the following statement are all equivalent, assuming `btn` is a `JButton` instance:

```
b.setText("Choose");
b.text = "Choose";
b{"text"} = "Choose";
```

The last form can also be used to access a key-value pair of a `Hashtable` object. It can even be used to obtain the values of buffer-local properties; the following two statements are equivalent:

```
buffer.getProperty("tabSize")
buffer{"tabSize"}
```

14.4.2. Special BeanShell Keywords

BeanShell uses special keywords to refer to variables or methods defined in the current or an enclosing block's scope:

- The keyword `this` refers to the current scope.
- The keyword `super` refers to the immediately enclosing scope.
- The keyword `global` refers to the top-level scope of the macro script.

The following script illustrates the use of these keywords:

```
a = "top\n";
foo() {
    a = "middle\n";
    bar() {
        a = "bottom\n";
        textArea.setSelectedText(global.a);
        textArea.setSelectedText(super.a);
        // equivalent to textArea.setSelectedText(this.a):
        textArea.setSelectedText(a);
    }

    bar();
}
foo();
```

When the script is run, the following text is inserted in the current buffer:

```
top
middle
bottom
```

14.4.3. Implementing Interfaces

As discussed in the macro example in Chapter 13, scripted objects can implement Java interfaces such as `ActionListener`. Which interfaces may be implemented varies depending upon the version of the Java runtime environment being used. If running under Java 1.1 or 1.2, BeanShell objects can only implement the AWT or Swing event listener interfaces contained in the `java.awt.event` and `javax.swing.event` packages, and the `java.lang.Runnable` interface. If running under Java 1.3 or 1.4, any interface can be implemented.

Frequently it will not be necessary to implement all of the methods of a particular interface in order to specify the behavior of a scripted object. Under Java 1.2 and below, BeanShell will automatically ignore calls on unimplemented members of an interface. Under Java 1.3 and above, however, the reflection mechanism will throw an exception for any missing interface methods, which will result in an error dialog box being shown when your macro runs; not a pretty sight. The solution is to implement the `invoke()` method, which is called when an undefined method is invoked on a scripted object. Typically, the implementation of this method will do nothing, as in the following example:

```
invoke(method, args) {}
```

14.4.4. BeanShell Commands

BeanShell comes with a large number of built-in scripted “commands” that are useful in many circumstances. Documentation for commands that are helpful when writing macros can be found in Chapter 19.

14.5. Debugging Macros

Here are a few techniques that can prove helpful in debugging macros.

14.5.1. Identifying Exceptions

An *exception* is a condition reflecting an error or other unusual result of program execution that requires interruption of normal program flow and some kind of special handling. Java has a rich (and extendable) collection of exception classes which represent such conditions.

jEdit catches exceptions thrown by BeanShell scripts and displays them in a dialog box. In addition, the full traceback is written to the activity log (see Appendix B for more information about the activity log).

There are two broad categories of errors that will result in exceptions:

- *Interpreter errors*, which may arise from typing mistakes like mismatched brackets or missing semicolons, or from BeanShell's failure to find a class corresponding to a particular variable.

  Interpreter errors are usually accompanied by the line number in the script, along with the cause of the error.

- *Execution errors*, which result from runtime exceptions thrown by the Java platform when macro code is executed.

  Some exceptions thrown by the Java platform can often seem cryptic. Nevertheless, examining the contents of the activity log may reveals clues as to the cause of the error.

14.5.2. Using the Activity Log as a Tracing Tool

Sometimes exception tracebacks will say what kind of error occurred but not where it arose in the script. In those cases, you can insert calls that log messages to the activity log in your macro. If the logged messages appear when the macro is run, it means that up to that point the macro is fine; but if an exception is logged first, it means the logging call is located after the cause of the error.

To write a message to the activity log, use the following method of the `Log` class:

- `public static void` **`log`**`(int` *`urgency`*`, Object` *`source`*`, Object` *`message`*`);`

The parameter `urgency` can take one of the following constant values:

- `Log.DEBUG`
- `Log.MESSAGE`

- `Log.NOTICE`
- `Log.WARNING`
- `Log.ERROR`

Note that the `urgency` parameter merely changes the string prefixed to the log message; it does not change the logging behavior in any other way.

The parameter `source` can be either an object or a class instance. When writing log messages from macros, set this parameter to `BeanShell.class` to make macro errors easier to spot in the activity log.

The following code sends a typical debugging message to the activity log:

```
Log.log(Log.DEBUG, BeanShell.class,
    "counter = " + String.valueOf(counter));
```

The corresponding activity log entry might read as follows:

```
[debug] BeanShell: counter = 15
```

Using message dialog boxes as a tracing tool

If you would prefer not having to deal with the activity log, you can use the `Macros.message()` method as a tracing tool. Just insert calls like the following in the macro code:

```
Macros.message(view,"tracing");
```

Execution of the macro is halted until the message dialog box is closed.

IV. Writing Plugins

This part of the user’s guide covers writing plugins for jEdit.

Like jEdit itself, plugins are written primarily in Java. While this guide assumes some working knowledge of the language, you are not required to be a Java wizard. If you can write a useful application of any size in Java, you can write a plugin.

This part of the user’s guide was written by John Gellene <`jgellene@nyc.rr.com`>.

Chapter 15. Introducing the Plugin API

The *jEdit Plugin API* provides a framework for hosting plugin applications without imposing any requirements on the design or function of the plugin itself. You could write a application that performs spell checking, displays a clock or plays chess and turn it into a jEdit plugin. There are currently over 40 released plugins for jEdit. While none of them play chess, they perform a wide variety of editing and file management tasks. A detailed listing of available plugins is available at the jEdit Plugin Central (http://plugins.jedit.org) web site.

Using the plugin manager feature of jEdit, users with an Internet connnection can check for new or updated plugins and install and remove them without leaving jEdit. See Chapter 8 for details.

In order to “plug in” to jEdit, a plugin must implement interfaces that deal with the following matters:

- Ths plugin must supply information about itself, such as its name, version, author, and compatibility with versions of jEdit.
- The plugin must provide for activating, displaying and deactivating itself upon direction from jEdit, typically in response to user input.
- The plugin may, but need not, provide a user interface.

  If the plugin has a visible interface, it can be shown in any object derived from one of Java top-level container classes: `JWindow`, `JDialog`, or `JFrame`. jEdit also provides a dockable window API, which allows plugin windows to be docked into views or shown in top-level frames, at the user’s request.

  Plugins can also act directly upon jEdit’s text area. They can add graphical elements to the text display (like error highlighting in the case of the ErrorList plugin) or decorations surrounding the text area (like the JDiff plugin’s summary views).
- Plugins may (and typically do) define *actions* that jEdit will perform on behalf of the plugin upon user request. Actions are short snippets of BeanShell code that provide the

"glue" between user input and specifc plugin routines.

  By convention, plugins display their available actions in submenus of jEdit's Plugins menu; each menu item corresponds to an action. The user can also assign actions to keyboard shortcuts, toolbar buttons or entries in the text area's right-click menu.

- Plugins may provide a range of options that the user can modify to alter its configuration.

  If a plugin provides configuration options in accordance with the plugin API, jEdit will make them available in the Global Options dialog. Each plugin with options is listed in the tree view in that dialog under Plugin Options. Clicking on the tree node for a plugin causes the corresponding set of options to be displayed.

As noted, many of these features are optional; it is possible to write a plugin that does not provide actions, configuration options, or dockable windows. The majority of plugins, however, provide most of these services.

In the following chapters, we will begin by briefly describing jEdit's host capabilities, which includes the loading and display of plugins. Next we will describe the principal classes and data structures that a plugin must implement. Finally, we will outline the building of a modest plugin, "QuickNotepad", that illustrates the requirements and some of the techniques of jEdit plugin design.

Plugins and different jEdit versions

As jEdit continues to evolve and improve, elements of the plugin API or jEdit's general API may change with a new jEdit release. For example, version 3.2 of jEdit introduced a set of `Selection` classes that enable multiple text selections in the text area. On occasion an API change will break code used by plugins, although efforts are made to maintain or deprecate plugin-related code where possible. While the majority of plugins are unaffected by most changes and will continue working, it is a good idea to monitor the jEdit change log and mailing lists for API changes and update your plugin as necessary.

Chapter 16. jEdit as a Plugin Host

A good way to start learning what a plugin requires is to look at what the host application does with one. We start our discussion of plugins by outlining how jEdit loads and displays them. This section only provides a broad overview of the more important components that make up jEdit; specifics of the API will be documented in subsequent chapters.

16.1. Loading Plugins

As part of its startup routine, jEdit's `main` method calls various methods to load and initialize plugins. This occurs after the application has done the following:

- parsed command line options;
- started the edit server (unless instructed not to by a command line option) and;
- loaded application properties, any user-supplied properties, and the application's set of actions that will be available from jEdit's menu bar (as well as the toolbar and keyboard shortcuts);

Plugin loading occurs before jEdit creates any windows or loads any files for editing. It also occurs before jEdit runs any startup scripts.

16.1.1. The JARClassLoader

Plugins are loaded from files with the `.jar` filename extension located in the `jars` subdirectories of the jEdit installation and user settings directories (see Section 6.4).

For each JAR archive file it finds, jEdit creates an instance of the `JARClassLoader` class. This is a jEdit-specific class that implements the Java platform's abstract class `ClassLoader`. The constructor for the `JARClassLoader` object does the following:

- Adds any class file with a name ending with `Plugin.class` to an internal collection of plugin class names maintained by the `JARClassLoader`. See Section 17.1.

- Loads any properties defined in files ending with the extension `.props` that are contained in the archive. See Section 17.4.2.
- Loads any data on the plugin's actions from a file named `actions.xml` (if it exists) contained at the top level of the archive file. See Section 17.4.1.
- Adds to a collection maintained by jEdit a new object of type `EditPlugin.JAR`. This is a data structure holding the name of the jar archive file, a reference to the `JARClassLoader` and a collection, initially empty, of plugins found in the archive file.

Once all plugin JAR files have been examined for the above resources, jEdit initializes all class files whose names end in `Plugin.class`, as identified in the first pass through the JAR archive. We will call these classes *plugin core classes*. Plugin core classes are the principal point of contact between jEdit and the plugin, and must extend jEdit's abstract `EditPlugin` class.

For each plugin core class, the loader first checks the plugin's properties to see if it is subject to any dependencies. For example, a plugin may require that the version of the Java runtime environment or of jEdit itself be equal to or above some threshold version. A plugin can also require the presence of another plugin. If any dependency is not satisified, the loader marks the plugin as "broken" and logs an error message.

If all dependencies are satisfied, a new instance of the plugin core class is created and added to the collection maintained by the appropriate `EditPlugin.JAR` object. By accessing that object, jEdit can keep track of plugins it has successfully loaded, and call methods or perform routines on them.

Class libraries

JAR files with no plugin core classes are also loaded by jEdit; no special initialization is performed on them, and the classes they contain are made available to other plugins.

Many plugins that rely on third-party class libraries ship them as separate JARs, for example.

A plugin that bundles extra JARs needs to define a property that lists these JAR files in order for the plugin manager to be able to remove the plugin completely. See Section 17.4.2.

16.1.2. Starting the Plugin

After creating and storing the plugin core object, jEdit calls the `start()` method of the plugin core class. This method is defined as an empty “no-op” in the `EditPlugin` abstract class, therefore it is not required that plugins provide their own implementation. Only trivial plugins will not need to perform some kind of initialization, however.

The `start()` method can perform initialization of the object’s data members. It can also register its identity and other information with jEdit’s *EditBus* object, which manages messaging between plugins and the host application. We will discuss the EditBus in more detail in Section 16.2.2 and Chapter 21.

At this point, we can identify the following practical requirements for a plugin:

- it must be packaged as a JAR archive;
- the JAR archive must contain at least one plugin core class whose name ends in `Plugin`;
- each plugin core class must exnted the `EditPlugin` abstract class;
- the JAR archive may contain data concerning actions for display in jEdit’s menu bar and elsewhere in a file entitled `actions.xml`; and

- the archive must contain at least one properties file having a `.props` extension. Certain properties giving information about the plugin must be defined.

We will provide more detail on these requirements later.

16.2. The User Interface of a Plugin

To display a user interface, plugins can either directly extend Java's `JFrame`, `JDialog`, or `JWindow` classes, or use jEdit's dockable window API. Plugin windows are typically defined in classes that are part of the plugin package but separate from the plugin core class.

16.2.1. The Role of the View Object

A `View` is jEdit's top-level frame window that contains one or more (if the view is split) text areas, a menu bar, a toolbar and other window decorations, as well as docked plugin windows. The `View` class performs two important operations that deal with plugins: creating plugin menu items, and managing dockable windows.

When a view is being created, it iterates through the collection of loaded plugins and calls the `createMenuItems()` method of each plugin core class. Again, implementing this method is not necessary, but very few plugins will be able to get away with not adding anything to jEdit's menu bar. As we will explain in the next chapter, the typical plugin, instead of creating Java `JMenu` and `JMenuItem` objects directly, relies on a method in a utility class to create menu entries.

The `View` also creates and initializes a `DockableWindowManager` object. This object is responsible for creating, closing and managing dockable windows.

The `View` class contains a number of methods that can be called from plugins; see Section 20.2 for details.

16.2.2. The DockableWindowManager and the EditBus

The `DockableWindowManager` keeps track of docked and floating windows. When the `View` object initializes its `DockableWindowManager`, the manager iterates through the list of registered dockable windows and causes those specified by the user to “auto open” in the Global Options dialog box to be displayed. The `DockableWindowManager` class is also invoked at any other time the user requests a dockable window is opened or closed.

The `DockableWindowManager` creates and displays plugin windows by routing messages through the application’s `EditBus` object that we mentioned earlier. The EditBus mantains a list of objects that have requested to receive messages. When a message is sent using this class, all registered components receive it in turn.

