jEdit 4.1 User’s Guide

jEdit 4.1 User's Guide

Legal Notice

Table of Contents

I. Using jEdit

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

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

Chapter 1. Starting jEdit

1.1. Conventions

Several conventions are used throughout jEdit's user interface and this manual. They will be described here.

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

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

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

In many dialog boxes, the default button (it has a heavy outline, or a special border, depending on the current Swing look and feel) can be activated by pressing **Enter**. Similarly, pressing **Escape** will usually close a dialog box.

Finally, some user interface elements (menus, menu items, buttons) have a certain letter in their label underlined. Pressing this letter in combination with the **Alt** key activates the associated user interface widget.

MacOS

jEdit tries to adapt itself to established conventions when running on MacOS.

If you are using MacOS, mentally substitute the modifier keys you see in this manual as follows:

- Read **Control** as **Command**
- Read **Alt** as **Option**

If you only have a one-button mouse, a right button click (to show a context menu, and so on) can be simulated by holding down **Control** while clicking. A middle button click (to insert the most recent selection in the text area) can be simulated by holding down **Option** while clicking.

1.2. Platform-Independent Instructions

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

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

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

For more information about command line switches that control the server feature, see Section 1.4. Note that if you are using jEditLauncher to start jEdit on Windows, this switch cannot be specified on the MS-DOS prompt command line when starting jEdit; it must be set as described in Section G.2.

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

The edit server and security

Not only does the server pick a random TCP port number on startup, it also requires that clients provide an *authorization key*; a randomly-generated number only 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. Starting jEdit on Windows

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

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

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

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

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

1.4. Command Line Usage

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

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

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

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

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

Option	Description

Option	Description
`-background`	Runs jEdit in background mode. In background mode, the edit server will continue listening for client connections even after all views are closed. See Chapter 1.
`-nogui`	Makes jEdit not open an initial view, and instead only open one when the first client connects. Can only be used in combination with the **`-background`** switch. You can use this switch to “pre-load” jEdit when you log in to your computer, for example.
`-norestore`	Disables automatic restore of previously open files on startup. This feature can also be set permanently in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.
**`-run=`*`script`*	Runs the specified BeanShell script. There can only be one of these parameters on the command line. See Section 14.3 for details.
`-server`	Stores the server port info in the file named `server` inside the settings directory.
**`-server=`*`name`*	Stores the server port info in the file named *`name`*. File names for this parameter are relative to the settings directory.
`-noserver`	Does not attempt to connect to a running edit server, and does not start one either. For information about the edit server, see Chapter 1.
**`-settings=`*`dir`*	Loads and saves the user-specific settings in the directory named *`dir`*, instead of the default *`user.home`*`/.jedit`. The directory will be created automatically if it does not exist. Has no effect when connecting to another instance via the edit server.
`-nosettings`	Starts jEdit without loading user-specific settings. See Section 6.4.
`-noplugins`	Causes jEdit to not load any plugins. See Chapter 8. Has no effect when connecting to another instance via the edit server.
`-nostartupscripts`	Causes jEdit to not run any startup scripts. See Section 14.2. Has no effect when connecting to another instance via the edit server.
`-usage`	Shows a brief command line usage message without starting jEdit. This message is also shown if an invalid switch was specified.
`-version`	Shows the version number without starting jEdit.

Option	Description
- -	Specifies the end of the command line switches. Further parameters are treated as file names, even if they begin with a dash. Can be used to open files whose names start with a dash, and so on.

Chapter 2. jEdit Basics

2.1. Buffers

A *buffer* is the jEdit term for an open file. Several buffers can be opened and edited at once; 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.

View>Show Buffer Switcher (keyboard shortcut: **Alt-‘**) has the same effect as clicking on the buffer switcher combo box.

2.1.1. Memory Usage

The maximum number of open buffers depends on available *Java heap memory*. When in the Java heap, a buffer uses approximately two and a half times it's size on disk. This overhead is caused by the file being stored internally in Unicode (see Section 3.5), and various meta-data such as line numbers.

The status bar at the bottom of the view displays used and total Java heap memory; see Section 2.2.2 for details. This can give you a rough idea of how much memory the currently opened files are using. The Java heap grows if it runs out of room, but it only grows to a certain maximum size, and attempts to allocate Java objects that would grow the heap beyond this size fail with out-of-memory errors.

As a result, if the maximum heap size is set too low, opening large files or performing other memory-intensive operations can fail, even if you have a lot of free system memory.

To change the heap size on Windows, run "Set jEdit Parameters" from the "jEdit" group in the Programs menu. Then, in the resulting dialog box, under "Command line options for Java executable", change the option that looks like so:

```
-mx32m
```

(See Section G.2 for more information about the “Set jEdit Parameters” dialog box.)

On Unix, edit the `jedit` shell script and change the line that looks like so:

```
JAVA_HEAP_SIZE=32
```

In both cases, replace “32” with the desired heap size, in megabytes.

2.2. Views

A *view* is the jEdit term for an editor window. 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>New Plain View creates a new view but without any tool bars or docked windows. This can be used to open a small, unobtrusive window for taking notes and so on.

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

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

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

View>Unsplit Current (shortcut: **Control-0**) removes the split containing the current text area only.

View>Unsplit All (shortcut: **Control-1**) removes all splits 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>Go to Previous Text Area (shortcut: **Alt-Page Up**) shifts keyboard focus to the previous text area.

View>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; see Section 6.3.

2.2.1. Window Docking

Various jEdit and plugin windows can optionally be docked into the view. This can be configured in the Docking pane of the Utilities>Global Options dialog box; see Section 6.3.

When windows are docked into the view, strips of buttons are shown in the left, right, top, and bottom sides of the text area. Each strip contains buttons for the windows docked in that location, as well as a close box. Clicking a window's button shows that dockable window; clicking the close box hides the window again.

The commands in the View>Docking menu move keyboard focus between docking areas.

For power users

Each dockable has three commands associated with it; one is part of the menu bar and opens the dockable. The other two commands are:

- Window Name (Toggle) - opens the dockable window if it is hidden, and hide it if its already open.
- Window Name (New Floating Instance) - opens a new instance of the dockable in a floating window, regardless of the docking configuration.

  Another way to open a new floating instance of a window that is already docked is to right-click on the appropriate strip of buttons; this shows a menu from which you can choose to open a new floating instance.

  This can be used to view two different directories side-by-side in two file system browser windows, for example.

These commands cannot be invoked from the menu bar. However, they can be added to the tool bar or context menu, and given keyboard shortcuts; see Section 6.3.

2.2.2. The Status Bar

The *status bar* at the bottom of the view consists of the following components, from left to right:

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

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

Double-clicking on the caret location indicator displays the Edit>Go to Line dialog box; see Section 4.6.

- A message area where various prompts and status messages are shown.
- The current buffer's edit mode, fold mode, and character encoding. Double-clicking one of these displays the Utilities>Buffer Options dialog box. For more information about these settings, see:
  - Section 6.1
  - Section 5.1
  - Section 5.6
  - Section 3.5
- A set of flags which indicate various editor features and settings. Clicking each flag will toggle the feature in question; hovering the mouse over a flag will show a tool tip with an explanation:
- Word wrap - see Section 4.8.
- Multiple selection mode - see Section 4.2.2.
- Overwrite mode - see Section 4.3.
- Line separator - see Section 3.4.
- A Java heap memory usage indicator, that shows used and total heap memory, in megabytes. Double-clicking this indicator opens the Utilities>Troubleshooting>Memory Status dialog box.

The content of the status bar can be customized in the Status Bar pane of the Utilities>Global Options dialog box.

For power users

To quickly toggle the line separator or word wrap settings without having to use the mouse, assign keyboard shortcuts to the Toggle Line Separator and Toggle Word Wrap commands in the Shortcuts pane of the Utilities>Global Options dialog box.

