

# **jEdit 3.1 User's Guide**

**Slava Pestov**

**John Gellene**

## **jEdit 3.1 User's Guide**

by Slava Pestov and John Gellene

Copyright © 1998-2001 by Slava Pestov

Copyright © 2001 by John Gellene

### **Legal Notice**

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no “Invariant Sections”, “Front-Cover Texts” or “Back-Cover Texts”, each as defined in the license. A copy of the license is included in the file `COPYING.DOC.txt` included with jEdit.

# Table of Contents

<b>I. Using jEdit .....</b>	<b>7</b>
1. Basic Concepts.....	8
1.1. Conventions .....	8
1.2. Starting jEdit .....	8
1.3. Buffers.....	9
1.4. Views.....	9
1.5. The Text Area.....	10
1.6. Command Repetition .....	11
2. Working With Files .....	12
2.1. Creating New Files .....	12
2.2. Opening Files .....	12
2.3. Saving Files.....	12
2.3.1. Autosave and Backups .....	13
2.3.2. Line Separators .....	13
2.4. The File System Browser.....	14
2.5. Reloading Files .....	15
2.6. Multi-Threaded I/O .....	16
2.7. Printing Files .....	16
2.8. Closing Files and Exiting jEdit .....	16
3. Editing Text.....	17
3.1. Moving The Caret .....	17
3.2. Selecting Text.....	17
3.3. Inserting and Deleting Text.....	18
3.4. Undo and Redo .....	19
3.5. Working With Words .....	19
3.6. Working With Lines.....	20
3.7. Working With Paragraphs .....	21
3.8. The Clipboard .....	21
3.9. Scrolling.....	22
3.10. Markers .....	22
3.11. Registers.....	23
3.11.1. Text Registers.....	23
3.11.2. Position Registers.....	24
3.12. Search and Replace .....	24

3.12.1. Searching For Text .....	24
3.12.2. Replacing Text .....	25
3.12.3. HyperSearch.....	25
3.12.4. Multiple File Search.....	25
3.12.5. The Search Bar.....	26
4. Edit Modes .....	28
4.1. Mode Selection .....	28
4.2. Syntax Highlighting.....	28
4.3. Writing Edit Modes.....	29
4.3.1. An XML Primer.....	29
4.3.2. The Preamble .....	30
4.3.3. The MODE Tag.....	30
4.3.4. The PROPS Tag .....	30
4.3.5. The RULES Tag.....	32
4.3.5.1. The TERMINATE Rule .....	32
4.3.5.2. The WHITESPACE Rule .....	33
4.3.5.3. The SPAN Rule.....	33
4.3.5.4. The EOL_SPAN Rule .....	34
4.3.5.5. The MARK_PREVIOUS Rule .....	35
4.3.5.6. The MARK_FOLLOWING Rule .....	35
4.3.5.7. The SEQ Rule .....	36
4.3.5.8. The KEYWORDS Rule .....	36
4.3.5.9. Token Types .....	37
4.4. Installing Edit Modes.....	37
5. Editing Source Code .....	39
5.1. Abbreviations.....	39
5.2. Bracket Matching.....	39
5.3. Tabbing and Indentation .....	40
5.3.1. Soft Tabs .....	40
5.3.2. Automatic Indent .....	41
5.4. Commenting Out Code .....	42
5.5. Folding .....	43
5.5.1. Narrowing .....	44
6. Customizing jEdit .....	46
6.1. The Buffer Options Dialog Box.....	46
6.2. Buffer-Local Properties.....	46
6.3. The Global Options Dialog Box .....	47

6.4. The jEdit Settings Directory .....	50
7. Installing and Using Plugins .....	52
7.1. The Plugin Manager.....	52
7.2. Installing Plugins .....	52
7.3. Updating Plugins.....	52
A. Keyboard Shortcuts .....	54
B. History Text Fields .....	60
C. Glob Patterns .....	61
D. Regular Expressions.....	62
E. The Activity Log .....	65
F. Command Line Usage .....	66
<b>II. Extending jEdit With Macros.....</b>	<b>68</b>
8. Macro Basics.....	69
8.1. What is BeanShell? .....	69
8.2. Recording Macros .....	70
8.3. How jEdit Organizes Macros .....	70
8.4. Single Execution Macros .....	71
9. A Few Simple Macros .....	73
9.1. The Mandatory First Example .....	73
9.2. Helpful Methods in the Macros Class.....	75
9.3. Now For Something Useful .....	77
10. A Dialog-Based Macro .....	81
10.1. Use of the Macro.....	81
10.2. Listing of the Macro.....	81
10.3. Analysis of the Macro.....	84
10.3.1. Import Statements .....	84
10.3.2. Create the Dialog .....	85
10.3.3. Create the Text Fields .....	86
10.3.4. Create the Buttons.....	87
10.3.5. Register the Action Listeners.....	87
10.3.6. Make the Dialog Visible .....	88
10.3.7. The Action Listener .....	89
10.3.8. Get the User's Input .....	89
10.3.9. Call jEdit Methods to Manipulate Text.....	90
10.3.10. The Main Routine .....	91
11. Macro Tips and Techniques .....	92
11.1. Getting Input for a Macro .....	92

11.1.1. Getting a Single Line of Text .....	92
11.1.2. Getting Multiple Data Items .....	93
11.1.3. Selecting Input From a List.....	96
11.1.4. Using a Single Keypress as Input .....	97
11.2. Using a Startup Macro .....	99
11.3. Debugging Macros.....	104
11.3.1. Identifying Exceptions .....	104
11.3.2. Using a Message Box As a Tracing Tool.....	105
11.3.3. Writing Trace Messages to the Activity Log .....	105
G. jEdit API Quick Reference .....	107
G.1. Class jEdit .....	107
G.2. Class View.....	110
G.3. Class DockableWindowManager .....	111
G.4. Class JEditTextArea .....	111
G.5. Class Buffer.....	115
G.6. Class Macros .....	117
G.7. Class SearchAndReplace .....	118
G.8. Class GUIUtilities .....	121
G.9. Class TextUtilities .....	121
G.10. Class MiscUtilities .....	122
G.11. Class BeanShell.....	123
H. Macros Included With jEdit .....	125
H.1. File Management Macros.....	125
H.2. Text Macros.....	125
H.3. Java Code Macros .....	127
H.4. Search Macros.....	128
H.4.1. The Find_Occurrence Macro Group .....	129
H.5. Console Plugin Macros .....	130
H.6. Macros for Other Plugins .....	132
H.7. Macros for Listing Properties .....	132
H.8. Miscellaneous Macros.....	133

# **I. Using jEdit**

# Chapter 1. Basic Concepts

## 1.1. Conventions

Whenever a specific menu item selection is referenced, 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. An additional jEdit-only convention is that menu items that end with “[x]” expect a character to be typed at the keyboard after being invoked.

Many jEdit commands can 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**.

## 1.2. Starting jEdit

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 double click on the jEdit icon. Unless initial files to open were specified on the command line or dropped onto jEdit’s icon, jEdit will load any files that were open in the previous editing session.

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 known as *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 up 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 Appendix F.

#### **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 accessible 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. 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.

## 1.4. Views

Each editor window is referred to 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 Section 1.2 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.

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.

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.

## 1.5. 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.

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 column and line number containing the caret, as well as the total number of lines in the buffer, is shown in the bottom-left corner of the text area.