EditBus subscribers must implement the `EBComponent` interface, which defines the single method `handleMessage()`. A `View`, for example, can receive and handle EditBus messages because it also implements `EBComponent`.

Plugins that wish to receive messages can explicitly provide implementations of this interface, and register them with the EditBus using the `EditBus.addToBus()` method. However, it can be more convinient to have the plugin core class extend the `EBPlugin` abstract class, which is identical to the `EditPlugin` class, except it implements the interface mentioned above, and automatically adds itself to the EditBus.

To activate a plugin window, the `DockableWindowManager` creates a `CreateDockableWindow` message object containing three data items: a reference to the view that will contain the plugin, the name of the plugin and the relative position of the window in which the plugin will be placed. That message is sent to the EditBus using the `send()` method of the `EditBus` class.

16.2.3. Message Routing and Dockable Window Creation

In the case of a `CreateDockableWindow` message, successive subscribers to the EditBus receive the message, through a call to each subscriber’s `handleMessage()` method, until one subscriber signals that it has handled the message. This occurs when a subscriber matches the message’s *`name`* data member with the name of a dockable window the plugin provides. The intended recipient then handles the message by creating an appropriate plugin window and attaching it to the message, so that when message routing is completed, the `DockableWindowManager` can retrieve and store the new plugin window.

As a final step in plugin activation, the manager create another window object that will contain the visible components of the plugin. This object implements the `DockableWindowContainer` interface; depending on the settings for the plugin selected by the user, it will either be a tabbed window pane in one of the docked windows attached to the `View` object, or a separate, floating frame window. Plugins need not be aware of the implementation details of the container.

Eventually the `DockableWindowManager` destroys the plugin window, whether docking or floating, in response to user action or as part of the destruction of the corresponding `View` object.

The `DockableWindowManager` and `EditBus` classes contain a number of methods that can be called from plugins; see Section 20.2 for details.

This summary shows that a plugin wishing to use the dockable window API has the following additional requirements:

- the plugin class must extend `EBPlugin` instead of `EditPlugin` in order to receive the `CreateDockableWindow` message;
- it must register its dockable windows in its `start()` method; and
- it must create and arrange any dockable windows it provides in response to the appropriate `CreateDockableWindow` message;

With this broad outline of how jEdit behaves as a plugin host in the background, we will next review the programming elements that make up a plugin.

Chapter 17. The jEdit Plugin API

17.1. Plugin Core Classes

As mentioned earlier, a plugin must provide a “plugin core class”, otherwise it will not do anything useful (but recall that a class library intended for use by other plugins need not provide a plugin core class). That class must extend either `EditPlugin` or its convinience subclass, `EBPlugin`. We begin our review of the jEdit plugin API with these two classes.

17.1.1. Class EditPlugin

This abstract class is the base for every plugin core class. Its methods provide for basic interaction between the plugin and jEdit. The class has four methods which are called by jEdit at various times. None of these methods are required to be implemented, but most plugins will override at least one.

- `public void` **`start`**`();`

  The jEdit startup routine calls this method for each loaded plugin. Plugins typically use this method to register information with the EditBus and perform other initialization.

- `public void` **`stop`**`();`

  When jEdit is exiting, it calls this method on each plugin. If a plugin uses or creates state information or other persistent data that should be stored in a special format, this would be a good place to write the data to storage. Note that most plugins will use jEdit’s properties API to save settings, and the persistance of properties is handled automatically by jEdit and requires no special processing in the `stop()` method.

- `public void` **`createMenuItems`**`(Vector` *`menuItems`*`);`

  When a `View` object is created, it calls this method on each plugin to obtain entries to be displayed in the view’s Plugins menu. The *`menuItems`* parameter is a `Vector` that accumilates menu items and menus as it is passed from plugin to plugin.

jEdit does not require a plugin to supply menu items. If menu items are desired, the easiest way to provide for them is to package the desired menu items as entries in the plugin's property file and implement `createMenuItems()` with a call to jEdit's `GUIUtilities.loadMenu()` method; for example:

```
public void createMenuItems(Vector menuItems)
{
    menuItems.addElement(GUIUtilities.loadMenu(
        "myplugin.menu"));
}
```

The parameter passed to `loadMenu()` is the name of a property containing menu data. We will explain the format of the menu data in Section 18.2.3.2

The `GUIUtilities.loadMenuItem()` method is also available for plugins that only wish to add a single menu item to the Plugins menu.

- `public void` **`createOptionPanes`**`(OptionsDialog` *`dialog`*`);`

  This method is called for each plugin during the creation of the Global Options dialog box. To show an option pane, the plugin should define an option pane class and implement `createOptionPane()` as follows:

  ```
  dialog.addOptionPane(new MyPluginOptionPane());
  ```

  Plugins can also define more than one option pane, grouped in an "option group". We will discuss the design and elements of the option pane API in Section 17.3.

This class defines two other methods which may be useful to some plugins, but are mainly of use to the jEdit core:

- `public String` **`getClassName`**

  This shortcut method returns `getClass().getName()`.

- `public EditPlugin.JAR` **`getJAR`**

  This method returns the `EditPlugin.JAR` data object associated with the plugin.

17.1.2. Class EBPlugin

Every plugin core class class that uses the EditBus for receiving messages must extend this class. This class implements the `EBComponent` interface, required for any object that wishes to receive EditBus messages.

The `EBComponent` interface contains a single method that an implementing class (including any class derived from `EBPlugin`) must provide:

- `public void` **`handleMessage`**`(EBMessage` *`message`*`);`

The parameter's type, `EBMessage`, is another abstract class which establishes the core elements of any message that is published to the EditBus. It has two attributes: an `EBComponent` that is the source of the message (the source will be null in some cases), and a boolean data member, `vetoed`. This flag indicates whether a prior recipient of the message has determined that the message has been handled and need not be passed on to other subscribers. The flag is set by a call to the `veto()` method of the `EBMessage`. Some message classes, however, are configured so that they cannot be vetoed, to ensure they are received by all subscribers.

Message classes extending `EBMessage` typically add other data members and methods to provide subscribers with whatever is needed to handle the message appropriately. Descriptions of specific message classes can be found in Chapter 21.

The `handleMessage()` method must specify the type of responses the plugin will have for various subclasses of the `EBMessage` class. Typically this is done with one or more `if` blocks that test whether the message is an instance of a derived message class in which the plugin has an interest, for example like so:

```
if(msg instanceof CreateDockableWindow)
    // create dockable window, if necessary
else if(msg instanceof BufferUpdate)
    // a buffer’s state has changed!
else if(msg instanceof ViewUpdate)
    // a view’s state has changed!
// ... and so on
```

If a plugin defines dockable windows, it should respond to a `CreateDockableWindow` message by creating the appropriate user interface objects and setting the relevant data field in the message, for example like so:

```
if(msg instanceof CreateDockableWindow)
{
    CreateDockableWindow cmsg = (CreateDockableWindow)msg;
    if(cmsg.getDockableWindowName().equals("myplugin"))
        cmsg.setDockableWindow(new MyPluginWindow());
}
```

Note that any object, whether or not derived from `EBComponent`, can send a message to the EditBus by calling the static method `EditBus.send()`. This method takes a single parameter, an `EBMessage` object that is the message being sent. Most plugins, however, will only concern themselves with receiving, not sending, messages.

17.2. Interface DockableWindow

The dockable plugin API consists of a single interface, `DockableWindow`. It links the visible components of a plugin with the dockable window management facility. The interface gives developers flexibility and minimizes code refactoring, for it can be implemented as part of the plugin's top-level display window or in a separate lightweight class. The dockable window API handles the display of windows as either docked or floating without specific direction from the plugin.

This interface provides the connection between the plugin's visible components and a top-level `View` object of the host application. As mentioned earlier, the plugin window class implementing this interface must be created by the plugin core class in response to a `CreateDockableWindow` message. After its creation, the plugin window object is attached to the message for routing back to jEdit.

The `DockableWindow` interface contains two methods that must be implemented by a derived plugin window class:

- `String` **`getName`**`();`

  This method should return the internal working name of the plugin window, used to key various properties.

- `Component` **`getComponent`**`();`

  This method should return the top-level visible component of the plugin. Typically this component is a `JPanel` containing other components, but any object derived from the Java `Component` class will suffice. If the top-level component implements the `DockableWindow` interface, so that the plugin window and the top-level visible window are implemented in the same class, the implementation of `getComponent()` would simply return `this`.

17.3. Plugin Option Pane Classes

The plugin API provides a mechanism for displaying a plugin's configuration options in the Global Options dialog. A plugin that allows user configuration should provide one or more implementations of jEdit's `OptionPane` interface to have configuration options displayed in a manner consistent wth the rest of the application.

17.3.1. Class AbstractOptionPane

Most plugin option panes extend this implementation of `OptionPane`, instead of implementing `OptionPane` directly. It provides a convenient default framework for laying out configuration options in a manner similar to the option panes created by jEdit itself. It is derived from Java's `JPanel` class and contains a `GridBagLayout` object for component management. It also contains shortcut methods to simplify layout.

The constructor for a class derived from `AbstractOptionPane` should call the parent constructor and pass the option pane's "internal name" as a parameter. The internal name is used to key a property where the option pane's label is stored; see Section 17.4.2. It should also implement two methods:

- `protected void` **`_init`**`();`

  This method should create and arrange the components of the option pane and initialize the option data displayed to the user. This method is called when the option pane is first displayed, and is not called again for the lifetime of the object.

- `protected void` **`_save`**`();`

  This method should save any settings, to the jEdit properties or other data store.

`AbstractOptionPane` also contains three shortcut methods, typically called from `_init()`, for adding components to the option pane:

- `protected void` **`addComponent`**`(String` *`label`*`, Component` *`comp`*`);`
- `protected void` **`addComponent`**`(Component` *`comp`*`);`

  These shortcut methods add components to the option pane in a single vertical column, running top to bottom. The first displays the text of the *`label`* parameter to the left of the `Component` represented by *`comp`*.

- `protected void` **`addSeparator`**`(String` *`label`*`);`

  This is another shortcut method that adds a text label between two horizontal separators to the option pane. The *`label`* parameter represents the name of a property (typically a property defined in the plugin's property file) whose value will be used as the separator text.

17.3.2. Class OptionGroup

In those cases where a single option pane is inadequate to present all of a plugin's configuration options, this class can be used to create a group of options panes. The group will appear as a single node in the options dialog tree-based index. The member option panes will appear as leaf nodes under the group's node. Threee simple methods create and populate an option pane:

- `public` **`OptionGroup`**`(String` *`name`*`);`

  The constructor's single parameter represents the internal name of the option group. The internal name is used to key a property where the option group's label is stored; see Section 17.4.2.

- `public void` **`addOptionPane`**`(OptionPane` *`pane`*`);`
- `public void` **`addOptionGroup`**`(OptionGroup` *`group`*`);`

  This pair of methods adds members to the option group. The second method enables option groups to be nested, for plugins with a particularly large set of configurable options.

17.4. Other Plugin Resources

There are three other types of files containing resources used by a plugin:

- a catalog of the plugin's user actions in a specified XML format, contained in a file named `actions.xml`;
- one or more properties files named with a `.props` extension, each containing key-value pairs in conventional Java format; and
- a help file written in HTML format. The name of this file must be specified in a property; see Section 17.4.2.

17.4.1. The Action Catalog

Actions define procedures that can be bound to a menu item, a toolbar button or a keyboard shortcut. They can perform any task encompassed in a public method of any class currently loaded in jEdit, including plugin classes and classes of the host application. Among other things, they can cause the appearance and disappearance of plugin windows.

To manage user actions, jEdit maintains a lookup table of actions using descriptive strings as keys. The values in the table are sets of statements written in BeanShell, jEdit's macro scripting language. These scripts either direct the action themselves, delegate to a method in one of the plugin's classes that encapsulates the action, or do a little of both. The scripts are usually short; elaborate action protocols are usually contained in compiled code, rather than an interpreted macro script, to speed execution.

Actions are defined by creating an XML file entitled `actions.xml` at the top level of the plugin JAR file. A sample action catalog looks like so:

```
<!DOCTYPE ACTIONS SYSTEM "actions.dtd">

<ACTIONS>
    <ACTION NAME="quicknotepad.toggle">
        <CODE>
            view.getDockableWindowManager()
                .toggleDockableWindow(QuickNotepadPlugin.NAME);
        </CODE>
        <IS_SELECTED>
            return view.getDockableWindowManager()
                .isDockableWindowVisible(QuickNotepadPlugin.NAME);
        </IS_SELECTED>
    </ACTION>

    <ACTION NAME="quicknotepad-to-front">
    <CODE>
      view.getDockableWindowManager()
                .addDockableWindow(QuickNotepadPlugin.NAME);
        </CODE>
    </ACTION>
</ACTIONS>
```

The defined elements have the following functions:

- `ACTIONS` is the top-level element and refers to the set of actions used by the plugin.
- An `ACTION` contains the data for a particular action. It has three attributes: a required `NAME`; an optional `NO_REPEAT`, which is a flag indicating whether the action should not

be repeated with the **Control-Enter** command (see Section 2.4); and an optional `NO_RECORD` which is a a flag indicating whether the action should be recorded if it is invoked while a user is recording a macro. The two flag attributes can have two possible values, “TRUE” or “FALSE”. In both cases, “FALSE” is the default if the attribute is not specified.

- An `ACTION` can have two child elements within it: a required `CODE` element which specifies the BeanShell code that will be executed when the action is invoked, and an optional `IS_SELECTED` element, used for checkbox menu items. The `IS_SELECTED` element contains BeanShell code that returns a boolean flag that will determine the state of the checkbox.

More discussion of the action catalog can be found in Section 18.2.3.1.