2.3. The Text Area and Gutter

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 text editing is performed closer than three lines from the top or bottom of the text area. This feature is called *electric scrolling*.

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.

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.

Many text area and gutter settings can be customized to suit your taste in the Text Area and Gutter panes of the Utilities>Global Options dialog box; see Section 6.3.

Chapter 3. Working With Files

3.1. Creating New Files

File>New (shortcut: **Control-N**) opens a new, empty, buffer. Another way to create a new file is to specify a non-existent file name when starting jEdit on the command line. A new file will be created on disk when the buffer is saved for the first time.

3.2. Opening Files

File>Open (shortcut: **Control-O**) displays a file system browser dialog box and loads the specified file into a new buffer.

Multiple files can be opened at once by holding down **Control** while clicking on them in the file system browser. The file system browser supports auto-completion; typing the first few characters of a listed file name will select the file.

More advanced features of the file system browser are described in Section 3.6.

File>Insert displays a file system browser dialog box and inserts the contents of the specified file at the caret position.

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

The Utilities>Current Directory menu lists all files and directories in the current buffer's directory. Selecting a file opens it in a buffer for editing; selecting a directory opens it in the file system browser (see Section 3.6).

Note: Files that you do not have write access to are opened in read-only mode, where editing is not permitted.

Tip: jEdit supports transparent editing of GZipped files; if a file begins with the GZip "magic number", it is automatically decompressed before loading and compressed when saving. To compress an existing file, you need to change a setting in the Utilities>Buffer Options dialog box; see Section 6.1 for details.

3.3. Saving Files

Changed made in 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 As renames the buffer and saves it in a new location. Note that using this command to save over another open buffer will close the other buffer, to stop two buffers from being able to share the same path name.

File>Save a Copy As saves the buffer to different location but does not rename it., but doesn't rename the buffer, and doesn't clear the "modified" flag. Note that using this command to save over another open buffer will automatically reload the other buffer.

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

> **Two-stage save**
>
> To prevent data loss in the unlikely case that jEdit should crash in the middle of saving a file, files are first saved to a temporary file named `#`*`filename`*`#save#`. If this operation is successful, the original file is replaced with the temporary file.
>
> However, in some situations, this behavior is undesirable. For example, on Unix saving files this way will result in the owner and group of the file being reset. If this bothers you, you can disable this so-called "two-stage save" in the Loading and Saving pane of the Utilities>Global Options dialog box.

3.3.1. Autosave and Crash Recovery

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

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

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

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

3.3.2. Backups

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

The behavior of the backup feature is specified in theLoading and Saving pane of the Utilities>Global Options dialog box.

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

- The Max number of backups setting determines the number of backups to save. Setting this to zero disables the backup feature. Settings this to more than one adds numbered suffixes to file names. By default only one backup is saved.
- If the Backup directory setting is non-empty, backups are saved in that location. Otherwise, they are saved in the same directory as the original file. The latter is the default behavior.
- The Backup filename prefix setting is the prefix that is added to the backed-up file name. This is empty by default.
- The Backup filename suffix setting is the suffix that is added to the backed-up file name. This is "~" by default.
- Backups can optionally be saved in a specified backup directory, instead of the directory of the original file. This can reduce clutter.
- The Backup on every save option is off by default, which results in a backup only being created the first time a buffer is saved in an editing session. If switched on, backups are created every time a buffer is saved.

3.4. Line Separators

Unix systems use newlines (`\n`) to mark line endings in text files. The MacOS uses carriage-returns (`\r`). Windows uses a carriage-return followed by a newline (`\r\n`). jEdit can read and write files in all three formats.

The line separator used by the in-memory representation of file contents is always the newline character. When a file is being loaded, the line separator used in the file on disk is stored in a per-buffer property, and all line-endings are converted to newline characters for the in-memory representation. When the buffer is consequently saved, the value of the property replaces newline characters when the buffer is saved to disk. The line separator used by a buffer can be changed in the Utilities>Buffer Options dialog box. See Section 6.1.

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

3.5. Character Encodings

An encoding specifies a way of storing characters on disk. jEdit can use any encoding supported by the Java platform. The current buffer's encoding is shown in the status bar.

The default encoding, used to load and save files for which no other encoding is specified, can be set in the Loading and Saving pane of the Utilities>Global Options dialog box.

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

To open a file stored using an encoding other than the default, select the encoding from the Commands>Encoding menu of the file system browser before opening the file.

The encoding to use when saving a specific buffer can be set in the Utilities>Buffer Options dialog box.

If a file is opened without an explicit encoding specified and it appears in the recent file list, jEdit will use the encoding last used when working with that file; otherwise the default encoding will be used.

Unfortunately, there is no way to obtain a list of all supported encodings using the Java APIs, so jEdit only lists a few of the most common encodings; however, any other supported encoding name can be typed in.

3.5.1. Commonly Used Encodings

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

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

On Windows, various other encodings, which are known as *code pages* and are identified by number, are used to store non-English text. The corresponding Java encoding name is `Cp` followed by the code page number.

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

16-bit Unicode files are automatically detected as such when opened, regardless of the encoding specified by the user. The closely-related `UTF8` encoding, which uses variable-length characters, is also supported, however UTF8 files are *not* auto-detected.

3.6. The File System Browser

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

The file system browser can be customized in the Utilities>Global Options dialog box.

3.6.1. Navigating the File System

The directory to browse is specified in the Path text field. Clicking the mouse in the text field automatically selects its contents allowing a new path to be quickly typed in. If a relative path is entered, it will be resolved relative to the current path. This text field remembers previously entered strings; see Appendix C. The same list of previously browsed directories is also listed in the Utilities>Recent Directories menu; selecting one opens it in the file system browser.

To browse a listed directory, double-click it (or if you have a three-button mouse, you can click the middle mouse button as well). Alternatively, click the disclosure widget next to a directory to list its contents in place.

To browse higher up in the directory hierarchy, double-click one of the parent directories in the parent directory list.

Files and directories in the file list are shown in different colors depending on what glob patterns their names match. The patterns and colors can be customized in the File System Browser>Colors pane of the Utilities>Global Options dialog box.

To see a specific set of files only (for example, those whose names end with `.java`), enter a glob pattern in the Filter text field. This text fields remembers previously entered strings.

See Appendix D for information about glob patterns.

Unopened files can be opened by double-clicking (or by clicking the middle mouse button). Open files have their names underlined, and can be selected by single-clicking. Holding down **Shift** while opening a file will open it in a new view.

Clicking a file or directory with the right mouse button displays a popup menu containing various commands.

> **Tip:** The file list sorting algorithm used in jEdit handles numbers in file names in an intelligent manner. For example, a file named `section10.xml` will be placed after a file named `section5.xml`. A conventional letter-by-letter sort would have placed these two files in the wrong order.

3.6.2. The Tool Bar

The file system browser has a tool bar containing a number of buttons. Each item in the Commands menu (described below) except Show Hidden Files and Encoding has a corresponding tool bar button.

3.6.3. The Commands Menu

Clicking the Commands button displays a menu containing the following items:

- Parent Directory - moves up in the directory hierarchy.
- Reload Directory - reloads the file list from disk.
- Root Directory - on Unix, goes to the root directory (`/`). On Windows and MacOS X, lists all mounted drives and network shares.
- Home Directory - displays your home directory.
- Directory of Current Buffer - displays the directory containing the currently active buffer.
- New File - opens new, empty, buffer in the current directory. The file will not actually be created on disk until the buffer is saved.
- New Directory - creates a new directory after prompting for the desired name.
- Search in Directory - displays the search and replace dialog box set to search all files in the current directory. If a file is selected when this command is invoked, its extension becomes the file name filter for the search; otherwise, the file name filter entered in the browser is used. See Section 4.12 for details.
- Show Hidden Files - toggles if hidden files are to be shown in the file list.

- Encoding - a menu for selecting the character encoding to use when opening files. See Section 3.5.