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.

## 1.6. 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 14 Control-D**” will delete 14 lines, and “**Control-Enter 8 #**” will insert “#####” in the buffer.

# Chapter 2. Working With Files

## 2.1. 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.

## 2.2. 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 to open can be selected by holding down **Control**.

**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.

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.

## 2.3. Saving Files

Changes 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.

### 2.3.1. Autosave and Backups

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 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 preserved in the buffer's file name suffixed with a tilde ("~"). For example, `paper.tex` will be backed up to `paper.tex~`.

The autosave and backup features can be configured in the Loading and Saving pane of the **Utilities>Global Options** dialog box.

### 2.3.2. Line Separators

The three major operating systems use different conventions to mark line endings in text files. The MacOS uses Carriage-Return characters (`\r`, `^M`) for that purpose. Unix uses Newline characters (`\n`, `^J`). Windows uses both (`\r\n`, `^M^J`). jEdit can read and write files in all three formats. When a file is opened, the line separator is automatically detected. The line separator can be set on a buffer-specific basis in the **Utilities>Buffer**

Options dialog box. The default for new files can be set in the Loading and Saving pane of the Utilities>Global Options dialog box.

## 2.4. The File System Browser

Utilities>File System Browser displays a file system browser window. 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.

The directory path to view 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 C for information about glob patterns. Pressing the **Up** and **Down** keys in both text fields recalls previously entered strings; see Appendix B 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

- **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 in the current view, display it in a new view, or close it. Unopened file popups have commands for opening, 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.

## 2.5. 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 unsaved changes). This feature can be disabled in the **General** pane of the **Utilities>Global Options** dialog box.

**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.

## 2.6. 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 is displayed in the menu bar, in addition to progress meters for each running operation. 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. Setting the number to zero disables multi-threaded I/O completely; doing this is not recommended.

## 2.7. 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.

## 2.8. 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 3. Editing Text

## 3.1. 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 3.5, Section 3.6 and Section 3.7.

## 3.2. Selecting Text

A *selection* is a block of text marked for further manipulation. A selection can either span a range of text or cover a rectangular area.

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.

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 any other command.

**Edit>Rectangular Selection** (shortcut: **Control-\\**) toggles the current selection between range and rectangle mode.

**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 5.3.1.

## 3.3. Inserting and Deleting Text

Text entered at the keyboard is inserted into the buffer. If overstrike mode is on, one character is deleted from in front of the caret position for every character that is inserted. To activate overstrike mode, press **Insert**. The caret is drawn as horizontal line while in overstrike mode. This serves as a reminder of the differing behavior.

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.
- In the current buffer for the duration of the editing session in the **Utilities>Buffer Options** dialog box.
- 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 5.3 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 3.5, Section 3.6 and Section 3.7.

## 3.4. 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.

## 3.5. 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**.

## 3.6. 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 caret to the start of the current line.

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.

## 3.7. 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 3.3 for information and word wrap and changing the wrap column.

## 3.8. The Clipboard

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.

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

**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.

### **The X Window System**

The X Window System on Unix actually has two storage areas for text; the “primary selection”, and the “clipboard”. jEdit only uses the clipboard. However, the XClipboard plugin (see Chapter 7 for information about installing plugins) allows read-only access to the primary selection.

## **3.9. 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.

## 3.10. Markers

Once a *marker* has been set at a particular location in a buffer, it can be quickly returned to at any time. Any number of markers can be set in each buffer, and markers are persistent; they are saved to `.filename.marks`, where *filename* is the file name. (The dot prefix makes the marker file hidden on Unix systems).

**Search>Set Marker** (shortcut: **Control-E Control-M**) prompts for a marker name (the default being the selected text) and set a marker with that name at the caret position.

Selecting a marker from the **Search>Go to Marker** menu moves the caret to the marker's location. Selecting a marker from the **Search>Clear Marker** menu removes it from the buffer.

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

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

## 3.11. Registers

Each *register* can hold either a text string or caret position for later use. Registers have single-character names, hence their number is limited by how many keys can be typed on your keyboard. Register contents are global to the editor; all buffers and views share the same set. Registers are not persistent; their contents are lost when jEdit exits. The register `$` is an alias for the clipboard, and therefore registers can be considered as an extension of the clipboard concept.

**Edit>Registers>View Registers** displays a dialog box for viewing register contents. The popup menu in the dialog box lists all defined registers; selecting one will display its contents and type (text or position). It is not possible to change or add registers in this dialog box; it is for viewing only.

All register commands except for **View Registers** listen for the next key press and operate on the register with that name. For example, to copy the selection to register `x`, press **Control-R Control-C X** (The character “x” will not be inserted into the buffer).

### 3.11.1. Text Registers

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

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

Edit>Registers>Append to Register (shortcut: **Control-R Control-A**) adds the selected text to the existing contents of the specified register.

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

### 3.11.2. Position Registers

Edit>Registers>Save Position to Register (shortcut: **Control-T**) stores the current buffer name and caret position in the specified register.

Edit>Registers>Go to Position in Register (shortcut: **Control-Y**) opens the buffer named in the specified register (if necessary), and moves the caret to the saved position.

Edit>Registers>Select to Position in Register (shortcut: **Control-U**) creates a selection from the current caret position to the position saved in the specified register.

Edit>Registers>Swap Position with Register (shortcut: **Control-K**) goes to the position stored in the specified register, and saves the previous position in that register.

**Note:** Caret positions cannot be saved to the register `$` (clipboard).

## 3.12. Search and Replace

### 3.12.1. 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. Pressing the **Up** and **Down** keys in this text field recalls previously entered strings; see Appendix B for details.



The search can be made case insensitive (for example, searching for “Hello” will match “hello”, “HELLO” and “HeLIO”) 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 D for more information about regular expressions. Note that regular expressions can only be used when searching forwards.

If the **Reverse search** check box is not selected, clicking **Find** will locate the next occurrence of the search string after the caret position. Otherwise, it will locate the previous occurrence before 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, a dialog box will be displayed, offering to restart the search from the beginning (or end, if reverse search is enabled) of the buffer.

**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.

### 3.12.2. Replacing Text

The replacement string can be entered in the **Replace with** text field of the search and replace dialog box. Pressing the **Up** and **Down** keys in this text field recalls previously entered strings; see Appendix B for details.

Clicking **Replace** will replace the current selection with the replacement string.

Clicking **Replace & Find** will replace the current selection, and locate the next occurrence of the search string. Clicking **Replace All** will replace all occurrences of the search string with the replacement string.

### 3.12.3. 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.

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

### 3.12.4. 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 filter entered in the **Filter** text field will be searched. The filter is specified in the form of a glob pattern; see Appendix C for more information about glob patterns.

If the **Directory** radio button is selected, all files in the directory whose names match the filter 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.

Pressing the **Up** and **Down** keys in the **Filter** and **Directory** text fields recalls previously entered strings; see Appendix B for details.

### 3.12.5. 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 or replacement can be done from the search bar.

Initially, the search bar is in *incremental search* mode. In incremental search mode, the first occurrence of the search string is located in the current buffer as is it is being typed. Subsequent occurrences can be located by pressing **Enter** and **Shift-Enter** to search forwards and backwards, 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 **Up** and **Down** keys can be used to recall previously entered strings; see Appendix B for details.