17.4.2. Plugin Properties

jEdit maintains a list of “properties”, which are name/value pairs used to store human-readable strings, user settings, and various other forms of meta-data. During startup, jEdit loads the default set of properties, followed by plugin properties stored in plugin JAR files, finally followed by user properties. Plugins can access properties from all three sources.

Property files contained in plugin JARs must end with the filename extension `.props`, and have a very simple syntax, which the following example suffices to describe:

```
# Lines starting with '#' are ignored.
name=value
another.name=another value
long.property=Long property value, split over \
    several lines
escape.property=Newlines and tabs can be inserted \
    using the \t and \n escapes
backslash.property=A backslash can be inserted by writing \\.
```

The following types of plugin information are supplied using properties:

- Information regarding the name, author, and version of the plugin. This information is required. Here is an example:

  ```
  plugin.MyPlugin.name=My Plugin
  plugin.MyPlugin.author=Joe Random Hacker
  plugin.MyPlugin.version=1.0.3
  ```

  Note that each property is prefixed with `plugin.`, followed by the fully qualified name of the plugin core class (including a package name, if there is one).

- Identification of any dependencies the plugin may have on a particular version of a Java runtime environment, the jEdit application, or other plugins.

  Each dependency is defined in a property prefixed with `plugin.`*`class name`*`.depend.`, followed by a number. Dependencies must be numbered in order, starting from zero.

  The value of a dependency property is one of the words `jdk`, `jedit`, `class` or `plugin`, followed by a Java version number, a jEdit build number, a class name, or plugin class name and plugin version number, respectively.

  Here are some examples:

  ```
  plugin.MyPlugin.depend.0=jdk 1.2
  plugin.MyPlugin.depend.1=jedit 03.02.97.00
  plugin.MyPlugin.depend.2=class com.ice.tar.tar
  plugin.MyPlugin.depend.3=plugin console.ConsolePlugin 3.0
  ```

- A list of external class library JARs shipped with the plugin. If your plugin bundles extra JARs, this property is required for the plugin manager to be able to remove the plugin completely.

  The property is a space-separated list of filenames. Here is an example:

  ```
  plugin.AntFarmPlugin.jars=crimson.jar jaxp.jar
  ```

- The titles of dockable windows, as displayed in a tabbed or floating container.

These labels are specified in properties named by the return value of the dockable window's `getName()` method, suffixed with `.title`. For example:

```
quick-notepad.title=QuickNotepad
```

- Labels for user actions for inclusion in menus and option panes relating to toolbars and keyboard shortcuts.

  Action labels are defined in properties named by the action's internal name as specified in the action catalog, followed by `.label`:

  ```
  myplugin.label=My Plugin
  myplugin-grok.label=Grok Current Buffer
  ```

- The list of menu items contained in plugin menus, if any.

  This is discussed in detail in Section 18.2.3.2.

- Labels and other information regarding the controls contained in the plugin's windows. These properties can be named any way you like, however take care not to choose names which may conflict with those in other plugins.

17.4.3. Plugin Documentation

While not required by the plugin API, a help file is an essential element of any plugin written for public release. A single web page is often all that is required. There are no specific requirements on layout, but because of the design of jEdit's help viewer, the use of frames should be avoided. Topics that would be useful include the following:

- a description of the purpose of the plugin;
- an explanation of the type of input the user can supply through its visible interface (such as mouse action or text entry in controls);
- a listing of available user actions that can be taken when the plugin does not have input focus;

- a summary of configuration options;
- information on development of the plugin (such as a change log, a list of “to do” items, and contact information for the plugin’s author); and
- licensing information, including acknowledgements for any library software used by the plugin.

The location of the plugin’s help file should be stored in the `plugin.`*`class name`*`.docs` property.

Chapter 18. Writing a Plugin

One way to organize a plugin project is to design the software as if it were a “stand alone” application, with three exceptions:

- The plugin can access the `View` object with which it is associated, as well as static methods of the `jEdit` class, to obtain and manipulate various data and host application objects;
- If the plugin has visible components, they are ultimately contained in a `JPanel` object instead of a top-level frame window; and
- The plugin implements the necessary elements of the jEdit plugin API that were outlined in the last chapter: a plugin core class, perhaps a number of plugin window classes, maybe a plugin option pane class, and a set of required plugin resources.

  Not every plugin has configurable options; some do not have a visible window. However, all will need a plugin core class and a minimum set of other resources.

We will now illustrate this approach by introducing an example plugin.

18.1. QuickNotepad: An Example Plugin

There are many applications for the leading operating systems that provide a “scratch-pad” or “sticky note” facility for the desktop display. A similar type of facility operating within the jEdit display would be a convenience. The use of docking windows would allow the notepad to be displayed or hidden with a single mouse click or keypress (if a keyboard shortcut were defined). The contents of the notepad could be saved at program exit (or, if earlier, deactivation of the plugin) and retrieved at program startup or plugin activation.

We will keep the capabilities of this plugin modest, but a few other features would be worthwhile. The user should be able to write the contents of the notepad to storage on demand. It should also be possible to choose the name and location of the file that will be used to hold the notepad text. This would allow the user to load other files into the notepad display. The path of the notepad file should be displayed in the plugin window, but will give the user the option to hide the file name. Finally, there should be an action by which a single click or keypress would cause the contents of the notepad to be written to the new text buffer for further processing.

The full source code for QuickNotepad is contained in jEdit's source code distribution. We will provide excerpts in this discussion where it is helpful to illustrate specific points. You are invited to obtain the source code for further study or to use as a starting point for your own plugin.

18.2. Writing a Plugin Core Class

The major issues encountered when writing a plugin core class arise from the developer's decisions on what features the plugin will make available. These issues have implications for other plugin elements as well.

- Will the plugin provide for actions that the user can trigger using jEdit's menu items, toolbar buttons and keyboard shortcuts?
- Will the plugin have its own visible interface?
- Will the plugin use jEdit's dockable window API?

  If a plugin will use the dockable window API, it must handle a targeted `CreateDockableWindow` message.
- Will the plugin respond to any other messages reflecting changes in the host application's state?
- Will the plugin have settings that the user can configure?

18.2.1. Choosing a Base Class

If the plugin will respond to EditBus messages, it must be derived from `EBPlugin`. Otherwise, `EditPlugin` will suffice as a base class.

Knowing what types of messages are made available by the plugin API is obviously helpful is determining both the base plugin class and the contents of a `handleMessage()` method. The message classes derived from `EBMessage` cover the opening and closing of the application, changes in the status of text buffers and their container and changes in user settings, as well as changes in the state of other program features. Specific message classes of potential interest to a plugin include the following:

- `EditorStarted`, sent during the application's startup routine, just prior to the creation of the initial `View`;
- `EditorExitRequested`, sent when a request to exit has been made, but before saving open buffers and writing user settings to storage;
- `EditorExiting`, sent just before jEdit actually exits;
- `EditPaneUpdate`, sent when an edit pane containing a text area (including a pane created by splitting an existing window) has been created or destroyed, or when a buffer displayed in an edit pane has been changed;
- `BufferUpdate`, sent when a text buffer is created, loaded, or being saved, or when its editing mode or markers have changed;
- `ViewUpdate`, sent when a `View` is created or closed; and
- `PropertiesChanged`, sent when the properties of the application or a plugin has been changed through the General Options dialog;

Detailed documentation for each message class can be found in Chapter 21.

18.2.2. Implementing Base Class Methods

18.2.2.1. General Considerations

If `EditPlugin` is selected as the base plugin core class, the implementations of `start()` and `stop()` in the plugin's derived class are likely to be trivial, or not present at all (in which case they will be "no-ops"). If `EBPlugin` is selected to provide messaging capability, however, there are a few fixed requirements.

If the plugin is to use the dockable window API, the “ internal names” of any dockable windows must be registered with the EditBus component. The EditBus stores such information in one of a number of “named lists”. Here is how the QuickNotepad plugin registers its dockable window:

```
EditBus.addToNamedList(DockableWindow.DOCKABLE_WINDOW_LIST, NAME);
```

The first parameter is a `String` constant identifying the dockable window list. The second is a static `String` constant which is initialized in the plugin core class as the dockable window’s internal name.

The use of `NAME` as the second parameter employs an idiom found in many plugins. The plugin class can include `static final String` data members containing information to be registered with the EditBus or key names for certain types of plugin properties. This makes it easier to refer to the information when a method such as `handleMessage()` examines the contents of a message. The kind of data that can be handled in this fashion include the following:

- the internal working name of dockable windows that will be used in the `CreateDockableWindow` message and elsewhere;
- a label for identifying the plugin’s menu;
- a prefix for labelling properties required by the plugin API; and
- a prefix to be used in labelling items used in the plugin’s option pane

18.2.2.2. Example Plugin Core Class

We will derive the plugin core class for QuickNotepad from `EBPlugin` to allow the plugin core object to subscribe to the EditBus and receive a `CreateDockableWindow` message. There are no other messages to which the plugin core object needs to respond, so the implementation of `handleMessage()` will only deal with one class of message. We will define a few static `String` data members to enforce consistent syntax for the name of properties we will use throughout the plugin. Finally, we will use a standalone plugin window class to separate the functions of that class from the visible component class we will create.

The resulting plugin core class is lightweight and straightforward to implement:

```
public class QuickNotepadPlugin extends EBPlugin {
    public static final String NAME = "quicknotepad";
    public static final String MENU = "quicknotepad.menu";
    public static final String PROPERTY_PREFIX
        = "plugin.QuickNotepadPlugin.";
    public static final String OPTION_PREFIX
        = "options.quicknotepad.";

    public void start() {
        EditBus.addToNamedList(DockableWindow
            .DOCKABLE_WINDOW_LIST, NAME);
    }

    public void createMenuItems(Vector menuItems) {
        menuItems.addElement(GUIUtilities.loadMenu(MENU));
    }

    public void createOptionPanes(OptionsDialog od) {
        od.addOptionPane(new QuickNotepadOptionPane());
    }

    public void handleMessage(EBMessage message) {
        if(message instanceof CreateDockableWindow) {
            CreateDockableWindow cmsg = (CreateDockableWindow)
                message;
            if (cmsg.getDockableWindowName().equals(NAME)) {
                DockableWindow win = new QuickNotepadDockable(
                    cmsg.getView(), cmsg.getPosition());
                cmsg.setDockableWindow(win);
            }
        }
    }
}
```

The implementations of `createMenuItems()` and `createOptionPane()` are typically trivial, because the real work will be done using other plugin elements. Menu creation is performed by a utility function in jEdit's API, using properties defined in the plugin's properties file. The option pane is constructed in its own class.

If the plugin only had a single menu item (for example, a checkbox item that toggled activation of a dockable window), we would call `GUIUtilities.loadMenuItem()` instead of `loadMenu()`. We will explain the use of both methods in the next section.

The constructor for `QuickNotepadDockable` takes the values of the `View` object and the docking position contained in the `CreateDockableWindow` message. This will enable the plugin to "know" where it is located and modify its behavior accordingly. In another plugin, it could enable the plugin to obtain and manipulate various data that are available through a `View` object.

18.2.3. Resources for the Plugin Core Class

18.2.3.1. Actions

The plugin's user action catalog, `actions.xml`, is the resource used by the plugin API to get the names and definitions of user actions. The following `actions.xml` file from the QuickNotepad plugin can provide a model:

```
<!DOCTYPE ACTIONS SYSTEM "actions.dtd">

<ACTIONS>
    <ACTION NAME="quicknotepad.toggle">
        <CODE>
            view.getDockableWindowManager()
                .toggleDockableWindow(QuickNotepadPlugin.NAME);
        </CODE>
        <IS_SELECTED>
            return view.getDockableWindowManager()
                .isDockableWindowVisible(QuickNotepadPlugin.NAME);
        </IS_SELECTED>
    </ACTION>

    <ACTION NAME="quicknotepad-to-front">
    <CODE>
      view.getDockableWindowManager()
                .addDockableWindow(QuickNotepadPlugin.NAME);
        </CODE>
    </ACTION>

    <ACTION NAME="quicknotepad.choose-file">
        <CODE>
          wm = view.getDockableWindowManager();
          wm.addDockableWindow(QuickNotepadPlugin.NAME);
          wm.getDockableWindow(QuickNotepadPlugin.NAME)
            .chooseFile();
        </CODE>
    </ACTION>

    <ACTION NAME="quicknotepad.save-file">
        <CODE>
          wm = view.getDockableWindowManager();
          wm.addDockableWindow(QuickNotepadPlugin.NAME);
          wm.getDockableWindow(QuickNotepadPlugin.NAME)
            .saveFile();
        </CODE>
    </ACTION>

    <ACTION NAME="quicknotepad.copy-to-buffer">
        <CODE>
          wm = view.getDockableWindowManager();
          wm.addDockableWindow(QuickNotepadPlugin.NAME);
          wm.getDockableWindow(QuickNotepadPlugin.NAME)
            .copyToBuffer();
        </CODE>
    </ACTION>
</ACTIONS>
```

This file defines five actions. The first action uses the QuickNotepad’s internal plugin window name to toggle its visible state. The second action places QuickNotepad at the top of a stack of overlapping plugin windows. The other actions use `wm`, the `DockableWindowManager` object contained in the current view, to find the QuickNotepad plugin window and call the desired method.

If you are unfamiliar with BeanShell code, you may nevertheless notice that the code statements bear a strong resemblance to Java code, with two exceptions: the variable `wm` is not typed, and the variable `view` is never assigned any value.

For complete answers to these and other BeanShell mysteries, see Part III in *jEdit 3.2 User's Guide*; two observations will suffice here. First, the BeanShell scripting language is based upon Java syntax, but allows variables to be typed at run time, so explicit types for variables such as `wm` need not be declared. Second, the variable `view` is predefined by jEdit's implementation of BeanShell to refer to the current `View` object.

A formal description of each element of the `actions.xml` file can be found in Section 17.4.1.