3.6.4. The Plugins Menu

Clicking the Plugins button displays a menu containing plugin commands. For information about plugins, see Chapter 8.

3.6.5. The Favorites Menu

Clicking the Favorites button displays a menu showing all directories in the favorites list. To add the selected directory to the favorites (or the current directory, if there is no selection), invoke Add to Favorites from this menu. To remove a directory from the favorites, invoke Edit Favorites, which will show the favorites list in the file system view; then select Delete from the appropriate directory's right-click menu.

3.6.6. Keyboard Shortcuts

The file system browser can be navigated from the keyboard:

- **Enter** - opens the currently selected file or directory.
- **Shift**-**Enter** - opens the currently selected file in a new view, or the currently selected directory in a new file system browser window.
- **Left** - goes to the current directory's parent.
- **Up** - selects previous file in list.
- **Down** - selects next file in list.
- / - displays the root directory.
- ~ - displays your home directory.
- - - displays the directory containing the current buffer.
- Typing the first few characters of a file's name will select that file.

The file system tree must have keyboard focus for these shortcuts to work. They are not active in the Path or Filter text fields.

3.7. Reloading From Disk

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 to reload the buffer from disk and lose any unsaved changes made in jEdit. This warning dialog box can be disabled in the General pane of the Utilities>Global Options dialog box; see Section 6.3.

File>Reload can be used to 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 discards unsaved changes in all open buffers and reload them from disk, asking for confirmation first.

3.8. Multi-Threaded I/O

To improve responsiveness and perceived performance, jEdit executes all buffer input/output operations asynchronously. While I/O is in progress, the status bar displays the number of remaining I/O operations. The Utilities>Troubleshooting>I/O Progress Monitor command displays a window with more detailed status information and progress meters. This window is floating by default, but it can be set to dock into the view in the Docking pane of the Utilities>Global Options dialog box; see Section 2.2.1. I/O requests can also be aborted in this window, however note that aborting a buffer save can result in data loss.

3.9. Printing

File>Print (shortcut: **Control-P**) prints the current buffer.

File>Page Setup displays a dialog box for changing your operating system's print settings, such as margins, page size, print quality, and so on.

The print output can be customized in the Printing pane of the Utilities>Global Options dialog box. The following settings can be changed:

- The font to use when printing.
- If a header with the file name should be printed on each page.
- If a footer with the page number and current date should be printed on each page.
- If line numbers should be printed.
- If the output should be color or black and white.
- The tab size to use when printing - this will usually be less than the text area tab size, to conserve space in the printed output.

3.10. Closing Files and Exiting jEdit

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

File>Close All (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**. After all buffers have been closed, a new untitled buffer is opened.

File>Exit (shortcut: **Control**-**Q**) will completely exit jEdit, prompting if unsaved buffers should be saved first.

Chapter 4. Editing Text

4.1. Moving The Caret

The simplest way to move the caret is to click the mouse at the desired location in the text area. The caret 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 screen line. Pressing it a second time moves the caret to the beginning of the current buffer 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 of the current screen line, the end of the current buffer line, and finally to the last visible line.

If soft wrap is disabled, a “screen line” is the same as a “buffer line”. If soft wrap is enabled, a screen line is a section of a newline-delimited buffer line that fits within the wrap margin width. See Section 4.8.

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

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

The Home and End keys

If you prefer more traditional behavior for the **Home** and **End** keys, you can reassign the respective keyboard shortcuts in the Shortcuts pane of the Utilities>Global Options.

By default, the shortcuts are assigned as follows:

- **Home** is bound to Smart Home.
- **End** is bound to Smart End.
- **Shift**-**Home** is bound to Select to Smart Home Position.
- **Shift**-**End** is bound to Select to Smart End Position.

However you can rebind them to anything you want, for example, various combinations of the following, or indeed any other command or macro:

- Go to Start/End of White Space ,
- Go to Start/End of Line,
- Go to Start/End of Buffer,
- Select to Start/End of White Space ,
- Select to Start/End of Line,
- Select to Start/End of Buffer,

For information about changing keyboard shortcuts, see Section 6.3.

4.2. Selecting Text

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

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

Holding down **Shift** in addition to a caret movement key (**Left**, **Up**, **Home**, etc) will extend a selection in the specified direction.

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

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

4.2.1. Rectangular Selection

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

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

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

> **Note:** Rectangular selections are implemented using character offsets, not absolute screen positions, so they might not behave as you might expect if a proportional-width font is being used or if soft wrap is enabled. The text area font can be changed in the Text Area pane of the Utilities>Global Options dialog box. For information about soft wrap, see Section 4.8.

4.2.2. Multiple Selection

Edit>More Selection>Multiple Selection (keyboard shortcut: **Control**-\) turns multiple selection mode on and off. In multiple selection mode, multiple fragments of text can be selected and operated on simultaneously, and the caret can be moved independently of the selection. The status bar indicates if multiple selection mode is active; see Section 2.2.2.

Various jEdit commands behave differently with multiple selections:

- Commands that copy text place the contents of each selection, separated by line breaks, in the specified register.
- Commands that insert (or paste) text replace each selection with the entire text that is being inserted.
- Commands that filter text (such as Spaces to Tabs, Range Comment, Replace in Selection, and so on) behave as if each block was selected independently, and the command invoked on each in turn.
- Line-based commands (such as Shift Indent Left, Shift Indent Right, and Line Comment) operate on each line that contains at least one selection.
- Caret movement commands that would normally deactivate the selection (such as the arrow keys, while **Shift** is not being held down), move the caret, leaving the selection as-is.

- Some older plugins may not support multiple selection at all.

Edit>More Selection>Select None (shortcut: **Escape**) deactivates the selection containing the caret, if there is one. Otherwise it deactivates all active selections.

Edit>More Selection>Invert Selection (shortcut: **Control-E I**) selects a set of text chunks such that all text that was formerly part of a selection is now unselected, and all text that wasn't, is selected.

> **Note:** Deactivating multiple selection mode while multiple blocks of text are selected will leave the selections in place, but you will not be able to add new selections until multiple selection mode is reactivated.

4.3. Inserting and Deleting Text

Text entered at the keyboard is inserted into the buffer. If overwrite mode is on, one character is deleted from in front of the caret position for every character that is inserted. To activate overwrite mode, press **Insert**. The caret is drawn as horizontal line while in overwrite mode. The status bar also indicates if overwrite mode is active; see Section 2.2.2 for details.

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

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 5.2 for details.

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

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

4.4. Undo and Redo

Edit>Undo (shortcut: **Control-Z**) reverses the most recent 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, information about the last 100 edits is retained; older edits cannot be undone. The maximum number of undos can be changed in the Editing pane of the Utilities>Global Options dialog box.

If you undo too many changes, Edit>Redo (shortcut: **Control-R**) can restore the changes again. For example, if some text was inserted, Undo will remove it from the buffer. Redo will insert it again.

4.5. Working With Words

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

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

Control-Backspace and **Control-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**) locates possible completions for the word at the caret, first by looking in the current edit mode's syntax highlighting keyword list, and then in the current buffer for words that begin with the word at the caret. This serves as a very basic code completion feature.

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**. Typing while the popup is visible will automatically update the popup and narrow the set of completions as necessary.

For power users

The default behavior of the **Control-Left** and **Control-Right** commands is to stop both at the beginning and the end of each word. However this can be changed by remapping these keystrokes to alternative actions whose names end with (Eat Whitespace) in the Shortcuts pane of the Utilities>Global Options dialog box.

4.6. Working With Lines

An entire line can be selected by triple-clicking with the mouse, or using the Edit>More Selection>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**) prompts for a line number and moves the caret there.

Edit>More Selection>Select Line Range (shortcut **Control**-**E Control**-**L**) prompts for two line numbers and selects all text between them.