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

**Control-.)** commands.

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

# Chapter 4. Edit Modes

An *edit mode* is an editor configuration intended to edit a specific type of file. Edit modes can specify syntax highlighting rules, auto indent behavior, and various useful customizations.

## 4.1. 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. To set the current buffer’s edit mode on a one-time basis, specify it in the **Utilities>Buffer Options** dialog box. To have a buffer open with a specific edit mode every time, 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:
```

## 4.2. Syntax Highlighting

Syntax highlighting is the display of programming language tokens using different fonts and colors. This makes the 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.

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.
- 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:
```

## 4.3. 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.

### 4.3.1. 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 is the `<PROPERTY>` tag 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.

## 4.3.2. The Preamble

Each mode definition must begin with the following:

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

## 4.3.3. The MODE Tag

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

## 4.3.4. 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="boxComment" VALUE="*" />
<PROPERTY NAME="blockComment" VALUE="//" />
<PROPERTY NAME="noWordSep" VALUE="_" />
<PROPERTY NAME="wordBreakChars" VALUE=" ,+-=<>/?^&*" />
</PROPS>
```

### 4.3.5. 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 4.3.5.3.

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 4.3.5.9 for a list of token types.

Each child element of the RULES tag defines a parser rule. More specific rules must be defined before general ones; for example, a rule matching the string “hello” rule must be placed before one matching “he”.

Here is an example RULES tag:

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



### 4.3.5.1. 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.

### 4.3.5.2. 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>
```

### 4.3.5.3. 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 4.3.5.9 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”.

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 CSS ruleset in another mode:

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

#### 4.3.5.4. 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 4.3.5.9 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>
```

#### 4.3.5.5. 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 4.3.5.9 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>
```

#### 4.3.5.6. 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 4.3.5.9 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>
```

#### 4.3.5.7. 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 4.3.5.9 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>
```

#### 4.3.5.8. The KEYWORDS Rule

There can only be one `KEYWORDS` tag per ruleset. The `KEYWORDS` rule defines keywords to highlight. Keywords are similar to `SEQs`, except that `SEQs` 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>
```

#### 4.3.5.9. 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

## 4.4. Installing Edit Modes

There are two locations where new edit modes can be installed; the `modes` subdirectory of the jEdit settings directory, and the `modes` subdirectory of the jEdit install directory. The location of the settings directory is operating system-specific; see Section 6.4.

Edit modes must be listed in a *mode catalog* file, otherwise they will not be available to jEdit. There is a catalog file in each mode directory, named `catalog`.

Catalogs are also written in XML. They consist of a `MODES` tag, containing a number of `MODE` tags. Each mode tag associates a mode name with an XML file and a filename and first line glob. 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>
```

The `MODE` tag supports the following attributes:

- `NAME` - the name of the edit mode, as it will appear in the **Buffer Options** dialog box, 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 C 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 C for information about glob patterns

# Chapter 5. Editing Source Code

## 5.1. 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 Abbrevs pane of the Utilities>Global Options dialog box.

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 Abbrevs pane of the Utilities>Global Options dialog box. If enabled, pressing the space bar will automatically expand the word before the caret, assuming it is a valid abbreviation.

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. 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.

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-I**) 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.

## 5.3. Tabbing and Indentation

jEdit makes a distinction between the *tab width*, which 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.
- 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.



### 5.3.1. 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.
- 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.

### 5.3.2. 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.
- 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**.

## 5.4. 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>Wing Comment** (shortcut: **Control-E Control-C**) encloses the selection with comment start and end strings. An example of Java wing commented code looks like so:

```
/* if(obj instanceof DBConnection)
    ((DBConnection)obj).close(); */
```

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 wing comment strings to “(“ and “)”:

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

**Edit>Source Code>Box Comment** (shortcut: **Control-E Control-B**) encloses the selection with comment start and end strings, and places the box comment string at the start of each line. An example of Java box commented code looks like so:

```
/* if(obj instanceof DBConnection)
*      ((DBConnection)obj).close(); */
```

The strings used for box commenting 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 box comment strings to “(\*)” and “\*\*”), with “\*\*”) placed at the start of each line:

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

**Edit>Source Code>Block Comment** (shortcut: **Control-E Control-K**) inserts the block comment string at the start of each selected line. An example of Java block commented code looks like so:

```
// if(obj instanceof DBConnection)
//      ((DBConnection)obj).close();
```

The block 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 block comment string to “#”:

```
:blockComment=#:
```

## 5.5. 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 buffer only, while displaying higher indent levels “zooms in” on the code 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. 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 current tab size, can be specified on a mode-specific or global basis in the **Utilities>Global Options** dialog box. 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**) 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 inaccessible. In such a case, simply invoke **Expand All Folds** to restore the visibility of the hidden lines.

### **5.5.1. 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 “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
blockComment	The block comment string. A block comment extends to the end of the line. For example, in Java mode the default value is “//”. See Section 5.4.
boxComment	The box comment string. A box comment is one delimited by the wing comment start and end strings, with the box comment string at the beginning of each line. For example, in Java mode the default value is “*”. See Section 5.4.

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 5.5.
commentEnd	The wing comment end string. For example, in Java mode the default value is “*/”. See Section 5.4.
commentStart	The wing comment start string. For example, in Java mode the default value is “/*”. See Section 5.4.
indentOnEnter	If set to “true”, indentation will be performed when the <b>Enter</b> key is pressed. See Section 5.3.
indentOnTab	If set to “true”, indentation will be performed when the <b>Tab</b> key is pressed. See Section 5.3.
indentSize	The width, in characters, of one indent. Must be an integer greater than 0. See Section 5.3.
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 3.3.
mode	The default edit mode for the buffer. See Chapter 4.
noTabs	If set to “true”, soft tabs (multiple space characters) will be used instead of “real” tabs. See Section 5.3.
noWordSep	A list of non-alphanumeric characters that are <i>not to be treated as word separators</i> .
syntax	If set to “false”, syntax highlighting will be not be performed. See Section 4.2.
tabSize	The tab width. Must be an integer greater than 0. See Section 5.3.
wordBreakChars	Characters, in addition to spaces and tabs, at which lines may be split when word wrapping. See Section 3.3.

## 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 Section 4.3.

## **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 Shortcut Editing Panes**

The Command Shortcuts, Plugin Shortcuts and Macro Shortcuts option panes let you change keyboard shortcuts. Each command can have up to two shortcuts associated with it.

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 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.

## 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>View 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.

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

- `jars` - this directory contains plugins. See Chapter 7.
- `macros` - this directory contains macros. See Part II in *jEdit 3.1 User's Guide*.
- `modes` - this directory contains custom edit modes. See Chapter 4.

- `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 7.
- `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 5.1.
- `activity.log` - a plain text file which contains the full activity log. See Appendix E.
- `history` - a plain text file which stores history lists, used by history text fields and the **Edit>Paste Previous** command. See Section 3.8 and Appendix B.
- `properties` - a plain text file which stores the majority of jEdit's settings.
- `recent` - a plain text file which stores the list of recently opened files and where the caret was positioned within them.
- `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 Section 1.2.

# Chapter 7. Installing and Using Plugins

Plugins are loadable modules of code that extend jEdit's functionality. This chapter only covers installing and updating plugins. Usage instructions for individual plugins can be found in the Help menu.

## 7.1. The Plugin Manager

Plugins>Plugin Manager displays the plugin manager window, which shows a list of installed plugins. Clicking on a plugin will display information about it. To remove some plugins, select them and click **Remove Selected Plugins**. This will issue a confirmation dialog box first.