18.2.3.2. Action Labels and Menu Items

Now that we have named and defined actions for the plugin, we have to put them to work. To do so, we must first give them labels that can be used in menu items and in the sections of jEdit's options dialog that deal with toolbar buttons, keyboard shortcuts and context menu items. We supply this information to jEdit through entries in the plugin's properties file. A call to `GUIUtilities.loadMenu()` or `GUIUtilities.loadMenuItem()` will read and extract the necessary labels from the contents of a properties file.

The following excerpt from `QuickNotepad.props` illustrates the format required for action labels and menu items:

```
# action labels
quicknotepad.toggle.label=QuickNotepad
quicknotepad-to-front.label=Bring QuickNotepad to front
quicknotepad.choose-file.label=Choose notepad file
quicknotepad.save-file.label=Save notepad file
quicknotepad.copy-to-buffer.label=Copy notepad to buffer

# application menu items
```

```
quicknotepad.menu.label=QuickNotepad
quicknotepad.menu=quicknotepad.toggle - quicknotepad.choose-file \
    quicknotepad.save-file quicknotepad.copy-to-buffer
```

`GUIUtilities.loadMenuItem()` and `GUIUtilites.loadMenu()` use syntatical conventions for the value of a menu property that simplifies menu layout. In `loadMenu()`, the use of the dash, as in the second item in the example menu list, signifies the placement of a separator. Finally, the character `'%'` used as a prefix on a label causes `loadMenu()` to call itself recursively with the prefixed label as the source of submenu data. Most plugins will not need to define menus that contain other submenus.

Note also that `quicknotepad-to-front` is not included in the menu listing. It will appear, however, on the Shortcuts pane of the Global Options dialog, so that the action can be associated with a keyboard shortcut.

18.3. Implementing a Dockable Window Class

The QuickNotepad plugin uses the dockable window API and provides one dockable window. Dockable window classes must implement the `DockableWindow` interface. There are basically two approaches to doing this. One is to have the top-level visible component also serve as the plugin window. The other is to derive a lightweight class that will create and hold the top-level window component. We will ilustrate both approaches.

18.3.1. Using a Single Window Class

A single window class must implement the `DockableWindow` interface as well as provide for the creation and layout of the plugin's visible components, and execution of user actions. The window for QuickNotepad will also implement the `EBComponent` so it can receive messages from the EditBus whenever the user has changed the plugin's settings in the Global Options dialog. Here is an excerpt from a class definition showing the implementation of both interfaces:

```
public class QuickNotepad extends JPanel
    implements EBComponent, DockableWindow
{
    private View view;
    private String position;
    ...
    public QuickNotepad(View view, String position) {
      this.view = view;
      this.position = position;
      ...
    }

    public String getName() {
        return QuickNotepadPlugin.NAME;
    }

    public Component getComponent() {
        return this;
    }
    ...
    public void handleMessage(EBMessage message) {
        if (message instanceof PropertiesChanged) {
            propertiesChanged();
        }
    }
    ...
}
```

This excerpt does not set forth the layout of the plugin's visible components, nor does it show how our user actions will be implemented. To provide more structure to the code, we will implement the `DockableWindow` interface in a separate, lightweight class.

18.3.2. An Action Interface

When an action is invoked, program control must pass to the component responsible for executing the action. The use of an internal table of BeanShell scripts that implement actions avoids the need for plugins to implement `ActionListener` or similar objects to respond to actions. Instead, the BeanShell scripts address the plugin through static methods, or if instance data is needed, the current `View`, its `DockableWindowManager`, and the plugin's `DockableWindow` object. When the plugin window class is separated from the class containing its visible components, there is one more link in the chain of control. This means that the plugin window should either execute the action itself or delgte it to the visible component.

We will use delegation between QuickNotepad's plugin window, which we will call `QuickNotepadDockable`, and the revised, slimmer `QuickNotepad` object. To represent that delegation we will employ a `QuickNotepadActions` interface as an organizational tool:

```
public interface QuickNotepadActions {
    void chooseFile();
    void saveFile();
    void copyToBuffer();
}
```

18.3.3. A Lightweight Dockable Window Class

Here is the complete definition of the `QuickNotepadDockable` class:

```
public class QuickNotepadDockable
    implements DockableWindow, QuickNotepadActions
{
    private QuickNotepad notepad;

    public QuickNotepadDockable(View view, String position) {
        notepad = new QuickNotepad(view, position);
    }

    public String getName() {
        return QuickNotepadPlugin.NAME;
    }

    public Component getComponent() {
        return notepad;
```

```
      }

      public void chooseFile() {
        notepad.chooseFile();
      }

      public void saveFile() {
        notepad.saveFile();
      }

      public void copyToBuffer() {
        notepad.copyToBuffer();
      }

  }
```

Once again we use a static data member of the plugin core class to provide a name for the plugin window to its `DockableWindowManager`. The last three methods implement the `QuickNotepadActions` interface.

18.4. The Plugin’s Visible Window

18.4.1. Class QuickNotepad

Here is where most of the features of the plugin will be implemented. To work with the dockable window API, the top level window will be a `JPanel`. The visible components reflect a simple layout. Inside the top-level panel we will place a scroll pane with a text area. Above the scroll pane we will place a panel containing a small tool bar and a label displaying the path of the current notepad file.

We have identified three user actions in the `QuickNotepadActions` interface that need implementation here: `chooseFile()`, `saveFile()`, and `copyToBuffer()`. As noted earlier, we also want the text area to change its appearance in immediate response to a change in user options settings. In order to do that, the window class must respond to a PropertiesChanged message from the EditBus.

Unlike the EBPlugin class, the EBComponent interface does not deal with the component's actual subscribing and unsubscribing to the EditBus. To accomplish this, we use a pair of methods inherited from the Java platform's JComponent that are called when the visible window becomes is assigned and unassigned to its DockableWindowContainer. These two methods, addNotify() and removeNotify(), are overridden to add and remove the visible window from the list of EditBus subscribers.

We will provide for two minor features when the notepad is displayed in the floating window. First, when a floating plugin window is created, we will give the notepad text area input focus. Second, when the notepad if floating and has input focus, we will have the **Escape** key dismiss the notepad window. An AncestorListener and a KeyListener will implement these details.

Here is the listing for the data members, the constructor, and the implementation of the EBComponent interface:

```
public class QuickNotepad extends JPanel
implements EBComponent, QuickNotepadActions
{
    private String filename;
    private String defaultFilename;
    private View view;
    private boolean floating;

    private QuickNotepadTextArea textArea;
    private QuickNotepadToolPanel toolPanel;

    //
    // Constructor
    //

    public QuickNotepad(View view, String position)
    {
        super(new BorderLayout());
    this.view = view;
    this.floating = position.equals(
        DockableWindowManager.FLOATING);

    this.filename = jEdit.getProperty(
        QuickNotepadPlugin.OPTION_PREFIX
        + "filepath");
    if(this.filename == null || this.filename.length() == 0)
    {
        this.filename = new String(jEdit.getSettingsDirectory()
            + File.separator + "qn.txt");
        jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
            + "filepath",this.filename);
    }
    this.defaultFilename = new String(this.filename);

    this.toolPanel = new QuickNotepadToolPanel(this);
    add(BorderLayout.NORTH, this.toolPanel);

    if(floating)
        this.setPreferredSize(new Dimension(500, 250));

    textArea = new QuickNotepadTextArea();
    textArea.setFont(QuickNotepadOptionPane.makeFont());
    textArea.addKeyListener(new KeyHandler());
    textArea.addAncestorListener(new AncestorHandler());

    JScrollPane pane = new JScrollPane(textArea);
    add(BorderLayout.CENTER, pane);

    readFile();
}

//
// Attribute methods
//

// for toolBar display
public String getFilename()
{
        return filename;
    }

    //
    // EBComponent implementation
    //

    public void handleMessage(EBMessage message)
    {
        if (message instanceof PropertiesChanged)
        {
            propertiesChanged();
        }
    }


    private void propertiesChanged()
    {
        String propertyFilename = jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX + "filepath");
        if(!defaultFilename.equals(propertyFilename))
        {
            saveFile();
            toolPanel.propertiesChanged();
            defaultFilename = propertyFilename.clone();
            filename = defaultFilename.clone();
            readFile();
        }
        Font newFont = QuickNotepadOptionPane.makeFont();
        if(!newFont.equals(textArea.getFont()))
        {
            textArea.setFont(newFont);
            textArea.invalidate();
        }
    }

    // These JComponent methods provide the appropriate points
    // to subscribe and unsubscribe this object to the EditBus

    public void addNotify()
    {
        super.addNotify();
        EditBus.addToBus(this);
    }


    public void removeNotify()
    {
        saveFile();
        super.removeNotify();
        EditBus.removeFromBus(this);
    }

    ...

}
```

This listing refers to a `QuickNotebookTextArea` object. It is currently implemented as a `JTextArea` with word wrap and tab sizes hard-coded. Placing the object in a separate class will simply future modifications.

18.4.2. Class QuickNotepadToolBar

There is nothing remarkable about the toolbar panel that is placed inside the `QuickNotepad` object. The constructor shows the continued use of items from the plugin's properties file.

```
public class QuickNotepadToolPanel extends JPanel
{
    private QuickNotepad pad;
    private JLabel label;

    public QuickNotepadToolPanel(QuickNotepad qnpad)
    {
        pad = qnpad;
        JToolBar toolBar = new JToolBar();
        toolBar.setFloatable(false);
```

```
            toolBar.add(makeCustomButton("quicknotepad.choose-file",
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.chooseFile();
                    }
                }));
            toolBar.add(makeCustomButton("quicknotepad.save-file",
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.saveFile();
                    }
                }));
            toolBar.add(makeCustomButton("quicknotepad.copy-to-buffer",
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.copyToBuffer();
                    }
                }));
            label = new JLabel(pad.getFilename(),
                SwingConstants.RIGHT);
            label.setForeground(Color.black);
            label.setVisible(jEdit.getProperty(
                QuickNotepadPlugin.OPTION_PREFIX
                + "show-filepath").equals("true"));
            this.setLayout(new BorderLayout(10, 0));
            this.add(BorderLayout.WEST, toolBar);
            this.add(BorderLayout.CENTER, label);
            this.setBorder(BorderFactory.createEmptyBorder(0, 0, 3, 10));
        }

        ...

    }
```

The method `makeCustomButton()` provides uniform attributes for the three toolbar buttons corresponding to three of the plugin's use actions. The menu titles for the user actions serve double duty as tooltip text for the buttons. There is also a `propertiesChanged()` method for the toolbar that sets the text and visibility of the label containing the notepad file path.

18.5. Designing an Option Pane

Using the default implementation provided by `AbstractOptionPane` reduces the preparation of an option pane to two principal tasks: writing a `_init()` method to layout and initialize the pane, and writing a `_save()` method to commit any settings changed by user input. If a button on the option pane should trigger another dialog, such as a `JFileChooser` or jEdit's own enhanced `VFSFileChooserDialog`, the option pane will also have to implement the `ActionListener` interface to display additional components.

The QuickNotepad plugin has only three options to set: the path name of the file that will store the notepad text, the visibility of the path name on the tool bar, and the notepad's display font. Using the shortcut methods of the plugin API, the implementation of `_init()` looks like this:

```
public class QuickNotepadOptionPane extends AbstractOptionPane
      implements ActionListener
{
    private JTextField pathName;
    private JButton pickPath;
    private FontSelector font;

    ...

    public void _init()
    {
        showPath = new JCheckBox(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "show-filepath.title"),
        jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX +  "show-filepath")
            .equals("true"));
        addComponent(showPath);

        pathName = new JTextField(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "filepath"));
        JButton pickPath = new JButton(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
```

```
                + "choose-file"));
            pickPath.addActionListener(this);

            JPanel pathPanel = new JPanel(new BorderLayout(0, 0));
            pathPanel.add(pathName, BorderLayout.CENTER);
            pathPanel.add(pickPath, BorderLayout.EAST);

            addComponent(jEdit.getProperty(
                QuickNotepadPlugin.OPTION_PREFIX + "file"),
                pathPanel);

            font = new FontSelector(makeFont());
            addComponent(jEdit.getProperty(
                QuickNotepadPlugin.OPTION_PREFIX + "choose-font"),
                font);
        }

        ...

    }
```

Here we adopt the vertical arrangement offered by use of the `addComponent()` method with one embellishment. We want the first “row” of the option pane to contain a text field with the current notepad file path and a button that will trigger a file chooser dialog when pressed. To place both of them on the same line (along with an identifying label for the file option), we create a `JPanel` to contain both components and pass the configured panel to `addComponent()`.

The `_init()` method uses properties from the plugin’s property file to provide the names of label for the components placed in the option pane. It also uses a property whose name begins with `PROPERTY_PREFIX` as a persistent data item - the path of the current notepad file. The elements of the notepad’s font are also extracted from properties using a static method of the option pane class.

The `_save()` method extracts data from the user input components and assigns them to the plugin’s properties. The implementation is straighforward:

```
public void _save()
```

```
{
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "filepath", pathName.getText());
    Font _font = font.getFont();
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "font", _font.getFamily());
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "fontsize", String.valueOf(_font.getSize()));
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "fontstyle", String.valueOf(_font.getStyle()));
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "show-filepath", String.valueOf(showPath.isSelected()));
}
```

The class has only two other methods, one to display a file chooser dialog in response to user action, and the other to construct a `Font` object from the plugin's font properties. They do not require discussion here.

18.6. Creating Other Plugin Resources

We have already covered in some detail one of the three types of resources that plugins include with their class files - the user action catalog - and the need for help documentation does not require extended discussion. The remaining resource is the properties file.