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

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

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

Edit>Text>Join Lines (shortcut: **Control**-**J**) removes any whitespace from the start of the next line and joins it with the current line. The caret is moved to the position where the two lines were joined. For example, if you invoke Join Lines with the caret on the first line of the following Java code:

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

It will be changed to:

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

4.7. Working With Paragraphs

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

Control-**Up** and **Control**-**Down** move the caret to the previous and next paragraph, respectively. Holding down **Shift** in addition to the above extends the selection a paragraph at a time.

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

Edit>Text>Format Paragraph (shortcut: **Control**-**E F**) splits and joins lines in the current paragraph to make it fit within the wrap column position. See Section 4.8 for information and word wrap and changing the wrap column.

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

4.8. Wrapping Long Lines

The *word wrap* feature splits lines at word boundaries in order to fit text within a specified wrap margin. The wrap margin position is indicated in the text are as a faint blue vertical line. There are two “wrap modes”, “soft” and “hard”; they are described below. The wrap mode can be changed in one of the following ways:

- On a global or mode-specific basis in the Editing pane of the Utilities>Global Options dialog box. See Section 6.3.
- In the current buffer for the duration of the editing session in the Utilities>Buffer Options dialog box. See Section 6.1.
- In the current buffer for future editing sessions by placing the following in one of the first or last 10 lines of the buffer, where *mode* is either “none”, “soft” or “hard”, and *column* is the desired wrap margin:

  ```
  :wrap=mode:maxLineLen=column:
  ```

4.8.1. Soft Wrap

In soft wrap mode, lines are automatically wrapped when displayed on screen. Newlines are not inserted at the wrap positions, and the wrapping is automatically updated when text is inserted or removed.

If end of line markers are enabled in the Text Area pane of the Utilities>Global Options dialog box, a colon (“:”) is painted at the end of wrapped lines.

Note that since jEdit only scrolls one whole “physical” (newline-delimited) line at a time, having lines wrapped into more sections than visible in the text area will render portions of the buffer inaccessible.

> **Tip:** If you enable soft wrap and set the wrap margin to 0, text will be wrapped to the width of the text area.

4.8.2. Hard Wrap

In hard wrap mode, inserting text at the end of a line will automatically break the line if it extends beyond the wrap margin. Inserting or removing text in the middle of a line has no effect, however text can be re-wrapped using the Edit>Text>Format Paragraph command. See Section 4.7.

Hard 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. The text area font can be changed in the Text Area pane of the Utilities>Global Options dialog box.

4.9. Scrolling

View>Scrolling>Scroll to Current Line (shortcut: **Control-E Control-J**) scrolls the text area in order to make the caret visible, if necessary. It does nothing if the caret is already visible.

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. If it is selected, scrolling one text area in a split view will scroll all other text areas in the view. Has no effect if the view is not split.

Mouse Wheel Scrolling

If you have a mouse with a scroll wheel and are running Java 2 version 1.4, you can use the wheel to scroll up and down in the text area. Various modifier keys change the action of the wheel:

- **Shift** - scrolls an entire page at a time.
- **Control** - scrolls a single line at a time.
- **Alt** - moves the caret up and down instead of scrolling.
- **Alt-Shift** - extends the selection up and down instead of scrolling.

4.10. Transferring Text

jEdit provides a rich set of commands for moving and copying text. Commands are provided for moving chunks of text from buffers to *registers* and vice-versa. A register is a holding area for an arbitrary length of text, with a single-character name. The system clipboard is mapped to the register named `$`. jEdit offers clipboard-manipulation commands similar to those found in other applications, in addition to a more flexible set of commands for working with registers directly.

4.10.1. 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.

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

The Cut and Copy commands replace the old clipboard contents with the selected text. There are two alternative commands which add the selection at the end of the existing clipboard contents, instead of replacing it.

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

Edit>More Clipboard>Copy Append (shortcut: **Control**-**E** **Control**-**A**) is the same as Cut Append except it does not remove the selection from the buffer.

4.10.2. Quick Copy

Quick copy is disabled by default, but it can be enabled in the Text Area pane of the Utilities>Global Options dialog box. When quick copy is enabled, clicking the middle mouse button in the text area inserts the most recently selected text at the clicked location. If you only have a two-button mouse, you can click the left mouse button while holding down **Alt** instead of middle-clicking.

This is implemented by storing the most recently selected text in the register named `%`.

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

Also, dragging with the middle mouse button creates a selection without moving the caret. As soon as the mouse button is released, the selected text is inserted at the caret position and the selection is deactivated. A message is shown in the status bar while text is being selected to remind you that this is not an ordinary selection.

4.10.3. General Register Commands

These commands require more keystrokes than the two methods shown above, but they can operate on any register, allowing an arbitrary number of text chunks to be retained at a time.

Each command prompts for a single-character register name to be entered after being invoked. Pressing **Escape** instead of specifying a register name will cancel the operation.

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

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

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

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

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

The last two commands display dialog boxes instead of prompting for a register name.

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

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

4.11. Markers

A *marker* is a pointer to a specific location within a buffer, which may or may not have a single-character *shortcut* associated with it. Markers are persistent; they are saved to `.`*`filename`*`.marks`, where *`filename`* is the name of the buffer. (The dot prefix makes the

markers file hidden on Unix systems.) Marker saving can be disabled in the Loading and Saving pane of the Utilities>Global Options dialog box; see Section 6.3.

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

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

Markers are listed in the Markers menu; selecting a marker from this menu will move the caret to its location.

Markers>Go to Previous Marker (shortcut: **Control-E Control-,**) goes to the marker immediately before the caret position.

Markers>Go to Next Marker (shortcut: **Control-E Control-.**) goes to the marker immediately after the caret position.

Markers with shortcuts allow for quicker keyboard-based navigation. The following commands all prompt for a single-character shortcut when invoked. Pressing **Escape** instead of specifying a shortcut will cancel the operation.

Markers>Add Marker With Shortcut (shortcut: **Control-T *key***) adds a marker with the specified shortcut. If marker with that shortcut already exists, it will remain in the buffer but lose its shortcut.

Markers>Go to Marker (shortcut: **Control-Y *key***) moves the caret to the location of the marker with the specified shortcut.

Markers>Select to Marker (shortcut: **Control-U *key***) creates a selection from the caret location to the marker with the specified shortcut.

Markers>Swap Caret and Marker (shortcut: **Control-U *key***) moves the caret to the location of the marker with the specified shortcut, and reassigns the marker to point to the former caret location. Invoke this command multiple times to flip between two locations in the buffer.

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

4.12. Search and Replace

4.12.1. 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. This text field remembers previously entered strings; see Appendix C for details.

If text was selected in the text area and the selection does not span a line break, the selected text becomes the default search string.

If the selection spans a line break, the Search in Selection and HyperSearch buttons will be pre-selected, and the search string field will be initially blank. (See Section 4.12.3 for information about the HyperSearch feature.)

Selecting the Ignore case check box makes the search case insensitive - for example, searching for “Hello” will match “hello”, “HELLO” and “HeLlO”.

Selecting the Regular expressions check box allows a regular expression to be used in the search string. Regular expressions can match inexact sequences of text that optionally span more than one line. Regular expression syntax is described in Appendix E.

The Backward and Forward buttons specify the search direction. Note that regular expressions can only be used when searching in a forward direction.

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

If no occurrences could be found and the Auto wrap check box is selected, the search will automatically restart from the beginning of the buffer (or the end, if searching backwards). If Auto wrap is not selected, a confirmation dialog box is shown before restarting the search.

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

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

4.12.2. Replacing Text

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

Clicking Replace & Find will perform a replacement in 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 in the current search scope (which is either the selection, the current buffer, or a set of buffers, as specified in the search and replace dialog box).

Occurrences of the search string can be replaced with either a replacement string, or the return value of a BeanShell script snippet. Two radio buttons in the search and replace

dialog box select between the two replacement modes, which are described in detail below.