## 7.2. Installing Plugins

Clicking the **Install New Plugins** button in the plugin manager will obtain a list of plugins which are not yet installed from the jEdit web site, and display them in the plugin installer dialog box.

Plugins to install can be selected by clicking the check box next to their names.

Radio buttons in the plugin installer dialog box select the location where plugins are to be installed. Plugins can be installed in either the `jars` subdirectory of the jEdit install directory, or the `jars` subdirectory of the jEdit settings directory. The location of the setting directory is operating system-specific; see Section 6.4.

Once plugins to install have been selected and the installation directory specified, clicking **Install** will begin downloading the new plugins.

**Note:** jEdit must be restarted before newly-installed plugins may be used.

## **7.3. Updating Plugins**

To check if updated versions of installed plugins are available, click the **Update Plugins** button in the plugin manager. The plugin manager will then connect to the jEdit web site and compare currently installed plugins against the latest available ones. Plugins to update can be selected by clicking the check box next to their names. Clicking **Update** will begin downloading the updated plugins.

**Note:** jEdit must be restarted before updated plugins may be used.

# Appendix A. Keyboard Shortcuts

## Files

For details, see Section 1.3, Section 1.4 and Chapter 2.

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

## Views

For details, see Section 1.4.

<b>Control-E Control-T</b>	Toggle gutter (line numbering).
<b>Control-2</b>	Split view horizontally.
<b>Control-3</b>	Split view vertically.
<b>Control-1</b>	Unsplit.
<b>Alt-Page Up</b>	Go to previous text area.
<b>Alt-Page Down</b>	Go to next text area.
<b>Control-E 1; 2; 3; 4</b>	Collapse/expand top; bottom; left; right docking area.

## Repeating

For details, see Section 1.6.

**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 3.1, Section 3.5, Section 3.6, Section 3.7 and Section 5.2.

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

## Selecting Text

For details, see Section 3.2, Section 3.5, Section 3.6, Section 3.7 and Section 5.2.

<b>Shift-Arrow</b>	Extend selection by one character or line.
<b>Control-Shift-Arrow</b>	Extend selection by one word or paragraph.
<b>Shift-Page Up; Shift-Page Down</b>	Extend selection by one screenful.

<b>Shift-Home</b>	Extend selection to first non-whitespace character of line, beginning of line, first visible line (repeated presses).
<b>Shift-End</b>	Extend selection to last non-whitespace character of line, end of line, last visible line (repeated presses).
<b>Control-Shift-Home</b>	Extend selection to beginning of buffer.
<b>Control-Shift-End</b>	Extend selection to end of buffer.
<b>Control-[</b>	Select code block.
<b>Control-E W; L; P</b>	Select word; line; paragraph.
<b>Control-E Control-L</b>	Select line range.
<b>Control-\</b>	Switch between range and rectangular selection mode.

## Scrolling

For details, see Section 1.4.

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

## Text Editing

For details, see Section 3.4, Section 3.3, Section 3.5, Section 3.6 and Section 3.7.

<b>Control-Z</b>	Undo.
<b>Control-E Control-Z</b>	Redo.
<b>Backspace; Delete</b>	Delete character before; after caret.
<b>Control-Backspace;</b>	Delete word before; after caret.
<b>Control-Delete</b>	
<b>Control-D; Control-E D</b>	Delete line; paragraph.



<b>Shift-Backspace; Shift-Delete</b>	Delete from caret to beginning; end of line.
<b>Control-E R</b>	Remove trailing whitespace from the current line (or all selected lines).
<b>Control-J</b>	Join lines.
<b>Control-B</b>	Complete word.
<b>Control-E F</b>	Format paragraph (or selection).

## Clipboard and Text Registers

For details, see Section 3.8 and Section 3.11.1.

<b>Control-X</b>	Cut selected text to clipboard.
<b>Control-C</b>	Copy selected text to clipboard.
<b>Control-V</b>	Paste clipboard contents.
<b>Control-R Control-X <i>key</i></b>	Cut selected text to register <i>key</i> .
<b>Control-R Control-C <i>key</i></b>	Copy selected text to register <i>key</i> .
<b>Control-R Control-A <i>key</i></b>	Append selected text to register <i>key</i> .
<b>Control-R Control-V <i>key</i></b>	Paste contents of register <i>key</i> .
<b>Control-E Control-V</b>	Paste previous.

## Markers and Position Registers

For details, see Section 3.10 and Section 3.11.2.

<b>Control-E Control-M</b>	Set marker.
<b>Alt-Up; Alt-Down</b>	Move caret to previous; next marker.
<b>Control-T <i>key</i></b>	Save position to register <i>key</i> .
<b>Control-Y <i>key</i></b>	Go to position saved in register <i>key</i> .
<b>Control-U <i>key</i></b>	Select to position saved in register <i>key</i> .
<b>Control-K <i>key</i></b>	Go to position saved in register <i>key</i> , and save previous position to that register.

## Search and Replace

For details, see Section 3.12.

<b>Control-F</b>	Open search and replace dialog box.
<b>Control-G</b>	Find next.
<b>Control-H</b>	Find previous.
<b>Control-E Control-F</b>	Find selection.
<b>Control-E Control-R</b>	Replace in selection.
<b>Control-E Control-G</b>	Replace in selection and find next.
<b>Control-E Control-A</b>	Replace all.
<b>Control-,</b>	Quick incremental search.
<b>Control-.</b>	Quick HyperSearch.

## Source Code Editing

For details, see Section 5.1, Section 5.3 and Section 5.4.

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

## Folding and Narrowing

For details, see Section 5.5 and Section 5.5.1.

<b>Alt-Backspace</b>	Collapse fold containing caret.
<b>Alt-Enter</b>	Expand fold containing caret one level only.
<b>Alt-Shift-Enter</b>	Expand fold containing caret fully.

<b>Control-E Enter <i>level</i></b>	Expand folds with level less than <i>level</i> , <i>collapse all others</i> .
<b>Control-E X</b>	Expand all folds.
<b>Control-E S</b>	Select fold.
<b>Control-E N</b>	Narrow to selection.

## Macros

For details, see Part II in *jEdit 3.1 User's Guide*.

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

## Appendix B. 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.

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

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

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

# Appendix C. 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*

Character class expressions (`[abc]`, etc) are supported in globs and behave exactly like regular expression character classes; see Appendix D for details.

# Appendix D. 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
- `\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 E. The Activity Log

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

**Utilities>View Activity Log** displays the last 500 lines of the activity log. 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.