The first type of property data is information about the plugin itself. The first few entries from the QuickNotepad plugin's properties file fulfills this requirement:

```
# general plugin information
plugin.QuickNotepadPlugin.name=QuickNotepad
plugin.QuickNotepadPlugin.author=John Gellene
plugin.QuickNotepadPlugin.version=1.0
plugin.QuickNotepadPlugin.docs=QuickNotepad.html
plugin.QuickNotepadPlugin.depend.0=jdk 1.2
plugin.QuickNotepadPlugin.depend.1=jedit 03.01.99.00
```

These properties are described in detail in Section 17.4.2 and do not require further discussion here.

Next in the file comes a property that sets the title of the plugin in docked or frame windows. The use of the suffix `.title` in the property's key name is required by the plugin API.

```
# dockable window name
quicknotepad.title=QuickNotepad
```

The next sections, consisting of the action label and menu item properties, have been discussed earlier in Section 18.2.3.2.

```
# action labels
quicknotepad.toggle.label=QuickNotepad
quicknotepad-to-front.label=Bring QuickNotepad to front
quicknotepad.choose-file.label=Choose notepad file
quicknotepad.save-file.label=Save notepad file
quicknotepad.copy-to-buffer.label=Copy notepad to buffer

# application menu items
quicknotepad.menu=quicknotepad.toggle - quicknotepad.choose-file \
  quicknotepad.save-file quicknotepad.copy-to-buffer
```

We have created a small toolbar as a component of QuickNotepad, so file names for the button icons follow:

```
# plugin toolbar buttons
quicknotepad.choose-file.icon=Open.gif
quicknotepad.save-file.icon=Save.gif
quicknotepad.copy-to-buffer.icon=Edit.gif
```

The menu labels corresponding to these icons will also serve as tooltip text.

Finally, the properties file set forth the labels and settings used by the option pane:

```
# Option pane labels
options.quicknotepad.label=QuickNotepad
options.quicknotepad.file=File:
options.quicknotepad.choose-file=Choose
options.quicknotepad.choose-file.title=Choose a notepad file
options.quicknotepad.choose-font=Font:
```

```
options.quicknotepad.show-filepath.title=Display notepad file path

# Initial default font settings
options.quicknotepad.show-filepath=true
options.quicknotepad.font=Monospaced
options.quicknotepad.fontstyle=0
options.quicknotepad.fontsize=14

# Setting not defined but supplied for completeness
options.quicknotepad.filepath=
```

We do not define a default setting for the `filepath` property because of differences among operating systems. We will define a default file programatically that will reside in the directory jEdit designates for user settings.

18.7. Compiling the Plugin

We have already outlined the contents of the user action catalog, the properties file and the documentation file in our earlier dicusssion. The final step is to compile the source file and build the archive file that will hold the class files and the plugin's other resources.

Publicly released plugins include with their source a makefile for the jmk utility. The format for this file requires few changes from plugin to plugin. Here is the version of `makefile.jmk` used by QuickNotepad and many other plugins:

```
# A plugin makefile
#
# To recompile this plugin, start jmk
# in the plugin's source directory.
#

jar_name = "QuickNotepad";

##
# javac executable and args
##
#javac_bin = "javac";
```

```
#javac_opts = "-deprecation";

javac_bin = "jikes";
javac_opts = "-g" "-deprecation" "+E";

# set up the class path
new_class_path = "../../jedit.jar;.";
old_class_path = (getprop "java.class.path");

# concatenate the old and new class paths
if (equal "", old_class_path) then class_path = new_class_path;
else class_path = (cat old_class_path ";" new_class_path); end

cmd_javac = javac_bin "-classpath" class_path javac_opts;

##
# jar executable and args
##
jar_bin = "jar";
jar_opts = "cf0";
cmd_jar = jar_bin jar_opts;

srcs = (subst ".java", ".class",
  (glob (join (join (dirs "."), "/"), "*Plugin.java" "*.java"))
);
jar = (cat "../" jar_name ".jar");

get_files = function (dummy)
{
  extensions = "class" "gif" "html" "props";
  file_globs = (join "/*.", extensions);
  other_files = "actions.xml";

  (glob (join (dirs "."), file_globs)) other_files
}
end;

"all": jar;

"%.class" : "%.java";
```

```
{
    exec cmd_javac <;
}

jar: srcs;
{
    exec cmd_jar @ (get_files "1");
}

"clean":;
{
  delete (glob (join (dirs "."), "/*.class"));
}

".PHONY": "all";
```

For a full discussion of the `jmk` file format and command syntax, you should consult the jmk documentation site (http://jmk.sourceforge.net/edu/neu/ccs/jmk/jmk.html). Modifying this makefile for a different plugin will likely only require three changes:

- the name of the plugin;
- the choice of compiler (made by inserting and deleting the comment character `'#'`); and
- the classpath variables for `jedit.jar` any plugins this one depends on.

If you have reached this point in the text, you are probably serious about writing a plugin for jEdit. Good luck with your efforts, and thank you for contributing to the jEdit project.

V. jEdit API Reference

This part of the user's guide covers the jEdit *application programmer interface*. The information in this part is only useful to macro and plugin developers; you do not need to read it if you only want to use jEdit.

The first two chapter covers BeanShell commands, which are can only be used in macros. The second chapter covers APIs useful to both macros and plugins. The final chapter covers the EditBus message system, which is typically only used by plugins.

This part of the user's guide was written by John Gellene <`jgellene@nyc.rr.com`>.

Chapter 19. BeanShell Commands

BeanShell includes a set of *commands*; subroutines that can be called from any script or macro. The following is a summary of those commands which may be useful within jEdit.

> **Note:** Plugins, because they are written in Java and not BeanShell, cannot make use of BeanShell commands.

19.1. Output Commands

- `void` **`print`**`(`*`arg`*`);`

  Writes the string value of the argument to the activity log, or if run from the Console plugin, to the current output window. If *`arg`* is an array, `print` runs itself recursively on the array's elements.

- `void` **`cat`**`(String` *`filename`*`);`

  Writes the contents of *`filename`* to the activity log.

- `void` **`javap`**`(String | Object | Class` *`target`*`);`

  Writes the public fields and methods of the specified class to the output stream of the current process. Requires Java 2 version 1.3 or greater.

19.2. File Management Commands

- `void` **`dir`**`(String` *`dirname`*`);`

  Displays the contents of directory *`dirname`*. The format of the display is similar to the Unix `ls -l` command.

- `File` **`pathToFile`**`(String` *`filename`*`);`

  Create a `File` object corresponding to `filename`. Relative paths are resolved with reference to the BeanShell interpreter's working directory.

- `void` **`cd`**`(String` *`dirname`*`);`

  Changes the working directory of the BeanShell interpreter to *`dirname`*.

- `void` **`pwd`**

  Writes the current working directory of the BeanShell interpreter to the output stream of the current process.

- **`mv`**`(String` *`fromFile`*`, String` *`toFile`*`);`

  Moves the file named by *`fromFile`* to *`toFile`*.

- `void` **`rm`**`(String` *`pathname`*`);`

  Deletes the file name by *`pathname`*.

19.3. Component Commands

- `Object` **`load`**`(String` *`filename`*`);`

  Loads and returns a serialized Java object from *`filename`*.

- `void` **`save`**`(Component` *`component`*`, String` *`filename`*`);`

  Saves *`component`* in serialized form to *`filename`*.

- `JFrame` **`frame`**`(Component` *`frame`*`);`

  Displays the component in a top-level `JFrame`, centered and packed. Returns the `JFrame` object.

- `Font` **`setFont`**`(Component` *`comp`*`, int` *`ptsize`*`);`

Set the font size of *`component`* to *`ptsize`* and returns the new font.

19.4. Resource Management Commands

- `URL` **`getResource`**`(String` *`path`*`);`

  Returns the resource specified by *`path`*. A absolute path must be used to return any resource available in the current classpath.

19.5. Script Execution Commands

- **`exec`**`(String` *`cmdline`*`);`

  Start the external process by calling `Runtime.exec()` on *`cmdline`*. Any output is directed to the output stream of the calling process.

- `void` **`source`**`(String` *`filename`*`);`

  Evaluates the contents of *`filename`* as a BeanShell script in the interpreter's current namespace.

- `Object` **`eval`**`(String` *`expression`*`);`

  Evaluates the string *`expression`* as a BeanShell script in the interpreter's current namespace. Returns the result of the evaluation of `null`.

- `bsh.This` **`run`**`(String` *`filename`*`);`

  Run the BeanShell script named by *`filename`* in a copy of the existing namespace. The return value represent the object context of the script, allowing you to access its variables and methods.

- `Thread` **`bg`**`(String` *`filename`*`);`

  Run the BeanShell script named by *`filename`* in a copy of the existing namespace and in a separate thread. Returns the `Thread` object so created.

- `void` **`server`**`(int` *`port`*`);`

  Createes a "server" version of the BeanShell interpreter that shares the same namespace as the current interpreter. The server interpreter listens on the designated port.

  This requires the `bsh.util` package, which is not included with jEdit. It can be found in the stand-alone BeanShell distribution, available from http://www.beanshell.org.

  Caution

  Security of this port is not guaranteed. Use this command with extreme caution.

19.6. BeanShell Object Management Commands

- `bsh.This` **`object`**

  Creates a new BeanShell `This` scripted object which can hold data members. You can use this to create an object for storing miscellaneous crufties, like so:

  ```
  crufties = object();
  crufties.foo = "hello world";
  crufties.counter = 5;
  ...
  ```

- `bsh.This` **`extend`**`(bsh.This` *`object`*`);`

  Creates a new BeanShell `This` scripted object that is a child of the parameter *`object`*.

- `bsh.This` **`super`**`(String` *`scopename`*`);`

  Returns a refernece to the BeanShell `This` object representing the enclosing method scope specified by *`scopename`*. This method work similar to the `super` keyword but can refer to enclosing scope at higher levels in a hierarchy of scopes.

- **`bind`**`(bsh.This` *`ths`*`, bsh.Namespace` *`namespace`*`);`

  Binds the scripted object *`ths`* to *`namespace`*.

- `void` **`unset`**`(String` *`name`*`);`

  Removes the variable named by *`name`* from the current interpreter namespace. This has the effect of "undefining" the variable.

- **`setNameSpace`**`(bsh.Namespace` *`namespace`*`);`

  Set the namespace of the current scope to *`namespace`*.

19.7. Other Commands

- `void` **`exit`**

  Calls `System.exit(0)`.

  Caution

  While this command is available, you should always call `jEdit.exit()` instead so the application will shutdown in an orderly fashion.

- `void` **`debug`**

  Toggles BeanShell's internal debug reporting to the output stream of the current process.

- **`getSourceFileInfo`**

  Returns the name of the file or other source from which the BeanShell interpreter is reading.

Chapter 20. General jEdit Classes

20.1. Class jEdit

This is the main class of the application. All the methods in this class are static methods, so they are called with the following syntax, from both macros and plugins:

```
jEdit.method(parameters)
```

Here are a few key methods:

- `public static Buffer` **`openFile`**`(View` *`view`*`, String` *`path`*`);`

  Opens the file named `path` in the given `View`. To open a file in the current view, use the predefined variable `view` for the first parameter.

- `public static Buffer` **`newFile`**`(View` *`view`*`);`

  This creates a new buffer captioned Untitled-<n>in the given `View`.

- `public static boolean` **`closeBuffer`**`(View` *`view`*`, Buffer` *`buffer`*`);`

  Closes the buffer named `buffer` in the view named `view`. The user will be prompted to save the buffer before closing if there are unsaved changes.

- `public static void` **`saveAllBuffers`**`(View` *`view`*`, boolean` *`confirm`*`);`

  This saves all open buffers with unsaved changes in the given `View`. The parameter *`confirm`* determines whether jEdit initially asks for confirmation of the save operation.

- `public static boolean` **`closeAllBuffers`**`(View` *`view`*`);`

  Closes all buffers in the given `View`. A dialog window will be displayed for any buffers with unsaved changes to obtain user instructions.

- `public static void` **`exit`**`(View` *`view`*`, boolean` *`reallyExit`*`);`

This method causes jEdit to exit. If *`reallyExit`* is false and jEdit is running in background mode, the application will simply close all buffers and views and remain in background mode.

- `public static String` **`getProperty`**`(String` *`name`*`);`

  Returns the value of the property named by `name`, or `null` if the property is undefined.

- `public static boolean` **`getBooleanProperty`**`(String` *`name`*`);`

  Returns a boolean value of `true` or `false` for the property named by `name` by examining the contents of the property; returns `false` if the property cannot be found.

- `public static void` **`setProperty`**`(String` *`name`*`, String` *`property`*`);`

  This method sets the property named by `name` with the value `property`. An existing property is overwritten.

- `public static void` **`setBooleanProperty`**`(String` *`name`*`, boolean` *`value`*`);`

  This method sets the property named by `name` to `value`. The boolean value is stored internally as the string "true" or "false".

- `public static void` **`setTemporaryProperty`**`(String` *`name`*`, String` *`property`*`);`

  This sets a property that will not be stored during the current jEdit session only. This method is useful for storing a value obtained by one macro for use by another macro.

- `public static String` **`getJEditHome`**

  Returns the path of the directory containing the jEdit executable file.

- `public static String` **`getSettingsDirectory`**

  Returns the path of the directory in which user-specific settings are stored. This will be null if jEdit was started with the **-nosettings** command-line switch; so do not blindly use this method without checking for a null return value first.

The jEdit object also maintains a number of collections which are useful in some situations. They include the following:

- `public static EditAction[]` **`getActions`**

  Returns an array of "actions" or short routines maintained and used by the editor.

- `public static EditAction` **`getAction`**`(String` *`action`*`);`

  Returns the action named `action`, or `null` if it does not exist.

- `public static Buffer[]` **`getBuffers`**

  Returns an array of open buffers.