4.12.2.1. Text Replace

If the Text button is selected, the search string is simply replaced with the replacement string.

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

If the search is case-insensitive, jEdit attempts to modify the case of the replacement string to match that of the particular instance of the search string being replaced. For example, searching for “label” and replacing it with “text”, will perform the following replacements:

- “String label” would become “String text”
- “setLabel” would become “setText”
- “DEFAULT_LABEL” would become “DEFAULT_TEXT”

4.12.2.2. BeanShell Replace

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

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

BeanShell syntax and features are covered in great detail in Part III in *jEdit 4.1 User’s Guide*, but here are some examples:

To replace each occurrence of “Windows” with “Linux”, and each occurrence of “Linux” with “Windows”, search for the following regular expression:

```
(Windows|Linux)
```

Replacing it with the following BeanShell snippet:

```
_1.equals("Windows") ? "Linux" : "Windows"
```

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

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

Replacing it with the following BeanShell snippet:

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

To replace arithmetic expressions contained in curly braces with the result of evaluating the expression, search for the following regular expression:

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

Replacing it with the following BeanShell snippet:

```
eval(_1)
```

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

4.12.3. HyperSearch

If the HyperSearch check box in the search and replace dialog box is selected, clicking Find lists all occurrences of the search string, instead of locating the next match.

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

If the Multiple results check box is selected in the results window, past search results are retained.

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

4.12.4. Multiple File Search

Search and replace commands can be performed over an arbitrary set of files in one step. The set of files to search is selected with a set of buttons in the search dialog box.

If the Current buffer button is selected, only the current buffer is searched. This is the default behavior.

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

If the Directory radio button is selected, all files contained in the specified directory whose names match the glob will be searched. The directory to search in can either be entered in the Directory text field, or chosen in a file selector dialog box by clicking the Choose button next to the field. 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 containing many files can take a long time.

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

Note that clicking the All Buffers or Directory radio buttons also selects the HyperSearch check box since that is what you would want, most of the time. However, normal match-by-match searching is supported for multiple files as well.

Two convenience commands are provided for performing multiple file searches.

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

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

4.12.5. The Search Bar

The search bar feature provides a convenient way to search in the current buffer without opening the search dialog box. The search bar does not support replacement or multiple file. Previously entered strings can be recalled in the search bar with the **Up** and **Down** arrow keys; see Appendix C.

By default, the search bar remains hidden until one of the quick search commands (described below) is invoked; however you can choose to have it always visible in the General pane of the Utilities>Global Options dialog box.

Search>Incremental Search Bar (shortcut: **Control-,**) displays the search bar if necessary, and gives it keyboard focus. If this command is invoked while there is a selection, the selection is placed in the search string field.

Search>Incremental Search for Word (shortcut: **Alt-,**) behaves like the above command except it places the word at the caret in the search string field.

Unless the HyperSearch check box is selected, the search bar will perform an *incremental search*. In incremental search mode, the first occurrence of the search string is located in the current buffer as it is being typed. Pressing **Enter** and **Shift-Enter** searches for the next and previous occurrence, respectively. Once the desired occurrence has been located, pressing **Escape** returns keyboard focus to the text area. Unless the search bar is set to be always visible (see above), pressing **Escape** will also hide the search bar.

> **Note:** Incremental searches cannot be not recorded in macros. If your macro needs to perform a search, use the search and replace dialog box instead. See Chapter 7 for information about macros.

Search>HyperSearch Bar (shortcut: **Control-.**) displays the search bar if necessary, gives it keyboard focus, and selects the HyperSearch check box. If this command is invoked while there is a selection, the selected text will be searched for immediately and the search bar will not be shown.

If the HyperSearch check box is selected, pressing **Enter** in the search string field will perform a HyperSearch in the current buffer.

Search>HyperSearch for Word (shortcut: **Alt-.**) performs a HyperSearch for the word at the caret. This command does not show the search bar or give it keyboard focus.

4.13. Command Repetition

The final feature discussed in this chapter provides a way to repeat a command any number of times.

To repeat a command multiple times, press **Control-Enter**, enter the desired repeat count, then invoke the command to repeat (either using a keyboard shortcut, or by selecting it from the menu bar). For example, "**Control-Enter 1 4 Control-D**" will delete 14 lines; "**Control-Enter 9 #**" will insert "#########" in the buffer.

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

Chapter 5. Editing Source Code

5.1. Edit Modes

An *edit mode* specifies syntax highlighting rules, auto indent behavior, and various other customizations for editing a certain file type. This section only covers using existing edit modes; information about writing your own can be found in Part II in *jEdit 4.1 User's Guide*.

5.1.1. Mode Selection

When a file is opened, jEdit first checks the file name against a list of known patterns. For example, files whose names end with ".c" are opened with C mode, and files named `Makefile` are opened with 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 opened with shell script mode.

File name and first line globs can be changed in the Editing pane of the Utilities>Global Options dialog box. See Appendix D for information about glob patterns.

The edit mode can be specified manually as well. The current buffer's edit mode can be set on a one-time basis in the Utilities>Buffer Options dialog box; see Section 6.1. To set a buffer's edit mode for future editing sessions, place the following in one of the first or last 10 lines of the buffer, where *`edit mode`* is the name of the desired edit mode:

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

A list of edit modes can be found in the Utilities>Buffer Options dialog box.

5.1.2. Syntax Highlighting

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

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

5.2. Tabbing and Indentation

jEdit makes a distinction between the *tab width*, which is is used when displaying hard 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 auto indent routines. Both can be changed in one of several ways:

- On a global or mode-specific basis in the Editing pane 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 or last 10 lines of the buffer, where *n* is the desired tab width, and *m* is the desired indent width:

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

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

Edit>Indent>Shift Indent Right (shortcut: **Alt**-**Right**) removes one level of indent from each selected line, or the current line if there is no selection. Pressing **Tab** while a multi-line selection is active has the same effect.

Edit>Indent>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.

5.2.1. Soft Tabs

Files containing hard tab characters may look less than ideal if the default tab size is changed, so some people prefer using multiple space characters instead of hard tabs to indent code.

This feature is known as *soft tabs*. Soft tabs can be enabled or disabled in one of several ways:

- On a global or mode-specific basis in the Editing pane 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 or last 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>Spaces to Tabs converts soft tabs to hard tabs in the current selection, or the entire buffer if nothing is selected.

Edit>Source>Tabs to Spaces converts hard tabs to soft tabs in the current selection, or the entire buffer if nothing is selected.

5.2.2. Automatic Indent

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

In the default configuration, pressing **Enter** will create a new line with the appropriate amount of indent automatically, and pressing **Tab** at the beginning of, or inside the leading whitespace of a line will insert the appropriate amount of indentation. Pressing it again will insert a tab character.

The behavior of the **Enter** and **Tab** keys can be configured in the Shortcuts pane of the Utilities>Global Options dialog. box, just as with any other key. The **Enter** key can be bound to one of the following, or indeed any other command or macro:

- Insert Newline.
- Insert Newline and Indent, which is the default.

The **Tab** can be bound to one of the following, or again, any other command or macro:

- Insert Tab.
- Insert Tab or Indent, which is the default.
- Indent Selected Lines.

See Section 6.3 for details.

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>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**.

5.3. Commenting Out Code

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

Comment strings are mode-specific, and some in some modes such as HTML different parts of a buffer can have different comment strings. For example, in HTML files, different comment strings are used for HTML text and inline JavaScript.

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

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

5.4. 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.

Positioning the caret immediately before or after a bracket will highlight the corresponding closing or opening bracket (assuming it is visible), and draw a scope indicator in the gutter. If the highlighted bracket is not visible, the text of the matching line will be shown in the status bar. If the matching line consists of only whitespace and the bracket itself, the *previous line* is shown instead. This feature is very useful when your code is indented as follows, with braces on their own lines:

```
public void someMethod()
{
    if(isOK)
    {
        doSomething();
    }
}
```

Invoking Edit>Source>Go to Matching Bracket (shortcut: **Control-]**) or clicking the scope indicator in the gutter moves the caret to the matching bracket.