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 F. Command Line Usage

On operating systems that support a command line, jEdit can be passed a multitude of options that 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:chapter5
```

Switch	Effect
-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 Section 1.2.
-newview	Only valid when connecting to another instance. Instead of opening files in an existing view, they will be opened in a new view.
-nogui	Only valid when running in background mode. Forces jEdit to start without an initial view.
-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. Has no effect when connecting to another instance via the edit server.
-noserver	Disables the edit server. Does not attempt to connect to the server, and does not start one either. For information about the edit server, see Section 1.2.
-nosettings	Starts jEdit without loading user-specific settings. See Section 6.4.
-nosplash	Starts jEdit without displaying the splash screen. Has no effect when connecting to another instance via the edit server.
-server= <i>name</i>	Stores the server port info in the file named <i>name</i> . <i>File names for this parameter are relative to the settings directory.</i>

Switch	Effect
-settings= <i>dir</i>	Loads and saves the user-specific settings from the directory named <i>dir</i> , instead of the default <i>user.home/.jedit</i> . <i>dir</i> will be created if it does not exist. 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.

## **II. Extending jEdit With Macros**

# Chapter 8. Macro Basics

This part of the user's guide is an introduction to using and writing macros in jEdit. First we will tell you a little about BeanShell, jEdit's macro scripting language, and how to invoke and organize your macros. Next, we will walk through a few simple macros. We then discuss some of the building blocks you will use in writing more complicated macros. Finally, we provide a reference guide of the most frequently used jEdit APIs.

Throughout this part of the user's guide, we will be referring to some of the plugins available for use with jEdit (in particular, the Console plugin). If you have not installed plugins, you really should consider doing so (see Chapter 7 for information), but you will not need plugins for most of what we discuss.

## 8.1. What is 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(tm).”

As you might gather from this short quote, BeanShell is very similar to Java and is designed to be easy for Java programmers to pick up. 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>).

## 8.2. 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. 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 undone when macros are displayed in the **Macros** menu. See Section 8.3 for details.

While recording is in progress, the string "(recording)" is displayed in the menu bar. jEdit records key strokes, menu item commands, tool bar clicks, and search and replace operations. 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. 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 easier 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>Play 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.

## 8.3. 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.

The listing of individual macros in the **Macros** menu item can be organized in a hierarchy using subdirectories in the general or user-specific macro folders. Each subdirectory containing BeanShell macros 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**. To update the menu listing by rescanning the general and user-specific macro directories, invoke **Macros>Rescan Macros**.

If a macro named `Startup.bsh` exists in either of the designated macro directories, it is executed near the end of the jEdit startup sequence. Unlike with other macros, any variables and methods defined in the startup macro are available to all other macros and BeanShell commands. If you take advantage of this feature, be sure to have only one macro named `Startup.bsh` (capitalization matters!), and place it in the either of the two macro directories. See Section 11.2 for a more detailed discussion on the use of a startup macro.

You can run any macro, regardless of where it is located, by invoking **Macros>Run Other Macro**. You will be presented with the usual file selection dialog box.

You can also assign a macro to a toolbar button, a keyboard shortcut or the context pop-up menu in the **Macro Shortcuts**, **Tool Bar** and **Context Menu** panes of the **Utilities>Global Options** dialog box.

### **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 7.

## **8.4. 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 9. A Few Simple Macros

Now we'll look at some simple macro scripts and show you how they work.

## 9.1. 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 famous 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 *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.
- 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 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.

There are no `import` statements in this script, and many BeanShell scripts written for jEdit will not require them. In Java, an `import` statement makes public classes from other files visible without a full specification of the class's classpath. Without an `import` statement or a fully qualified name, Java cannot identify most classes using a single name as an identifier. The `import` statement operates similarly to the `using` statement in C++.

In implementing BeanShell, jEdit causes a number of `import` statements to be read into the BeanShell interpreter everytime a script runs. These “hidden” import statements make the jEdit editor core classes available without specifying a full classpath. If you have downloaded the jEdit source code, you can find the automatically imported classes in the file `org/gjt/sp/jedit/jedit.bsh`.

Sometimes it is unclear if an `import` statement is required. You will rarely go wrong if you keep a few points in mind:

- You will not need an import statement for any macro that you record.
- When you are constructing a new object in a macro (for example, a dialog window or other GUI elements) you will need use import statements or fully qualified class names for those components - see Chapter 10 for details.
- If you try to use a class that is not imported and is not specified by its full classpath, the BeanShell interpreter will complain, usually with an error message relating to the offending line of code.

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. With the exception of a designated startup macro (see Section 8.3), variables defined in a BeanShell script (other than the script’s return value) are visible only within the script. You don’t need to know anything about Java packages to write BeanShell macros.

## 9.2. 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 `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);`

If this format is unfamiliar to you, here is a short explanation. The format is used in defining the methods in the source code of the `Macros` class. It is also found in a format called *javadoc* that extracts information from source code to provide online documentation to the class. For example, you can find the javadoc for jEdit's `Macros` class on the Giant Java Tree web site (<http://www.gjt.org/javadoc/org/gjt/sp/jedit/Macros.html>).

The first three words for each of these *declarations* tell you how the method is 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 displays an empty input field initially, the other lets you specify a 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`. BeanShell will complain if you call any methods on the `null` object. In most cases, you will want to exit gracefully from the script with a `return` statement, because the user intended to cancel macro execution.

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?",
    "Slava Pestov");
```

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 is good 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 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 */
```

## 9.3. 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 text currently selected with the contents of the `String` parameter. If no text is selected, the effect of this operation is to insert text at the caret 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.util.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:

```
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.
- 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 concatenated `String`.

**One more pre-defined variable**

In addition to `view`, `buffer` and `textArea`, there is one more pre-defined variable available for use in macros – `editPane`, which is set to the current `EditPane` instance. An edit pane contains a text area and buffer switcher, and among other things, contains methods for selecting the buffer to edit. Views can contain multiple edit panes if they are split. Edit panes will not be discussed any further because they are not particularly useful from a macro writer's perspective.



# Chapter 10. A Dialog-Based Macro

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

## 10.1. Use of the Macro

Next we will look at a macro that 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);
```

## 10.2. 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 statements (see Section 10.3.1)
import javax.swing.*;
import javax.swing.border.*;

// main routine
void prefixSuffixDialog()
{
    // create dialog object (see Section 10.3.2)
    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 10.3.3)
    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 10.3.4)
    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 10.3.5)
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 10.3.6)
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 10.3.7)
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 10.3.8)
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 10.3.9)
    end = textArea.getSelectionEndLine() + 1;
    for(line = textArea.getSelectionStartLine();
        line < end; ++line)
    {
        offsetBOL = textArea.getLineStartOffset(line);
        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 10.3.10)
prefixSuffixDialog();

// end of Add_Prefix_and_Suffix.bsh
```

## 10.3. Analysis of the Macro

### 10.3.1. Import Statements

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

These statements make all classes in the specified packages visible to the BeanShell interpreter without having to use the *fully qualified class name* each time one of the classes is referenced. As we mentioned previously (see Section 9.1), jEdit's implementation of BeanShell causes a number of classes to be automatically imported.

Those classes that are not automatically imported must be named by a full qualified name or be the subject of an `import` statement.

Most often, `import` statements are used when a macro needs classes that are part of the Java platform but which BeanShell does not implicitly import. These include GUI elements found in the Swing or AWT packages. In this macro, we are using a number of Swing classes: `JDialog`, `JPanel`, `JLabel`, `JButton` and `EmptyBorder`. We take a sweeping approach for the sake of convenience by importing all of the classes in the `javax.swing` and `javax.swing.border` packages.

### 10.3.2. 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 permits input of the prefix and suffix string, 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) to run the macro.

The Java object we need is a `JDialog` object. 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.

In basic terms, a `JDialog` object is designed as a window containing a single object called a *content pane*. The content pane in turns contains the various visible components of the dialog. While `JDialog` creates its own content pane when it is itself constructed, we will 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.

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.

### 10.3.3. 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 identifying 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 in the history file. 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 six pixel spacing 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.