- `public static Properties` **`getProperties`**

  Returns a Java `Properties` object (a class derived from `Hashtable`) holding all properties currently used by the program. The constituent properties fall into three categories: application properties, "site" properties, and "user" properties. Site properties take precedence over application properties with the same "key" or name, and user properties take precedence over both application and site properties. User settings are written to a file named `properties` in the user settings directory upon program exit or whenever `jEdit.saveSettings()` is called.

- `public static int` **`getBufferCount`**

  Returns the number of open buffers.

- `public static Buffer` **`getBuffer`**`(String` *`path`*`);`

  Returns the `Buffer` object containing the file named `path`. or `null` if the buffer does not exist.

- `public static Mode[]` **`getModes`**

  Returns an array containing all editing modes used by jEdit.

- `public static Mode` **`getMode`**`(String` *`name`*`);`

Returns the editing mode named by `name`, or `null` if such a mode does not exist.

- `public static EditPlugin[]` **`getPlugins`**

  Returns an array containing all existing plugins.

- `plugin static EditPlugin` **`getPlugin`**`(String` *`name`*`);`

  Returns the plugin named by `name`, or `null` if such a plugin does not exist.

20.2. Class View

This class represents the “parent” or top-level frame window in which the editing occurs. It contains the various visible components of the program, including the editing pane, menubar, toolbar, and any docking windows containing plugins.

Some useful methods from this class include the following:

- `public void` **`splitHorizontally`**

  Splits the view horizontally.

- `public void` **`splitVertically`**

  Splits the view vertically.

- `public void` **`unsplit`**

  Unsplits the view.

- `public synchronized void` **`showWaitCursor`**

  Shows a “waiting” cursor (typically, an hourglass).

- `public synchronized void` **`hideWaitCursor`**

  Removes the “waiting” cursor. This method and `showWaitCursor()` are implemented using a reference count of requests for wait cursors, so that nested calls work correctly; however, you should be careful to use these methods in tandem.

- `public StatusBar` **`getStatus`**

  Eeach `View` displays a `StatusBar` at its bottom edge. It shows the current cursor position, the editing mode of the current buffer and other information. The method `setMessage(String message)` can be called on the return value of `getStatus()` to display reminders or updates. The message remains until the method is called again. To display a temporary message in the status bar, call `setMessageAndClear(String message)`, which will erase the message automatically after ten seconds.

- `public DockableWindowManager` **`getDockableWindowManager`**

  The object returned by this method keeps track of all dockable windows. See Section 20.5.

20.3. Class Registers

A `Register` is string of text indexed by a single character. Typically the text is taken from selected buffer text and the index character is a keyboard character selected by the user.

The application maintains a single `Registers` object consisting of an dynamically sized array of `Register` objects. The `Registers` class defines a number of methods that give each register the properties of a virtual clipboard.

The following methods provide a clipboard operations for register objects:

- `public static void` **`copy`**`(JEditTextArea` *`textArea`*`, char` *`register`*`);`

  Saves the selected text in the designated *`textArea`* to the register indexed at *`register`*. This will replace the existing contents of the designated register.

- `public static void` **`cut`**`(JEditTextArea` *`textArea`*`, char` *`register`*`);`

  Saves the selected text in the designated *`textArea`* to the register indexed at *`register`*, and removes the text from the text area. This will replace the existing contents of the designated register.

- `public static void` **`append`**`(JEditTextArea` *`textArea`*`, char` *`register`*`, String` *`separator`*`, boolean` *`cut`*`);`
- `public static void` **`append`**`(JEditTextArea` *`textArea`*`, char` *`register`*`, String` *`separator`*`);`
- `public static void` **`append`**`(JEditTextArea` *`textArea`*`, char` *`register`*`);`

  These three methods append the selected text in the *`textArea`* to the designated register. If the *`cut`* parameter is not specified, the selected text remains in the text area. If the *`separator`* parameter is not specified, a newline character is used to separate the appended text from any existng register text.

The following methods provide a lower-level interface for working with registers:

- `public static void` **`setRegister`**`(char` *`name`*`, Register` *`register`*`);`
- `public static void` **`setRegister`**`(char` *`name`*`, Register` *`newRegister`*`);`
- `public static void` **`clearRegister`**`(char` *`name`*`);`

  Sets the text of the designated register to `null`. If the register is one of the two registers reserved by the application (as discussed in the next section), the text value is set to an empty string.

- `public static Register` **`getRegister`**`(char` *`name`*`);`
- `public static Register[]` **`getRegisters`**

20.4. Interface Registers.Register

This interface requires implementation of two methods: `setValue()`, which takes a `String` parameter, and `toString()`, which return a textual representation of the register's contents. Two classes implement this interface. A `ClipboardRegister` is tied to the contents of the application's clipboard. The application assigns a `ClipboardRegister` to the register indexed under the character `$`. A `StringRegister` is created for registers assigned by the user. In addition, the application assigns to the `StringRegister` indexed under `%` the last text segment selected in the text area.

A `Register` object does not maintain a copy of its index key. Indexing is performed by the `Registers` object.

20.5. Class DockableWindowManager

Windows conforming to jEdit's dockable window API can appear in "panes" located above, below or to the left or right of the main editing pane. They can also be displayed in "floating" frame windows. A `DockableWindowManager` keeps track of the plugins associated with a particular `View`. Each `View` object contains an instance of this class.

- `public DockableWindow` **`getDockableWindow`**`(String` *`name`*`);`

  This method returns the `DockableWindow` object named by the `name` parameter. The name of a `DockableWindow` is a required property of the plugin. If there is no `DockableWindow` bearing the requested name, the method returns `null`.

- `public void` **`addDockableWindow`**`(String` *`name`*`);`

  If the `DockableWindow` named by the `name` parameter does not exist, a message is sent to the associated plugin to create it. The `DockableWindow` is then made visible.

- `public void` **`showDockableWindow`**`(String` *`name`*`);`
- `public void` **`removeDockableWindow`**`(String` *`name`*`);`
- `public void` **`toggleDockableWindow`**`(String` *`name`*`);`

  These methods, respectively show, hide and toggle the visibility of the `DockableWindow` object named by the `name` parameter. If the `DockableWindowManager` does not contain a reference to the window, these methods send an error message to the activity log and have no other effect. Only `addDockableWindow()` can cause the creation of a `DockableWindow`.

20.6. Class JEditTextArea

This class is the visible component that displays the file being edited. It is derived from Java's `JComponent` class.

Methods in this class that deal with selecting text rely upon classes derived from jEdit's `Selection` class. The "Selection API" permits selection and concurrent manipulation of multiple, non-contiguous regions of text. After describing the selection classes, we will outline the selection methods of `JEditTextArea`, followed by a listing of other methods in this class that are useful in writing macros.

20.6.1. Class Selection

This is an *abstract class* which holds data on a region of selected text. As an abstract class, it cannot be used directly, but instead serves as a parent class for specific types of selection structures. The definition of `Selection` contains two child classes used by the Selection API:

- `Selection.Range` - representing an ordinary range of selected text
- `Selection.Rect` - representing a rectangular selection region

A new instance of either type of `Selection` can be created by specifying its starting and ending caret positions:

```
selRange = new Selection.Range(start, end);

setRect = new Selection.Rect(start, end);
```

Both classes inherit or implement the following methods of the parent `Selection` class:

- `public int` **`getStart`**
- `public int` **`getEnd`**

  Retrieves the buffer position representing the start or end of the selection.

- `public int` **`getStartLine`**
- `public int` **`getEndLine`**

  Retrieves the zero-based index number representing the line on which the selection starts or ends.

- `public int` **`getStart`**`(Buffer` *`buffer`*`, int` *`line`*`);`
- `public int` **`getEnd`**`(Buffer` *`buffer`*`, int` *`line`*`);`

  These two methods return the position of the beginning or end of that portion of the selection falling on the line referenced by the *`line`* parameter. The parameter *`buffer`* is required because a `Selection` object is a lightweight structure that does not contain a reference to the `Buffer` object to which it relates.

  These methods do not check whether the *`line`* parameter is within the range of lines actually covered by the selection. They would typically be used within a loop defined by the `getStartLine()` and `getEndLine()` methods to manipulate selection text on a line-by-line basis. Using them without range checking could cause unintended behavior.

20.6.2. Selection methods in JEditTextArea

A `JEditTextArea` object maintains an `Vector` of current `Selection` objects. When a selection is added, the `JEditTextArea` attempts to merge the new selection with any existing selection whose range contains or overlaps with the new item. When selections are added or removed using by these methods, the editing display is updated to show the change in selection status.

Here are the principal methods of `JEditTextArea` dealing with `Selection` objects:

20.6.2.1. Adding and removing selections

- `public void` **`setMultipleSelectionEnabled`**`(boolean` *`multi`*`);`

Set multiple selection on or off according to the value of *`multi`*. This only affects the ability to make multiple selections in the user interface; macros and plugins can manipulate them regardless of the setting of this flag. In fact, in most cases, calling this method should not be necessary.

- `public Selection[]` **`getSelection`**

  Returns an array containing a copy of the current selections.

- `public int` **`getSelectionCount`**

  Returns the current number of selections. This can be used to test for the existence of selections.

- `public Selection` **`getSelectionAtOffset`**`(int` *`offset`*`);`

  Returns the `Selection` containing the specific offset, or `null` if there is no selection at that offset.

- `public void` **`addToSelection`**`(Selection` *`selection`*`);`
- `public void` **`addToSelection`**`(Selection[]` *`selection`*`);`

  Adds a single `Selection` or an array of `Selection` objects to the existing collection maintined by the `JEditTextArea`. Nested or overlapping selections will be merged where possible.

- `public void` **`extendSelection`**`(int` *`offset`*`, int` *`end`*`);`

  Extends the existing selection containing the position at *`offset`* to the position represented by *`end`*. If there is no selection containing *`offset`* the method creates a new `Selection.Range` extending from *`offset`* to *`end`* and adds it to the current collection.

- `public void` **`removeFromSelection`**`(Selection` *`sel`*`);`
- `public void` **`removeFromSelection`**`(int` *`offset`*`);`

  These methods remove a selection from the current collection. The second version removes any selection that contains the position at *`offset`*, and has no effect if no such selection exists.

20.6.2.2. Getting and setting selected text

- `public String` **`getSelectedText`**`(Selection` *`s`*`);`
- `public String` **`getSelectedText`**`(String` *`separator`*`);`
- `public String` **`getSelectedText`**

  These three methods return a `String` containing text corresponding to the current selections. The first version returns the text corresponding to a particular selection named as the parameter, allowing for iteration through the collection or focus on a specific selection (such as a selection containing the current caret position). The second version combines all selection text in a single `String`, separated by the `String` given as the *`separator`*. The final version operates like the second version, separating individual selections with newline characters.

- `public void` **`setSelectedText`**`(Selection` *`s`*`, String` *`selectedText`*`);`
- `public void` **`setSelectedText`**`(String` *`selectedText`*`);`

  The first version changes the text of the selection represented by *`s`* to *`selectedText`*. The second version sets the text of all active selections; if there are no selections, the text will be inserted at the current caret position.

  The second version of `setSelectedText()` is the method that will typically be used in macro scripts to insert text.

- `public int[]` **`getSelectedLines`**

  Returns a sorted array of line numbers on which a selection or selections are present. The current line is included in the array whether or not it is part of a selection.

  This method is the most convenient way to iterate through selected lines in a buffer. The line numbers in the array returned by this method can be passed as a parameter to such methods as `getLineText()`, as discussed below.

20.6.2.3. Other selection methods

The following methods perform selection operations without using `Selection` objects as parameters or return values. These methods should only be used in macros.

- `public void` **`selectBlock`**

  Selects the code block surrounding the caret.

- `public void` **`selectWord`**
- `public void` **`selectLine`**
- `public void` **`selectParagraph`**
- `public void` **`selectFold`**

  Selects the “fold” (a portion of text sharing a given indentation level) that contains the line where the editing caret is positioned.

- `public void` **`selectFoldAt`**`(int` *`line`*`);`

  Selects the fold containing the line referenced by *`line`*.

- `public void` **`selectAll`**
- `public void` **`selectNone`**
- `public void` **`indentSelectedLines`**

20.6.3. Other methods in JEditTextArea

20.6.3.1. Editing caret methods

These methods are used to get, set and move the position of the editing caret:

- `public int` **`getCaretPosition`**

Returns a zero-based index of the caret position in the existing buffer.

- `public void` **`setCaretPosition`**`(int` *`caret`*`);`

  Sets the caret position at *`caret`* and deactivates any selection of text.

- `public void` **`moveCaretPosition`**`(int` *`caret`*`);`

  This moves the caret to the position represented by *`caret`* without affecting any selection of text.

- `public int` **`getCaretLine`**

  Returns the line on which the caret is positioned.

Each of the following shortcut methods moves the caret. If the *`select`* parameter is set to `true`, the intervening text will be selected as well.

- `public void` **`goToStartOfLine`**`(boolean` *`select`*`);`
- `public void` **`goToEndOfLine`**`(boolean` *`select`*`);`
- `public void` **`goToStartOfWhiteSpace`**`(boolean` *`select`*`);`
- `public void` **`goToEndOfWhiteSpace`**`(boolean` *`select`*`);`
- `public void` **`goToFirstVisibleLine`**`(boolean` *`select`*`);`
- `public void` **`goToLastVisibleLine`**`(boolean` *`select`*`);`
- `public void` **`goToNextCharacter`**`(boolean` *`select`*`);`
- `public void` **`goToPrevCharacter`**`(boolean` *`select`*`);`
- `public void` **`goToNextWord`**`(boolean` *`select`*`);`
- `public void` **`goToPrevWord`**`(boolean` *`select`*`);`
- `public void` **`goToNextLine`**`(boolean` *`select`*`);`
- `public void` **`goToPrevLine`**`(boolean` *`select`*`);`
- `public void` **`goToNextParagraph`**`(boolean` *`select`*`);`

- `public void` **`goToPrevParagraph`**`(boolean` *`select`*`);`
- `public void` **`goToNextBracket`**`(boolean` *`select`*`);`
- `public void` **`goToPrevBracket`**`(boolean` *`select`*`);`

20.6.3.2. Methods for scrolling the text area

- `public void` **`scrollUpLine`**
- `public void` **`scrollUpPage`**
- `public void` **`scrollDownLine`**
- `public void` **`scrollUpPage`**
- `public void` **`scrollToCaret`**`(boolean` *`doElectricScroll`*`);`

  Scrolls the text area to ensure that the caret is visible. The *`doElectricScroll`* parameter detemines whether “electric scrolling” will occur. This leaves a minimum number of lines between the caret line and the top and bottom of the editing pane.