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

Holding down **Control** while clicking the scope indicator in the gutter or a bracket in the text area will select all text between the two matching brackets.

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

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

Bracket highlighting in the text area and bracket scope display in the gutter can be customized in the Text Area and Gutter panes of the Utilities>Global Options dialog box; see Section 6.3.

> **Note:** jEdit's bracket matching algorithm only checks syntax tokens with the same type as the original bracket, so for example unmatched brackets inside string literals and comments will be skipped when matching brackets that are part of program syntax.

5.5. 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-;**. An abbreviation can either be global, in which case it can be used in all edit modes, or specific to a single mode.

Abbreviations can be edited in the Abbreviations pane of the Utilities>Global Options dialog box; see Section 6.3. The Java, VHDL. XML and XSL edit modes include some pre-defined abbreviations you might find useful. Other modes do not have any abbreviations defined by default.

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

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

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

5.5.1. Positional Parameters

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

Java mode defines an abbreviation "F" that is set to expand to the following:

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

Expanding `F#j#array.length#` will insert the following text into the buffer:

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

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

If you do not specify the correct number of positional parameters when expanding an abbreviation, any missing parameters will be blank in the expansion, and extra parameters will be ignored. A status bar message will be shown stating the required number of parameters.

5.6. Folding

Program source code and other structured text files can be thought of as containing a hierarchy of sections, which themselves might contain sub-sections. The folding feature lets you selectively hide and show these sections, replacing hidden ones with a single line that serves as an “overview” of that section.

Folding is disabled by default. To enable it, you must choose one of the available folding modes. “Indent” mode creates folds based on a line’s leading whitespace; the more leading whitespace a block of text has, the further down it is in the hierarchy. For example:

```
This is a section
  This is a sub-section
  This is another sub-section
    This is a sub-sub-section
Another top-level section
```

“Explicit” mode folds away blocks of text surrounded with “{{{” and “}}}”. For example:

```
{{{ The first line of a fold.
When this fold is collapsed, only the above line will be visible.

{{{ A sub-section.
With text inside it.
}}}

{{{ Another sub-section.
}}}

}}}
```

Both modes have distinct advantages and disadvantages; indent folding requires no changes to be made to a buffer's text and does a decent job with most program source. Explicit folding requires "fold markers" to be inserted into the text, but is more flexible in exactly what to fold away.

Some plugins might add additional folding modes; see Chapter 8 for information about plugins.

Folding can be enabled in one of several ways:

- On a global or mode-specific basis in the Editing pane 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 or last 10 lines of a buffer, where *mode* is either "indent", "explicit", or the name of a plugin folding mode:

  `:folding=`*mode*`:`

> **Warning**
>
> When using indent folding, portions of the buffer may become inaccessible if you change the leading indent of the first line of a collapsed fold. If you experience this, you can use the Expand All Folds command to make the text visible again.

5.6.1. Collapsing and Expanding Folds

The first line of each fold has a triangle drawn next to it in the gutter (see Section 2.3 for more information about the gutter). The triangle points toward the line when the fold is collapsed, and downward when the fold is expanded. Clicking the triangle collapses and expands the fold. To expand all sub-folds as well, hold down the **Shift** while clicking.

The first line of a collapsed fold is drawn with a different background color, and the number of lines in the fold is shown to the right of the line's text.

Folds can also be collapsed and expanded using menu item commands and keyboard shortcuts.

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

Folding>Expand Fold One Level (keyboard shortcut: **Alt**-**Enter**) expands the fold containing the caret position. Nested folds will remain collapsed, and the caret is positioned on the first nested fold (if any).

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

Folding>Collapse All Folds (keyboard shortcut: **Control**-**E C**) collapses all folds in the buffer.

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

5.6.2. Navigating Around With Folds

Folding>Go to Parent Fold (keyboard shortcut: **Control**-**e u**) moves the caret to the fold containing the one at the caret position.

Folding>Go to Previous Fold (keyboard shortcut: **Alt**-**Up**) moves the caret to the fold immediately before the caret position.

Folding>Go to Next Fold (keyboard shortcut: **Alt**-**Down**) moves the caret to the fold immediately after the caret position.

5.6.3. Miscellaneous Folding Commands

Folding>Add Explicit Fold (keyboard shortcut: **Control**-**E A**) is a convenience command that surrounds the selection with "{{{" and "}}}". If the current buffer's edit mode defines comment strings (see Section 5.3) the explicit fold markers will automatically be commented out as well.

Folding>Select Fold (keyboard shortcut: **Control**-**E S**) selects all lines within the fold containing the caret position. **Control**-clicking a fold expansion triangle in the gutter has the same effect.

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

Sometimes it is desirable to have files open with folds initially collapsed. This can be configured as follows:

- On a global or mode-specific basis in the Editing pane of the Utilities>Global Options dialog box.
- In the current buffer for future editing sessions by placing the following in the first or last 10 lines of a buffer, where `level` is the desired fold level:

`:collapseFolds=`*`level`*`:`

5.6.4. Narrowing

The narrowing feature temporarily “narrows” the display of a buffer to a specified region. Text outside the region is not shown, but is still present in the buffer.

Folding>Narrow Buffer to Fold (keyboard shortcut: **Control-E N N**) hides all lines the buffer except those in the fold containing the caret.

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

Folding>Expand All Folds (keyboard shortcut: **Control-E X**) shows lines that were hidden as a result of 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. Changes made in this dialog box are not retained after the buffer is closed.

The following settings can be changed here:

- The line separator (see Section 3.4)
- The character encoding (see Section 3.5)
- If the file should be GZipped on disk (see Section 3.2)
- The edit mode (see Section 5.1)
- The fold mode (see Section 5.6)
- The wrap mode and margin (see Section 4.8)
- The tab width (see Section 5.2)
- The indent width
- If soft tabs should be used (see Section 5.2)

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 and last 10 lines for colon-enclosed name/value pairs. For example, placing the following in a buffer changes the indent width to 4 characters, enables soft tabs, and activates the Perl edit mode:

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

Adding buffer-local properties to a buffer takes effect after the next time the buffer is saved.

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

Property name	Description

Property name	Description
`collapseFolds`	Folds with a level of this or higher will be collapsed when the buffer is opened. If set to zero, all folds will be expanded initially. See Section 5.6.
`folding`	The fold mode; one of “none”, “indent”, “explicit”, or the name of a plugin folding mode. See Section 5.6.
`indentSize`	The width, in characters, of one indent. Must be an integer greater than 0. See Section 5.2.
`maxLineLen`	The maximum line length and wrap column position. Inserting text beyond this column will automatically insert a line break at the appropriate position. See Section 4.3.
`mode`	The default edit mode for the buffer. See Section 5.1.
`noTabs`	If set to “true”, soft tabs (multiple space characters) will be used instead of “real” tabs. See Section 5.2.
`noWordSep`	A list of non-alphanumeric characters that are *not* to be treated as word separators. Global default is “_”.
`tabSize`	The tab width. Must be an integer greater than 0. See Section 5.2.
`wordBreakChars`	Characters, in addition to spaces and tabs, at which lines may be split when word wrapping. See Section 4.3.
`wrap`	The word wrap mode; one of “none”, “soft”, or “hard”. See Section 4.8.

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.

6.3.1. The Abbreviations Pane

The Abbreviations option pane can be used to enable or disable automatic abbreviation expansion, and to edit currently defined abbreviations.

The combo box labeled “Abbrev set” selects the abbreviation set to edit. The first entry, “global”, contains abbreviations available in all edit modes. The subsequent entries correspond to each mode’s local set of abbreviations.