### 10.3.4. 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.

### 10.3.5. 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 class specification that a derived class implements. 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 is treated 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.

### 10.3.6. 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` methods. 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.

### 10.3.7. 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 simple: 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.

### 10.3.8. 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()`.

### 10.3.9. Call jEdit Methods to Manipulate Text

```
// text manipulation begins here using calls
// to jEdit methods
end = textArea.getSelectionEndLine() + 1;
for(line = textArea.getSelectionStartLine();
    line < end; ++line)
{
    offsetBOL = textArea.getLineStartOffset(line);
    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.getSelectionStartLine()` and `textArea.getSelectionEndLine()`. For each of the lines in the loop's range, we 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(line)`;
- 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`.

- 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.

## 10.3.10. 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.

# Chapter 11. Macro Tips and Techniques

## 11.1. Getting Input for a Macro

The dialog-based macro discussed in Chapter 10 reflects an approach to obtaining input that is conventional from a Java perspective. 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.

### 11.1.1. Getting a Single Line of Text

As mentioned earlier in Section 9.2, 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()`.

### 11.1.2. 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

// *****import statements for excerpt*****

import javax.swing.Box;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

// *****portions of macro script omitted*****
```

## Chapter 11. Macro Tips and Techniques

```
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 beginning 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 takes the approach of listing every “imported” class in a separate `import` statement. If you don’t feel a need to keep track of which classes you are using in a macro, the global statement `import javax.swing.*;` will work just as well.
- 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 text to be displayed in the field as a parameter. When the message box is displayed, the initial 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 `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 it was necessary to set 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 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 final 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 the defined constants the option type (using **OK** and **Cancel** buttons) and the message type (a `QUERY_MESSAGE` that 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()`).

### 11.1.3. 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()

import javax.swing.JOptionPane;

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, a method returning 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
```

### 11.1.4. 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 the character value as a parameter to a BeanShell script.

The jEdit class `InputHandler` is an abstract class 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 code);
```

The contents of the `code` parameter will be run as a BeanShell script, with one important modification. The first time the string `__char__` appears in the parameter script, it will be substituted by the character value of the next key pressed after `readNextChar()` is called. The value of the key pressed 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( "ch = __char__;" ) ;
script.append( "match = text.indexOf(ch, 1);" ) ;
script.append( "if(match != -1) {" ) ;
```

```
script.append(    "if(ch != '\\n') ++match;"                );
script.append(    "textArea.select(start, start + match - 1);" );
script.append(    "}"   );

view.getInputHandler().readNextChar(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.
- It is also important that the child script define and set a local variable equal to the `__char__` if the value will be used more than once. This is because `__char__` is not a true variable but a placeholder token in the child script string. The implementation of `readNextChar` substitutes the character value of the keypress for `__char__` the first time it occurs in the child script string; then it executes the resulting string as a macro.
- Finally, access to the `InputHandler` object used by `jEdit` is available by calling `getInputHandler()` on the current view.

## 11.2. Using a Startup Macro

One useful feature in `jEdit` allows the user to run a macro script at startup near the end of the program's initialization routine. The name of the macro script must be `Startup.bsh` (capitalization matters!) and it must be located in one of the two macro directories.

The startup macro can perform additional initialization that cannot be handled by command line options or ordinary configuration options. It could, for example, open a

designated file (whether or not it was open at the close of the preceding jEdit session). It could write logging information to jEdit's activity log or an external file, or activate another application. It can also add variables, methods and scripted object to the namespace used by the BeanShell interpreter when running other macro scripts. This last feature 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**.

Here are a few short scripts that illustrate the startup macro process.

**Startup.bsh.** This macro defines a method and several variables, and looks for a second startup macro. If the second macro can be found, it is executed and the variable `orphanMessage` is modified.

```
// Startup.bsh

foo()
{
    fooMessage = "This is from Startup.bsh and "
                + "is part of a foo() object.";
    return this;
}

orphanMessage = "This is from Startup.bsh "
                + "but is not part of a scripted object.";

startup2 = Macros.getMacro("Startup2");
if(startup2 != null)
{
    BeanShell.runScript(view, startup2.path, false, false);
    orphanMessage += "\nStartup2 was found.";
}
else
{
    orphanMessage += "\nStartup2 was not found.";
}

// end of Startup.bsh
```

**Startup2.bsh.** This macro is called by the startup macro. It adds its own method and variable, and modifies one of the variables defined in `Startup.bsh` if it can find it.

```
// Startup2.bsh

bar()
{
    barMessage = "This is from Startup2.bsh and "
        + "is part of a bar() object.";
    return this;
}

orphanMessage2 = "This is from Startup2.bsh "
    + "but is not part of a scripted object.";

if(orphanMessage != null)
    orphanMessage = orphanMessage.toUpperCase();

// end of Startup2.bsh
```

**Namespace\_Test.bsh.** We will run this macro manually after startup. Among other things, it obtains the BeanShell namespace through successive calls to `jEdit`'s `BeanShell.getInterpreter()` and `BeanShell`'s own `Interpreter.getNamespace()` methods. The names and values of variables in the namespace are obtained by calls to `BeanShell`'s `Namespace.getVariableNames()` and `Namespace.getVariable()` methods. The rest of the script formats output and displays the result in a new buffer.

```
// Namespace_Test.bsh

jEdit.newFile(view);

namespace = BeanShell.getInterpreter().getNamespace();
v = namespace.getVariableNames();

sb = new StringBuffer();

sb.append("***** begin test of startup ");
sb.append("macro namespace capability *****\n\n");
sb.append("List of BeanShell namespace variables:\n\n");

for (i = 0; i < v.length; ++i)
{
    sb.append(v[i] + "\n");
    o = namespace.getVariable(v[i]);
    if(o != null)
```

```
{
    sb.append(o)
      .append("\n\n");
}
else
{
    sb.append("Value of variable ")
      .append(String.valueOf(i))
      .append(" is null.\n");
}
}
sb.append("***** end variable list *****\n\n");
textArea.setSelectedText(sb.toString());

f = foo();
b = bar();

sb.setLength(0);

sb.append("\n\n")
  .append("foo().fooMessage = \n")
  .append(f.fooMessage)
  .append("\n\n");

sb.append("bar().barMessage = \n")
  .append(b.barMessage)
  .append("\n\n");

sb.append("orphanMessage = \n")
  .append(this.orphanMessage)
  .append("\n\n");

sb.append("orphanMessage2 = \n")
  .append(this.orphanMessage2)
  .append("\n\n");

sb.append("***** end test *****\n");

textArea.setSelectedText(sb.toString());

// end of Namespace_Test.bsh
```

**The results.** Here are the result of running `Namespace_Test.bsh` after startup occurs:

```
***** begin test of startup macro namespace capability *****

List of BeanShell namespace variables:

startup2
Startup2

classLoader
org.gjt.sp.jedit.JARClassLoader@3bc473

orphanMessage
THIS IS FROM STARTUP.BSH BUT IS NOT PART OF A SCRIPTED OBJECT.
Startup2 was found.

bsh
'this' reference (XThis) to Bsh object: Bsh System Object

orphanMessage2
This is from Startup2.bsh but is not part of a scripted object.