- `public void` **`centerCaret`**

  Scrolls the text area so that the line containing the edit caret is vertically centered.

- `public void` **`setFirstLine`**`(int` *`firstLine`*`);`
- `public int` **`getFirstLine`**

  This pair of methods deals with the line number of the first line displayed at the top of the text area. Lines that are hidden by folds or narrowing are ignored when making this “virtual” line count, so the line number will not necessarily conform to the line numbers displayed in the text area’s gutter. In addition, the virtual line index is zero-based, so `getFirstLine()` will always return zero for the first line of text.

  To convert a virtual line count to a physical count or vice versa, see Section 20.7.3.4.

- `public void` **`setElectricScroll`**`(int` *`electricScroll`*`);`

- `public int` **`getElectricScroll`**

  The “electric scroll” attribute is the number of lines above and below the editing caret that always remain visible when scrolling.

20.6.3.3. Methods for calculating editing positions

- `public int` **`getLineOfOffset`**`(int` *`offset`*`);`

  Returns the line on which the given offset is found.

- `public int` **`getLineStartOffset`**`(int` *`line`*`);`
- `public int` **`getLineEndOffset`**`(int` *`line`*`);`

  Returns the offset of the beginning or end of the given line.

20.6.3.4. Other methods for retrieving text

These methods can retrieve buffer text without regard to a selection or the position of the editing caret.

- `public String` **`getText`**`(int` *`start`*`, int` *`len`*`);`

  Returns the text located between buffer offset positions.

- `public String` **`getLineText`**`(int` *`lineIndex`*`);`

  Returns the text on the given line.

- `public String` **`getText`**

  Returns the entire text in the text area.

- `public void` **`setText`**`(String` *`text`*`);`

  Sets (and replaces) the entire text of the text area.

20.6.3.5. Methods for deleting text

- `public void` **`delete`**

  Deletes the character to the left of the editing caret.

- `public void` **`deleteWord`**
- `public void` **`deleteLine`**
- `public void` **`deleteParagraph`**
- `public void` **`deleteToStartOfLine`**
- `public void` **`deleteToEndOfLine`**

20.6.3.6. Methods for modifying text

- `public void` **`toLowerCase`**
- `public void` **`toUpperCase`**

  These two methods operate on all selected text, including multiple selections.

- `public void` **`joinLines`**

  Joins the current line with the following line.

- `public void` **`setOverwriteEnabled`**`(boolean` *`overwrite`*`);`
- `public boolean` **`isOverwriteEnabled`**

  Sets and gets whether added text will overwrite text at the editing caret or whether it will be inserted immediately to the right of the caret.

- `public void` **`userInput`**`(char` *`ch`*`);`

  Inserts the character at the caret position as if it were typed at the keyboard (keyboard input is actually passed to this method). Unlike `setSelectedText()`, or `insertString()` in the `Buffer` class, this method triggers any active text formatting features such as auto indent, abbreviation expansion and word wrap.

20.6.3.7. Methods for creating comments

- `public void` **`lineComment`**

  This inserts the line comment string at the beginning of each selected line.

- `public void` **`rangeComment`**

  This surrounds each selected text chunk with the comment start and end strings.

20.6.3.8. Methods for getting buffer statistics

- `public int` **`getBufferLength`**

  Returns the number of characters in the buffer.

- `public int` **`getLineCount`**

  Returns the number of lines in the buffer being edited.

- `public int` **`getVirtualLineCount`**

  Returns the number of “virtual” or visible lines in the buffer being edited, which may be less than the total number of lines because of folding or narrowing.

  To convert a virtual line count to a physical count or vice versa, see Section 20.7.3.4.

- `public int` **`getLineLength`**`(int` *`line`*`);`

  Returns the length of the line number `line` (using a zero-based count).

20.7. Class Buffer

A `Buffer` represents the contents of an open text file as it is maintained in the computer's memory (as opposed to how it may be stored on a disk). It is derived from Java's `PlainDocument` class.

20.7.1. File attribute methods

- `public String` **`getName`**
- `public String` **`getPath`**
- `public File` **`getFile`**

  This method may return `null` if the file is stored on a remote file system (for example, if the FTP or Archive plugins are in use). This method should be avoided if possible.

- `public boolean` **`isNewFile`**

  Returns whether a buffer lacks a corresponding version on disk.

- `public boolean` **`isDirty`**

  Returns whether there have been unsaved changes to the buffer.

- `public boolean` **`isReadOnly`**
- `public boolean` **`isUntitled`**

20.7.2. Editing attribute methods

- `public Mode` **`getMode`**
- `public void` **`setMode`**`(Mode` *`mode`*`);`

  Gets and sets the editing mode for the buffer.

- `public int` **`getIndentSize`**
- `public int` **`getTabSize`**

  These methods return the size of an initial indentation at the beginning of a line and the distance between tab stops, each measured in character columns. If these properties are not individually set for a specific buffer, they are inherited from the properties of the buffer's associated editing mode.

The following two methods are inherited by the `Buffer` class.

- `public void` **`putProperty`**`(Object` *`key`*`, Object` *`value`*`);`
- `public Object` **`getProperty`**`(Object` *`key`*`);`

  The `Buffer` object maintains a table of properties that describe a broad range of attributes. The value of each property is stored using a key for indexing purposes, usually a `String` that names the particular property. Property values can be set and retreived using these two methods. The `Object` returned by `getProperty()` usually has to be cast to a derived type to be useful. Most of these properties are documented in Section 6.2.

These two methods provide shortcuts for getting snd setting boolean properties.

- `public static boolean` **`getBooleanProperty`**`(String` *`name`*`);`

  Returns a boolean value of `true` or `false` for the property named by `name` by examining the contents of the property; returns `false` if the property cannot be found.

- `public static void` **`setBooleanProperty`**`(String` *`name`*`, boolean` *`value`*`);`

  This method sets the property named by `name` to `value`. The boolean value is stored internally as the string “true” or “false”.

20.7.3. Editing action methods

20.7.3.1. General editing methods

- `public void` **`reload`**`(View` *`view`*`);`

  Reloads the buffer from disk into *`view`*, asking for confirmation if the buffer has unsaved changes.

- `public boolean` **`save`**`(View` *`view`*`, String` *`path`*`);`
- `public boolean` **`save`**`(View` *`view`*`, String` *`path`*`, boolean` *`rename`*`);`

  The *`rename`* parameter causes a buffer's name to change if set to *`true`*; if *`false`*, a copy is saved to *`path`*.

- `public boolean` **`saveAs`**`(View` *`view`*`, boolean` *`rename`*`);`

  Prompts the user for a new name for saving the file.

- `public void` **`beginCompoundEdit`**
- `public void` **`endCompoundEdit`**

  Marks the beginning and end of a series of editing operations that will be dealt with by a single Undo command.

- `public void` **`removeTrailingWhiteSpace`**`(int[]` *`lines`*`);`

  Removes trailing whitespace in the lines referenced by the index numbers in *`lines`* array.

The following methods are inherited by `Buffer` from its parent class.

- `public String` **`getText`**`(int` *`offset`*`, int` *`length`*`);`
- `public void` **`getText`**`(int` *`offset`*`, int` *`length`*`, Segment` *`text`*`);`

These methods extract a portion of buffer text having length `length` beginning at offset position `offset`. The first method returns a newly created `String` containing the requested excerpt. The second version initializes an existing `Segment` object with the location of the requested excerpt. The `Segment` object represents array locations within the `Buffer` object's data and should be used on a read-only basis. Calling `toString()` on the `Segment` will create a new object suitable for manipulation.

- `public void` **`insertString`**`(int` *`offset`*`, String` *`text`*`, AttributeSet` *`attr`*`);`

  This method inserts the string `text` at offset `offset` in the buffer. The attribute `attr` is not used by jEdit and should be left as `null`.

- `public int` **`getLength`**

  This method returns the number of characters in the buffer.

20.7.3.2. Marker methods

Buffers may have one or more *markers* which serve as textual bookmarks. A `Marker` has three key attributes: the `Buffer` to which it relates, the line number to which the marker refers, and an optional shortcut character. The shortcut identifies the the key that can be pressed with the Markers>Go To Marker command to move the editing caret to the marker line location.

The position and shortcut character of a `Marker` object can be retrieved with the methods `getPosition()` and `getShortcut()`.

The `Buffer` class includes the following methods to set and retrieve markers:

- `public void` **`addMarker`**`(char` *`shortcut`*`, int` *`pos`*`);`

  Adds a marker for the line indicated by *`pos`* using *`shortcut`*. Set *`shortcut`* to `'\0'` to indicate the absence of a shortcut.

- `public Vector` **`getMarkers`**

Returns a `Vector` containing the buffer’s current markers.

- `public Marker` **`getMarkerAtLine`**`(int` *`line`*`);`

  Returns the first marker at the specified line, or `null` if no marker is present at the line.

- `public Marker` **`getMarker`**`(char` *`shortcut`*`);`

  Returns the marker with the specified shortcut, or `null` if no such marker exists.

- `public void` **`removeMarker`**`(int` *`line`*`);`

  Removes all markers at the specified line.

- `public void` **`removeAllMarkers`**

  Removes all markers in the buffer.

20.7.3.3. Folding methods

The “folding” features of jEdit allow sections of source code with a given indentation level to be hidden, creating “folds” that can be hidden and expanded, as well as a virtual line numbering scheme that skips hidden, folded lines. The following methods in the `Buffer` class deal with the folding mechanism.

- `public boolean` **`collapseFoldAt`**`(int` *`line`*`);`

  Collapses the fold that contains the specified line number. The method returns `false` if there are no folds in the buffer for the indicated line.

- `public boolean` **`expandFoldAt`**`(int` *`line`*`, boolean` *`fully`*`, JEditTextArea` *`textArea`*`);`

  Expands the fold that contains the specified line number. If *`fully`* is true, all folds at the line will be expanded, otherwise only one level of folding will be expanded. The *`textArea`* parameter is provided to the method to facilitiate scrolling after folds are expanded.

The method returns `false` if there are no folds in the buffer for the indicated line.

- `public void` **`expandFolds`**`(int` *`level`*`);`

  This method expands all folds in the buffer up to *`level`* and collapses all folds with a higher level. The *`level`* parameter represents the number of indentations, not the actual number of indented spaces.

- `public void` **`expandAllFolds`**

  Expands all folds in the buffer.

- `public void` **`narrow`**`(int` *`start`*`, int` *`end`*`);`

  Narrows the visible portion of the buffer to the specified line range. To undo the narrowing, call the `Buffer.expandAllFolds()` method.

20.7.3.4. Virtual and physical line indices

When jEdit's folding or narrowing features are used to hide portions of a buffer, the "virtual" line count visible in the text area is generally not equal to the "physical" line count of the buffer represented by the gutter's display. The following pair of methods translate one enumeration to the other.

- `public int` **`virtualToPhysical`**`(int` *`lineNo`*`);`
- `public int` **`physicalToVirtual`**`(int` *`lineNo`*`);`

20.8. Class Macros

The following shortcut methods are useful in displaying output messages or obtaining input from a macro.

- `public static void` **`message`**`(View` *`view`*`, String` *`message`*`);`

  Displays the text of *`message`* (with an information icon) in a modal message box centered on the designated *`view`*.

- `public static void` **`error`**`(View` *`view`*`, String` *`message`*`);`

  Similar to `message` but displays an error icon.

- `public static String` **`input`**`(View` *`view`*`, String` *`prompt`*`);`
- `public static String` **`input`**`(View` *`view`*`, String` *`prompt`*`, String` *`defaultValue`*`);`

  Displays the text of *`prompt`*, a text input field, and a question icon in the designated *`view`*. In the second version, the text field will initially contain the text of *`defaultValue`*. Returns the contents of the text field if the dialog box is dismissed by pressing the OK button, or `null` if the Cancel button is pressed.

20.9. Class SearchAndReplace

Search and replace routines are undertaken by jEdit's `SearchAndReplace` class.

The following static methods allow you to set or get the parameters for a search. You can do this prior to or even without activating the search dialog.

- `public static void` **`setSearchString`**`(String` *`search`*`);`
- `public static String` **`getSearchString`**
- `public static void` **`setReplaceString`**`(String` *`replace`*`);`
- `public static String` **`getReplaceString`**
- `public static void` **`setIgnoreCase`**`(boolean` *`ignoreCase`*`);`
- `public static boolean` **`getIgnoreCase`**
- `public static void` **`setRegexp`**`(boolean` *`regexp`*`);`

- `public static boolean` **`getRegexp`**

  Determines whether the search term is interpreted as a regular expression.

- `public static void` **`setReverseSearch`**`(boolean` *`reverse`*`);`
- `public static boolean` **`getReverseSearch`**

  Determines whether a reverse search will conducted from the current position to the beginning of a buffer. Currently, only literal reverse searches are supported.

- `public static void` **`setBeanShellReplace`**`(boolean` *`beanshell`*`);`
- `public static boolean` **`getBeanShellReplace`**

  Determines whether the replace string will be interpreted as a BeanShell expression.