To change an abbreviation or its expansion, either double-click the appropriate table entry, or click a table entry and then click the Edit button. This will display a dialog box

for modifying the abbreviation.

The Add button displays a dialog box where you can define a new abbreviation. The Remove button removes the currently selected abbreviation from the list.

See Section 5.5.1 for information about positional parameters in abbreviations.

6.3.2. The Appearance Pane

The Appearance pane can be used to change the appearance of user interface controls such as buttons, labels and menus.

6.3.3. The Context Menu Pane

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

6.3.4. The Docking Pane

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

6.3.5. The Editing Pane

The Editing option pane contains settings such as the tab size, syntax highlighting and soft tabs on a global or mode-specific basis.

When changing mode-specific settings, 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. See Appendix D for information about glob patterns.

This option pane does not change XML mode definition files on disk; it merely writes values to the user properties file which override those set in mode files. To find out how to edit mode files directly, see Part II in *jEdit 4.1 User's Guide*.

6.3.6. The General Pane

The General pane contains various miscellaneous settings, such as the number of recent files to remember, if the buffer list should be sorted, and so on.

6.3.7. The Gutter Pane

The Gutter option pane contains settings to customize the appearance of the gutter.

6.3.8. The Loading and Saving Pane

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

6.3.9. The Printing Pane

The Printing option pane contains settings to control the appearance of printed output.

6.3.10. The Proxy Servers Pane

The Proxy Servers option pane lets you specify HTTP and SOCKS proxy servers to use when jEdit makes network connections (for example, when the plugin manager downloads plugins).

6.3.11. The Shortcuts Pane

The Shortcuts option pane associates keyboard shortcuts with commands. Each command can have up to two shortcuts associated with it.

The combo box at the top of the option pane selects the command set to edit. Command sets include the set of all built-in commands, the commands of each plugin, and the set of macros.

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

6.3.12. The Status Bar Pane

The Status Bar option pane contains settings to customize the status bar, or disable it completely.

6.3.13. The Syntax Highlighting Pane

The Syntax Highlighting pane can be used to customize the fonts and colors for syntax highlighting.

6.3.14. The Text Area Pane

The Text Area pane contains settings to customize the appearance of the text area.

6.3.15. The Tool Bar Pane

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

6.3.16. The File System Browser Panes

The File System Browser group contains two option panes, General and Colors. The former contains various file system browser settings. The latter configures glob patterns used for coloring the file list. See Section 3.6 for more information.

6.4. The jEdit Settings Directory

jEdit stores settings, macros, and plugins as files inside the *settings directory*. In most cases, editing these files by hand is not necessary, since graphical tools and editor 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>Troubleshooting>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 directory other than the default. For example, the following command will instruct jEdit to store all settings in the `jedit` subdirectory of the `C:` drive:

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

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

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

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

- `abbrevs` - a plain text file which stores all defined abbreviations. See Section 5.5.
- `activity.log` - a plain text file which contains the full activity log. See Appendix B.

- `history` - a plain text file which stores history lists, used by history text fields and the Edit>Paste Previous command. See Section 4.10 and Appendix C.
- `jars` - this directory contains plugins. See Chapter 8.
- `macros` - this directory contains macros. See Chapter 7.
- `modes` - this directory contains custom edit modes. See Part II in *jEdit 4.1 User's Guide*.
- `PluginManager.download` - this directory is usually empty. It only contains files while the plugin manager is downloading a plugin. For information about the plugin manager, see Chapter 8.
- `printspec` - a binary file which stores printing settings when running under Java 2 version 1.4.
- `properties` - a plain text file which stores the majority of jEdit's settings.
- `recent.xml` - an XML file which stores the list of recently opened files. jEdit remembers the caret position and character encoding of each recent file, and automatically restores those values when one of the files is opened.
- `server` - a plain text file that only exists while jEdit is running. The edit server's port number and authorization key is stored here. See Chapter 1.
- `session` - a list of files, used when restoring previously open files on startup.
- `settings-backups` - this directory contains numbered backups of all automatically-written settings files (`abbrevs`, `activity.log`, `history`, `properties`, `recent.xml`, and `session`).

Chapter 7. Using Macros

Macros in jEdit are short scripts written in a scripting language called *BeanShell*. They provide an easy way to automate repetitive keyboard and menu procedures, as well as access to the objects and methods created by jEdit. Macros also provide a powerful facility for customizing jEdit and automating complex text processing and programming tasks. This section describes how to record and run macros. A detailed guide on writing macros appears later; see Part III in *jEdit 4.1 User's Guide*.

7.1. Recording Macros

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

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

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

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

Mouse clicks in the text area are *not* recorded; use text selection commands or arrow keys instead.

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 it.

The file name extension `.bsh` is automatically appended to the macro name, and all spaces are converted to underscore characters, in order to make the macro name a valid file name. These two operations are reversed when macros are displayed in the Macros menu; see Section 7.3 for details.

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

Macros>Record Temporary Macro (shortcut: **Control-M Control-M**) begins recording to a buffer named `Temporary_Macro.bsh`. Once recording of a temporary macro is complete, jEdit does not display the buffer containing the recorded commands, but the name `Temporary_Macro.bsh` will be visible on any list of open buffers. By switching to that buffer, you can view the commands, edit them, and save them if you wish to a permanent macro file. Whether or not you look at or save the temporary macro contents, it is immediately available for playback.

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

Only one temporary macro is available at a time. If you begin recording a second temporary macro, the first is erased and cannot be recovered unless you have saved the contents to a file with a name other than `Temporary_Macro.bsh`. If you do not save the temporary macro, you must keep the buffer containing the macro script open during your jEdit session. To have the macro available for your next jEdit session, save the buffer `Temporary_Macro.bsh` as an ordinary macro with a descriptive name of your choice. The new name will then be displayed in the Macros menu.

7.2. Running Macros

Macros supplied with jEdit, as well as macros that you record or write, are displayed under the Macros menu in a hierarchical structure. The jEdit installation includes about 30 macros divided into several major categories. Each category corresponds to a nested submenu under the Macros menu. An index of these macros containing short descriptions and usage notes is found in Appendix F.

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

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

7.3. How jEdit Organizes Macros

Every macro, whether or not you originally recorded it, is stored on disk and can be edited as a text file. The file name of a macro must have a `.bsh` extension in order for jEdit to be aware of it. 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 home directory, and the `macros` subdirectory of the user-specific settings directory (see Section 6.4 for information about the settings directory). Any macros you record will be stored in the user-specific directory.

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

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

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

You can browse the user and system macro directories by opening the `macros` directory from the Utilities>jEdit Home Directory and Utilities>Settings Directory menus.

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

Chapter 8. Installing and Using Plugins

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

This chapter covers installing, updating and removing plugins. Documentation for the plugins themselves can be found in Help>jEdit Help, and information about writing plugins can be found in Part IV in *jEdit 4.1 User's Guide*.

8.1. The Plugin Manager

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

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

8.2. Installing Plugins

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

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

To install plugins from the plugin manager, make sure you are connected to the Internet and click the Install Plugins button in the plugin manager window. The plugin manager will then download information about available plugins from the jEdit web site, and present a list of plugins compatible with your jEdit release which may be installed.

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

The Total download size field shows the total size of all plugins chosen for installation, along with any plugins that will be automatically downloaded in order to fulfill dependencies. The Download size field in the plugin information area only shows the size of the currently selected plugin.

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

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

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

> **Proxy Servers and Firewalls**
>
> If you are connected to the Internet through an HTTP proxy or SOCKS firewall, you will need to specify the relevant details in the Proxy Servers pane of the Utilities>Global Options dialog box.

8.3. Updating Plugins

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

Appendix A. Keyboard Shortcuts

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

Files

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

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

Views

For details, see Section 2.2.