***** end variable list *****


foo().fooMessage =
This is from Startup.bsh and is part of a foo() object.

bar().barMessage =
This is from Startup2.bsh and is part of a bar() object.

orphanMessage =
THIS IS FROM STARTUP.BSH BUT IS NOT PART OF A SCRIPTED OBJECT.
Startup2 was found.

orphanMessage2 =
This is from Startup2.bsh but is not part of a scripted object.

***** end test *****
```

You will see that objects and variables from both macros were available to the test macro when it ran after startup. In addition, the script in `Startup2.bsh` altered the variable `orphanMessage` defined in `Startup.bsh` by capitalizing the string contents. After that,

control returned to `Startup.bsh`, which added additional lower casetext to `orphanMessage`.

This example shows that the startup macro feature can provide additional opportunities for controlling jEdit's behavior. It also allows the creation of customized variables and methods that a user can apply in other macros or BeanShell code executed during the editing session.

## 11.3. Debugging Macros

Here are a few techniques that can prove helpful in debugging macros.

### 11.3.1. 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. Exceptions that are not handled within an application will usually result in an abrupt exit from the program. BeanShell saves a jEdit session from an untimely demise by catching any exceptions thrown by a macro and halting macro execution. When a BeanShell macro throws an exception in this fashion, it almost certainly means there is a bug in the macro.

Exceptions thrown while BeanShell runs a macro script, like other exceptions occurring during jEdit execution, result in entries being written to jEdit's activity log, see Appendix E. 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;
- *Execution errors*, which result from runtime exceptions thrown by the Java platform when macro code is executed.

An execution error will usually cause two exceptions to be written to the activity log: an exception tied to the `AWT-EventQueue` that orchestrates program execution, and an exception tied to BeanShell itself. An interpreter error results in only a single exception thrown by BeanShell. Usually the interpreter error is the subject of a message box.



The error messages emitted by BeanShell can often seem cryptic. The *stack trace* written by jEdit for an unhandled exception often refers to unrevealing methods like `Interpreter.eval()`. Nevertheless, examining the contents of the activity log may reveal clues as to the cause of a macro bug. If only an interpreter exception appears, it is a good idea to look for typing mistakes, uninitialized variables, or class names that are neither fully qualified nor subjects of an `import` statement. If there is an underlying Java exception thrown as well, the Java exception message may offer better clues as to the source of the error. The clues may include the type of exception thrown and the type of the object that the macro was manipulating when the exception occurred.

### 11.3.2. Using a Message Box As a Tracing Tool

Sometimes the activity log or other information will tell you what kind of error occurred but not where it arose in the macro script. If the error causes the display of an error box, there is a simple technique that will often pinpoint or narrow the range of offending code, without requiring repeated consultation of the activity log. Insert the following line of code at some point in the macro near the suspected error:

```
Macros.message(view, "tracing");
```

Of course, the message can be as descriptive as you like, and multiple messages can be used at different points in the macro code. It would be helpful to use a common tag as part of each function call, or in adjacent comments, so the messages can be easily found and removed from the finished macro script. If a “tracing” message pops up during macro execution, you know that the macro got that far in the script safely. If the BeanShell message appears first, the error is occurring before the “tracing” message box command is reached. Iteration of these steps will usually identify the location of a macro error.

### 11.3.3. Writing Trace Messages to the Activity Log

The most laborious, but often the most effective technique, is to insert code in the macro that writes additional messages to the activity log before the exception occurs, then to examine the activity log by choosing **Utilities>View Activity Log** after the macro completes execution. The messages would trace program execution and, when desired, report the value of significant variables. 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 the defined values `Log.DEBUG`, `Log.MESSAGE`, `Log.NOTICE`, `Log.WARNING` or `Log.ERROR`. The `log()` method will write a formatted string that contains the level of urgency and representations of the `source` and `message` parameters.

The following code would send a typical debugging message to the activity log:

```
Log.log(Log.DEBUG, BeanShell.class,  
        "counter = " + String.valueOf(counter)  
        + "; line = " + line);
```

The corresponding activity log entry might read as follows:

```
[debug] BeanShell: counter = 15; line = The corresponding  
        activity log entry might read as follows:
```

# Appendix G. jEdit API Quick Reference

The source code for jEdit covers over 200 classes (not counting classes from other open source packages like BeanShell and the GNU regular expression package). Embedding a BeanShell interpreter within jEdit makes most of those classes available to the user to help customize and extend the program. For the vast majority of circumstances, however, only a handful of classes will be needed for writing a macro script. The following is a quick guide to some of the principal jEdit classes that are useful in writing macros.

## G.1. Class jEdit

This is the main class of the application. The methods likely to be invoked in macros are all static methods, so they are called with the following syntax:

```
jEdit.name_of_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, Untitled buffer 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.

## Appendix G. jEdit API Quick Reference

- `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, simply close all buffers and views and remain in background mode.

- `public static final String getProperty(String name);`

Returns the value of the property named by `name`, or null if the property is undefined.

- `public static final boolean getBooleanProperty(String name);`

Returns a boolean value of true or false for the property named by `name` by examining the String contents of the property; returns false if the property cannot be found.

- `public static final 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 final 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 final 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 getSettingDirectory();`

Returns the path of the directory in which user-specific settings are stored.

The jEdit object also maintains a number of collections which a macro may need to use. 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 final 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 precedences other 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 plugin applications.
- `plugin static EditPlugin getPlugin(String name);`  
Returns the plugin named by name, or null if such a plugin does not exist.

## G.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 the macro writer should be careful to use these methods in tandem.
- `public DockableWindowManager getDockableWindowManager();`

The object returned by this method keeps track of all plugins that can be contained in dockable windows above, below and to the left and right of the editing pane.

Calling this method followed by a call to

`DockableWindowManager.getDockableWindow(String pluginName)` is an efficient way to access and (if necessary) activate a plugin application.

## G.3. Class `DockableWindowManager`

Windows conforming to `jEdit`'s docking API can appear in window “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 return `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 application 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`.

## G.4. Class JEditTextArea

This class is the visible component that displays the file being edited. It is derived from Java's `JComponent` class.

There are many methods in `JEditTextArea` that can be helpful in writing macros. Here is a summary grouped by function.

Methods to get, set and move the position of the editing caret:

- `public final int getCaretPosition();`  
Returns a zero-based index of the caret position in the existing buffer.
- `public final void setCaretPosition(int caret);`  
Sets the caret position at *caret* without selecting text.
- `public final void moveCaretPosition(int caret);`  
This moves the caret without moving the mark (the other end of a selection of text). This has the effect of extending or reducing the selected text.
- `public final int getCaretLine();`  
Returns the line on which the caret is positioned.
- `public final 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.

Methods to get and set selected text:

- `public final int getSelectionStart();`
- `public final int getSelectionEnd();`  
Returns the buffer position of the beginning or end of the current selection.



- `public final int getSelectionStartLine();`

- `public final int getSelectionEndLine();`

Returns the line containing the position of the selection's beginning or end.

- `public void select(int start, int end);`

- `public final void setSelectionStart(int selectionStart);`

- `public final void setSelectionEnd(int selectionEnd);`

Set the beginning or end of the selected text (or both) at the given buffer positions.

- `public void selectBlock();`

Selects the code block surrounding the caret.

- `public void selectWord();`

- `public void selectLine();`

- `public void selectParagraph();`

- `public void selectAll();`

- `public void selectNone();`

- `public final String getSelectedText();`

- `public void setSelectedText(String selectedText);`

Get and replaces the selected text.