- `public static void` **`setAutoWrapAround`**`(boolean` *`wrap`*`);`
- `public static boolean` **`getAutoWrapAround`**

  Determines whether a search will automatically "wrap" to the beginning of a buffer after the search reaches the buffer's end. If this flag is set to `false`, a dialog will request confirmation of a wrap-around search.

- `public static void` **`setSearchFileSet`**`(SearchFileSet` *`fileset`*`);`

  A `SearchFileSet` is an abstract class representing the set of files that are the subject of a search. There are four classes derived from `SearchFileSet`:

DirectoryListSet

This represents a set of files taken from a directory. It can be extended recursively to include files in subdirectories. The constructor for this class has the following syntax:

- `public` **`DirectoryListSet`**`(String` *`directory`*`, String` *`glob`*`, boolean` *`recurse`*`);`

  The parameter *`glob`* is the glob pattern that determines which files from the directory will be selected (see Appendix D for information about glob patterns), and `recurse` determines whether the selection will recurse into subdirectories.

class AllBufferSet

This class represents the set of all buffers currently open. The constructor for this class takes a file mask as a single parameter:

- `public` **`AllBufferSet`**`(String` *`glob`*`);`

class CurrentBufferSet

This class represents a buffer set consisting of the current buffer only. The constructor has no parameters.

- `public` **`CurrentBufferSet`**

class BufferListSet

This class represents a buffer set containing an arbitrary set of files specified by the user. The constructor takes a single `Vector` parameter containing the path names of the files to be searched.

- `public` **`BufferListSet`**`(Vector` *`files`*`);`

The actual tasks of searching and replacing, based on these parameters, are performed by the following methods. The return value of each indicates whether the operation succeeded.

- `public static boolean` **`find`**`(View` *`view`*`);`

  This will select the next instance of matching text if the search is successful.

- `public static boolean` **`replace`**`(View` *`view`*`);`

  This will replace the each occurrence of the “search string” in selected text with the “replace string”. If no text is selected, the method has no effect.

- `public static boolean` **`replace`**`(View` *`view`*`, Buffer` *`buffer`*`, int` *`start`*`, int` *`end`*`);`

  This will replace the each occurrence of the “search string” in the specified range with the “replace string”.

- `public static boolean` **`replaceAll`**`(View` *`view`*`);`

  This method performs a replacement in all buffers in the `SearchFileSet`. Text selection is ignored.

- `public static boolean` **`hyperSearch`**`(View` *`view`*`);`

  This collects all instances of matching text in the members of the `SearchFileSet` and displays them in a dedicated window. Text selection is ignored.

The “HyperSearch” and “Keep dialog” features, as reflected in checkbox options in the search dialog, are not handled from within `SearchAndReplace`. If you wish to have these options set before the search dialog appears, make a prior call to either or both of the following:

```
jEdit.setBooleanProperty(“search.hypersearch.toggle”,true);
jEdit.setBooleanProperty(“search.keepDialog.toggle”,true);
```

If you are not using the dialog to undertake a search or replace, you may call any of the search and replace methods (including `hyperSearch()`) without concern for the value of these properties.

To create and display the search and replace dialog, first assign desired values to the search settings using the methods described above. Then create a new `SearchDialog` object using the following constructor:

- `public` **`SearchDialog`**`(View` *`view`*`, String` *`searchString`*`, int` *`searchIn`*`);`

The parameter *`searchIn`* can take the defined constant values `CURRENT_BUFFER`, `ALL_BUFFERS` or `DIRECTORY`, defined in the `SearchDialog` class. This parameter determines which file set radio button to preselect in the dialog box.

20.10. Class GUIUtilities

The methods dealing with creating menus and menu items are described in Section 18.2.3.2. One other static method in this class encapsulates the creation and display of jEdit's custom file chooser dialog box.

- `public static String[] `**`showVFSFileDialog`**`(View `*`view`*`, String `*`path`*`, int `*`type`*`, boolean `*`multipleSelection`*`);`

  This method displays the `VFSFileChooserDialog` provided by jEdit. If *`path`* is set to `null`, the dialog will display the directory of the current buffer. The *`type`* parameter can either be `JFileChooser.OPEN_DIALOG` or `JFileChooser.SAVE_DIALOG` (you might need to import the `JFileChooser` class from the `javax.swing` package). The final parameter determines whether multiple selection of files is permitted.

20.11. Class TextUtilities

This class contains a number of static methods that can be helpful in handling buffer text.

- `public static int `**`findMatchingBracket`**`(Buffer `*`buffer`*`, int `*`line`*`, int `*`offset`*`);`

  Returns the offset of the bracket matching the one at offset *`offset`* of line *`line`* of the buffer; returns -1 if the bracket is unmatched or if the specified character is not a bracket. The method throws a `BadLocationException` if the *`line`* or *`offset`* parameters are out of range.

- `public static int `**`findWordStart`**`(String `*`line`*`, int `*`pos`*`, String `*`noWordSep`*`);`
- `public static int `**`findWordEnd`**`(String `*`line`*`, int `*`pos`*`, String `*`noWordSep`*`);`

  Returns the position on which the word found on line *`line`*, position *`line`* begins or ends. The parameter *`noWordSep`* contains those non-alphanumeric characters that will be treated as part of a word for purposes of finding the beginning or end of word (such as an underscore character).

- `public static String` **`format`**`(String` *`text`*`, int` *`maxLineLength`*`);`

  Reformats a string and inserts line separators as necessary so that no line exceeds *`maxLineLength`* in length.

- `public static String` **`spacesToTabs`**`(String` *`in`*`, int` *`tabSize`*`);`
- `public static String` **`tabsToSpaces`**`(String` *`in`*`, int` *`tabSize`*`);`

  Makes the indicated change based upon a tab size of *`tabSize`*.

20.12. Class MiscUtilities

This class is another collection of static utility methods.

These methods extract various elements from a path name:

- `public static String` **`getFileName`**`(String` *`path`*`);`
- `public static String` **`getFileExtension`**`(String` *`name`*`);`
- `public static String` **`getParentOfFile`**`(String` *`path`*`);`

  Returns the directory containing the specified local file.

The following method creates a string of whitespace characters that uses as many tabs as possible:

- `public static String` **`createWhiteSpace`**`(int` *`len`*`, int` *`tabSize`*`);`

  If *`tabSize`* is set to zero, the string will consist entirely of space characters. To get a whitespace string tuned to the current buffer's settings, call this method as follows:

  ```
  myWhitespace = MiscUtilities.createWhiteSpace(myLength,
      buffer.getTabSize());
  ```

Here are two sorting methods, one for simple arrays and one for Java `Vector` objects:

- `public static void` **`quicksort`**`(Object[]` *`obj`*`, Compare` *`compare`*`);`
- `public static void` **`quicksort`**`(Vector` *`vector`*`, Compare` *`compare`*`);`

The type of the second parameter in both methods is a Java *interface* defined inside the `MiscUtilities` class. Any Java class implementing an interface must implement each of the methods set forth in the interface's abstract specification. The `Compare` interface consists of a single method:

- `public int` **`compare`**`(Object` *`obj1`*`, Object` *`obj2`*`);`

To work correctly with the `quicksort` algorithm, this method should return a negative value if *`obj1`* is ordered prior to *`obj2`*, a positive value if *`obj2`* is prior, and zero if the two objects are equivalent for ordering purposes.

When writing macros, keep in mind that under Java versions earlier than 1.3, BeanShell cannot implement arbitrary interfaces such as `Compare` (although, as we have noted in Section 14.4.3, a BeanShell method can implement a number of specific listener interfaces). Fortunately, jEdit provides a number of classes implementing `Compare` for sorting purposes. Among them are `StringCompare` and `StringICaseCompare`. Both classes compare `String` object; the latter class compares two strings on a case-insentive basis.

Calling `quicksort` on a `Vector` of `String` objects could therefore take the following form:

```
MiscUtilities.quicksort(myVectorOfStrings,
    new StringICaseCompare());
```

There is no return value, but the `Vector` provided as the first parameter will be now be sorted on a case-insensitive basis.

20.13. Class BeanShell

This class integrates the BeanShell interpreter into jEdit. One method is worth mentioning here because it can be used in a macro to chain together execution of several macros:

- `public static void` **`runScript`**`(View` *`view`*`, String` *`path`*`, boolean` *`ownNamespace`*`, boolean` *`rethrowBshErrors`*`);`

This method runs the script file identified by *`path`*. Within that script, references to *`buffer`*, *`textArea`* and *`editPane`* are determined with reference to the *`view`* parameter. If *`rethrowBshErrors`* is set to true, any runtime exception thrown by the child script will be rethrown to the parent script for additional handling.

The parameter *`ownNamespace`* determines whether a separate namespace will be established for the BeanShell interpreter. If set to `false`, methods and variables defined in the script will be available to all future uses of BeanShell; if set to `true`, they will be lost as soon as the script finishes executing. jEdit uses a value of `false` when running startup scripts, and a value of `true` when running all other macros.

Chapter 21. EditBus Classes

This section describes the `EditBus` class itself, as well as the abstract `EBMessage` class and all classes that derive from it. See Section 16.2.3 for an overview of how the EditBus works.

21.1. Class EditBus

This class provides a messaging system for all components that implement the `EBComponent` interface, including `View` and `EBPlugin` objects.

- `public static void` **`addToBus`**`(EBComponent` *`component`*`);`
- `public static void` **`removeFromBus`**`(EBComponent` *`component`*`);`

  Adds or removes a subscribing component.

- `public static void` **`addToNamedList`**`(Object` *`tag`*`, Object` *`entry`*`);`
- `public static void` **`removeFromNamedList`**`(Object` *`tag`*`, Object` *`entry`*`);`

  Manages arbitiriary lists of objects. Used by jEdit to manage dockable windows. The ErrorList plugin also uses these methods to manage error sources.

- `public EBComponent[]` **`getComponents`**

  Returns an array of all components connected to the EditBus.

- `public void` **`send`**`(EBMessage` *`message`*`);`

  Send the specified message to all subscribers on the EditBus.

21.2. Interface EBComponent

This interface is required for any class that subscribes to messages published on the EditBus. It contains a single method.

- `public void` **`handleMessage`**`(EBMessage` *`message`*`);`

21.3. Class EBMessage

This abstract class defines a message that can be sent on the EditBus to subscribing components. It contains two attributes that can be obtained with the following methods:

- `public Component` **`getSource`**
- `public boolean` **`isVetoed`**
- `public void` **`veto`**

  This sets the *`vetoed`* state to `true`, which terminates circulation of the message to subscribers on the EditBus. To prevent a message from being vetoed, the message object must be derived from the abstract class `EBMessage.NonVetoable`. An object of this class will throw an `InternalError` if the `veto()` method is called on it.

A summary of classes derived from `EBMessage` can be found in the following sections.

21.4. Class BufferUpdate

This message is sent when the status of a buffer changes. It may not be vetoed by a subscriber, so that all subscribers are assured of receiving it regardless of an individual subscriber’s response.

- `public Buffer` **`getBuffer`**
- `public View` **`getView`**

  This may be `null` with some message types.

- `public Object` **`getWhat`**

  Returns one of the following constants defined in the `BufferUpdate` class:

- `CREATED`
- `LOAD_STARTED`
- `DIRTY_CHANGED` - a change in the buffer’s “dirty” status
- `MARKERS_CHANGED`
- `MODE_CHANGED`
- `ENCODING_CHANGED`
- `SAVING`

21.5. Class CreateDockableWindow

This message is sent by the `addDockableWindow()` method of the `DockableWindowManager` class; see Section 16.2.3.

- `public View` **`getView`**
- `public String` **`getDockableWindowName`**

  Returns the internal name of the requested dockable window. Your plugin should check if this is the name of one of the dockables it provides, and if so, call `setDockableWindow()` with the new dockable window instance.

- `public String` **`getPosition`**

  Returns one of the following constants defined in the `DockableWindowManager` class:

  - `FLOATING`
  - `TOP`
  - `BOTTOM`
  - `LEFT`
  - `RIGHT`

- `public void` **`setDockableWindow`**`(DockableWindow` *`window`*`);`

  Attaches a dockable window to the message. This prevents the message from being passed on to further subscribers.

21.6. Class EditorExiting

This message signifies that the host application is about to exit. The message has no parameters and may not be vetoed.

21.7. Class EditorExitRequested

This message signifies that a request has been made for the host application to exit. The request is subject to cancellation in response to a request to write a modified buffer to storage. It may not be vetoed.

- `public View` **`getView`**

21.8. Class EditorStarted

This message signifies that the host application has started. The message is sent before any views are created. The message has no parameters and it may not be vetoed.

21.9. Class EditPaneUpdate

This message is sent when the status of a edit pane changes. It may not be vetoed.

- `public EditPane` **`getEditPane`**
- `public Object` **`getWhat`**

Returns one of the following constants defined in the `EditPaneUpdate` class:

- `CREATED`
- `DESTROYED`
- `BUFFER_CHANGED` - a change in the buffer displayed in the edit pane

21.10. Class MacrosChanged

This message signifies that the list of available macros have changed. The message has no parameters and may not be vetoed.

21.11. Class PropertiesChanged

This message is sent when configuration settings have been changed through any of the option panes in the options dialog. The message has no parameters and may be vetoed.

21.12. Class SearchSettingsChanged

This message is sent when settings in the “Search and Replace” dialog have changed. The message has no parameters and may be vetoed.

21.13. Class VFSUpdate

This message is sent when the status of a file or directory changes. This allows subscribers that display or operate upon files an opportunity to adjust their state. This message may not be vetoed.

- `public String` **`getPath`**

21.14. Class ViewUpdate

This message is sent when the status of a view changes. It may not be vetoed.

- `public View` **`getView`**
- `public Object` **`getWhat`**

  Returns one of the following constants defined in the `ViewUpdate` class:

  - `CREATED`
  - `CLOSED`