- `public void indentSelectedLines();`

Methods to get buffer text without regard to a selection:

- `public final String getText(int start, int len);`

Returns the text located between buffer offset positions.

- `public final String getLineText(int lineIndex );`

Returns the text on the given line.

- `public String getText();`

Returns the entire text in the text area.

## Appendix G. *jEdit* API Quick Reference

- `public void setText(String text);`

Sets (and replaces) the entire text of the text area.

Methods for creating comments:

- `public void blockComment();`

This creates a block-style comment for each line in the selected text.

- `public void boxComment();`

This creates a box-style comment encompassing the line in the selected text.

- `public void wingComment();`

This creates a comment, using a single set of comment delimiters, beginning and ending with the selected text.

Shortcut methods that move the caret and select text (each taking a boolean parameter to determine whether or not intervening text will be selected):

- `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);`

Methods to delete 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();`

Methods to get statistics on the buffer being edited:

- `public final int getBufferLength();`  
Returns the length of the buffer being edited.
- `public final int getLineCount();`  
Returns the number of lines in the buffer being edited.
- `public final int getLineLength();`  
Returns the length of the line number `line` (using a zero-based count).

## G.5. 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 stored on a disk). It is derived from Java's `PlainDocument` class.

Here are some useful methods from the `Buffer` class:

- `public final String getName();`
- `public final String getPath();`
- `public final File getFile();`
- `public final Mode getMode();`
- `public void setMode(Mode mode);`

Gets and sets the editing mode for the buffer.

- `public int getIndentSize();`
- `public int getTabSize();`

These method returns 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.

- `public void beginCompoundEdit();`
- `public void endCompoundEdit();`

Marks the beginning and end of a series of operations that will be dealt with by a single Undo command.

- `public void addMarker(String name, int start, int end);`

Sets a marker named `name` for the section of text beginning at offset `start` and ending at offset `end`.

- `public void removeMarker(String name);`
- `public void removeAllMarkers();`

- `public final boolean isNewFile();`

Returns whether a buffer lacks a corresponding version on disk.

- `public final boolean isDirty();`

Returns whether there have been unsaved changes to the buffer.

- `public final boolean isReadOnly();`
- `public final boolean isUntitled();`
- `public boolean save(View view, String path);`
- `public boolean save(final View view, String path, final 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 removeTrailingWhiteSpace(int first, int last);`

Removes trailing whitespace from lines *first* to *last*.

The following methods are inherited by `Buffer` from its parent class. They are useful in extracting text from a `Buffer` object for searching purposes or other manipulation.

- `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.

## G.6. Class Macros

The following shortcut methods are useful in displaying output messages or obtaining

input.

- `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);`

Displays the text of *prompt*, a text input field, and a question icon in the designated *view*. 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.

- `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*. 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.

## G.7. 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 be conducted from the current position to the beginning of a buffer. Currently, only literal reverse searches are supported.

- `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 three classes derived from `SearchFileSet`:

#### *class 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 C 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();`

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 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.

- `public static void showSearchDialog(View view, String defaultFind);`

When activated, the dialog will reflect any options programatically set by `setIgnoreCase()`, `setRegexp()` and `setSearchFileSet()`, but not the search or replace strings. The parameter *defaultFind* (which may be `null`) contains the search text that will be displayed in the corresponding field of the dialog.

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.

## G.8. Class **GUIUtilities**

One static method in this class encapsulates the creation and display of file selection dialogs.

- `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.

## G.9. 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*.

## G.10. 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 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 editing mode, call this method as follows:

```
myWhitespace = MiscUtilities.createWhiteSpace(myLength,  
buffer.getMode().getProperty(tabSize).intValue());
```

Here, `getProperty()` returns an encapsulating `Integer` object. It yields its “primitive” `int` value with a call to `intValue()`.

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.

Except under JDK 1.3, `BeanShell` cannot implement arbitrary interfaces such as `Compare` (although, as we have noted earlier, a `BeanShell` method can implement a number of specific listener interfaces). Fortunately for macro writers, `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-insensitive 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.

## G.11. 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 and become visible to the user.

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 the startup macro, and a value of `true` when running all other macros.

# Appendix H. 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.

## H.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.
- `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.

## H.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.

- `Delete_Marker_at_Caret.bsh`

Deletes any existing marker placed in a buffer at the caret line.

Alerts user if no marker exists.

- `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.

- `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 protocol.

Rot13 is a simple encoding scheme involving fixed character substitution. A second application of the protocol 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.

## H.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.

## H.4. Search Macros

These macros provide various shortcuts to search methods. A group of macros in this category allow the user to search for 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 occurrence 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.

- `Search_Buffer.bsh`

Presets search settings for current buffer and displays search and replace dialog.

- `Search_Directory_Tree.bsh`

Presets search settings for “HyperSearch” in current directory and subdirectories, then displays search and replace dialog.

- `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.

### **H.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.

## H.5. Console Plugin Macros

The Console plugin allows the user to run commands in various available shells and view output in a dedicated text window. The following macros automate the use of Console and the output of commands that it processes.

More information about the Console plugin is found in its separate help file.

- `Display_Console_Output.bsh`

Copies contents of Console's output window to a new text buffer.

- `Go_to_Console.bsh`

Sets the input focus to the "Console" shell.

The plugin is opened if necessary. Binding this macro to a shortcut provides immediate access to the Console plugin from the keyboard.

- `Run_java.bsh`

Runs java on the current buffer in the Console plugin.

If the JCompiler plugin is installed, the full classname will be passed to java; otherwise the current directory name will be passed as the value of '-classpath'.

- `Run_javac.bsh`

Runs javac on the current buffer in the Console plugin.

The classpath for javac is set as the current buffer's directory.

- `Run_jikes.bsh`

Runs jikes on the current buffer in the Console plugin.

The jikes program is a popular, free alternative Java compiler. This macro executes jikes on the current buffer using its directory as the classpath.

- `Run_jmk.bsh`

Runs jmk in the Console plugin on a target supplied by the user in an input dialog.

The jmk program is a make utility written in Java.

- `Run_make.bsh`

Runs make in the Console plugin on a target supplied by the user in an input dialog.

- `Run_Last_Console_Command.bsh`

Reruns the last command run in the Console shell.

- `Run_Perl_Script.bsh`

Runs the current buffer in a Perl interpreter and displays output in the Console plugin.

The Perl interpreter must be supplied by the user. The macro will display an error message if the buffer's filename does not have a `.pl` extension. The macro also prompts for any command line parameters to be passed to the Perl interpreter.

## H.6. Macros for Other Plugins

These macros work with other plugins written for jEdit.

- `Go_to_Clipper.bsh`

Sets the input focus to the Clipper plugin.

- `Go_to_File_System_Browser.bsh`

Sets the input focus to the file system browser.

- `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()`.

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

- `Show_Dual_Diff.bsh`

Runs the JDiff plugin on two files supplied to a dialog.

## H.7. 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.

## H.8. 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 read-only view of the combined contents of “Command Shortcuts”, “Macro Shortcuts” and “Plugin Shortcuts” option panes. 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 BeanShell script. Opens 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 executing macros like `Go_to_Console.bsh` or `Go_to_Clipper.bsh`.

- `Notepad.bsh`

Displays a tabbed set of text windows in a floating frame.

- `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 achieved 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 “`dt`” 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 **dt** in a text buffer.
4. Press the space bar to expand **dt** 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.