Control-E Control-T	Turn gutter (line numbering) on and off.
Control-0	Remove split containing current text area only.
Control-1	Remove all splits.
Control-2	Split view horizontally.
Control-3	Split view vertically.
Alt-Page Up	Send keyboard focus to previous text area.
Alt-Page Down	Send keyboard focus to next text area.
Control-E Control-Up; **Control-Left**; **Control-Down**; **Control-Right**	Send keyboard focus to top; bottom; left; right docking area.
Control-E Control-‘	Close currently focused docking area.
Control-E Control-E	Send keyboard focus back to current text area.

Repeating

For details, see Section 4.13.

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

Moving the Caret

For details, see Section 4.1, Section 4.5, Section 4.6, Section 4.7 and Section 5.4.

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

Selecting Text

For details, see Section 4.2, Section 4.5, Section 4.6, Section 4.7 and Section 5.4.

Shift-*Arrow*	Extend selection by one character or line.
Control-Shift-*Arrow*	Extend selection by one word or paragraph.
Shift-Page Up; **Shift-Page Down**	Extend selection by one screenful.
Shift-Home	Extend selection to first non-whitespace character of line, beginning of line, first visible line (repeated presses).
Shift-End	Extend selection to last non-whitespace character of line, end of line, last visible line (repeated presses).
Control-Shift-Home	Extend selection to beginning of buffer.
Control-Shift-End	Extend selection to end of buffer.
Control-[	Select code block.

Control-E W; **L**; **P**	Select word; line; paragraph.
Control-E Control-L	Select line range.
Control-	Switch between single and multiple selection mode.

Scrolling

For details, see Section 2.2.

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

Text Editing

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

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

Clipboard and Registers

For details, see Section 4.10.

Control-X or **Shift-Delete**	Cut selected text to clipboard.
Control-C or **Control-Insert**	Copy selected text to clipboard.
Control-E Control-U	Append selected text to clipboard, removing it from the buffer.

Control-E Control-A	Append selected text to clipboard, leaving it in the buffer.
Control-V or **Shift-Insert**	Paste clipboard contents.
Control-E Control-P	Vertically paste clipboard contents.
Control-R Control-X ***key***	Cut selected text to register *key*.
Control-R Control-C ***key***	Copy selected text to register *key*.
Control-R Control-U ***key***	Append selected text to register *key*, removing it from the buffer.
Control-R Control-A ***key***	Append selected text to register *key*, leaving it in the buffer.
Control-R Control-V ***key***	Paste contents of register *key*.
Control-R Control-P ***key***	Vertically paste contents of register *key*.
Control-E Control-V	Paste previous.

Markers

For details, see Section 4.11.

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

Search and Replace

For details, see Section 4.12.

Control-F	Open search and replace dialog box.
Control-G	Find next.
Control-H	Find previous.
Control-E Control-B	Search in open buffers.
Control-E Control-D	Search in directory.
Control-E Control-R	Replace in selection.

Control-E Control-G	Replace in selection and find next.
Control-,	Incremental search bar.
Control-.	HyperSearch bar.
Alt-,	Incremental search for word under the caret.
Alt-.	HyperSearch for word under the caret.

Source Code Editing

For details, see Section 5.5, Section 5.2 and Section 5.3.

Control-;	Expand abbreviation.
Alt-Left; **Alt-Right**	Shift current line (or all selected lines) left; right.
Shift-Tab; **Tab**	Shift selected lines left; right. Note that pressing **Tab** with no selection active will insert a tab character at the caret position.
Control-I	Indent current line (or all selected lines).
Control-E Control-C	Wing comment selection.
Control-E Control-B	Box comment selection.

Folding and Narrowing

For details, see Section 5.6 and Section 5.6.4.

Alt-Backspace	Collapse fold containing caret.
Alt-Enter	Expand fold containing caret one level only.
Alt-Shift-Enter	Expand fold containing caret fully.
Control-E X	Expand all folds.
Control-E A	Add explicit fold.
Control-E S	Select fold.
Control-E Enter *`key`*	Expand folds with level less than *`key`*, collapse all others.
Control-E N N	Narrow to fold.
Control-E N S	Narrow to selection.
Alt-Up Alt-Down	Moves caret to previous; next fold.
Control-E U	Moves caret to the parent fold of the one containing the caret.

Macros

For details, see Chapter 7.

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

Alternative Shortcuts

A few frequently-used commands have alternative shortcuts intended to help you keep your hands from moving all over the keyboard.

Alt-J; **Alt-L**	Move caret to previous, next character.
Alt-I; **Alt-K**	Move caret up, down one line.
Alt-Q; **Alt-A**	Move caret up, down one screenful.
Alt-Z	First non-whitespace character of line, beginning of line, first visible line (repeated presses).
Alt-X	Last non-whitespace character of line, end of line, last visible line (repeated presses).

Appendix B. The Activity Log

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

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

jEdit writes the following information to the activity log:

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

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

Appendix C. History Text Fields

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

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

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

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

Appendix D. Glob Patterns

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

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

In addition to the above, a number of "character class expressions" may be used as well:

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

Here are some examples of glob patterns:

- **`*`** - all files.
- **`*.java`** - all files whose names end with ".java".
- **`*.[ch]`** - all files whose names end with either ".c" or ".h".

- **`[^#]*`** - all files whose names do not start with “#”.

Appendix E. Regular Expressions

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

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

Positional Operators

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

One-Character Operators

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

Character Class Operator

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

- `[[:alpha:]]` matches any alphabetical character
- `[[:blank:]]` matches a space or horizontal tab
- `[[:cntrl:]]` matches a control character
- `[[:digit:]]` matches a decimal digit
- `[[:graph:]]` matches a non-space, non-control character
- `[[:lower:]]` matches a lowercase letter
- `[[:print:]]` same as `[[:graph:]]`, but also space and tab
- `[[:punct:]]` matches a punctuation character
- `[[:space:]]` matches any whitespace character, including newlines
- `[[:upper:]]` matches an uppercase letter
- `[[:xdigit:]]` matches a valid hexadecimal digit

Subexpressions and Backreferences

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

Branching (Alternation) Operator

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

Repeating Operators

These symbols operate on the previous atomic expression.

- `?` matches the preceding expression or the null string
- `*` matches the null string or any number of repetitions of the preceding expression

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

Stingy (Minimal) Matching

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

Appendix F. Macros Included With jEdit

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

In addition to the macros included with jEdit, a very large collection of user-contributed macros is available in the “Downloads” section of the `community.jedit.org` web site. There are detailed descriptions for each macro as well as a search facility.

F.1. File Management Macros

These macros automate the opening and closing of files.

- `Browse_Directory.bsh`

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

- `Close_All_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.

F.2. 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.

- `Preview_Javadoc_of_Buffer.bsh`

  Creates and displays javadoc for current buffer.

  The macro includes configuration variables for using different doclets for generating javadocs and for generating javadocs of the package of which the current buffer is a part. Details for use are included in the note accompanying the macro's source code.

F.3. Macros for Listing Properties

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

- `jEdit_Properties.bsh`

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

- `System_Properties.bsh`

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

- `Look_and_Feel_Properties.bsh`

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

F.4. 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.

- `Display_Abbreviations.bsh`

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

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

- `Display_Shortcuts.bsh`

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

  The macro provides a combined read-only view of command, macro and plugin shortcuts. Pressing a letter key will scroll the table to the first entry beginning with that letter. A further option is provided to write the shortcut assignments in a text

buffer for printing as a reference. Notes in the source code listing point out some display options that are configured by modifying global variables.

- `Evaluate_Buffer_in_BeanShell.bsh`

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

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

- `Hex_Convert.bsh`

  Converts byte characters to their hex equivalent, and vice versa.

- `Include_Guard.bsh`

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

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

- `Make_Bug_Report.bsh`

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

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

- `Run_Script.bsh`

  Runs script using interpreter based upon buffer’s editing mode (by default, determined using file extension). You must have the appropriate interpreter (such as Perl, Python, or Windows Script Host) installed on your system.

- `Show_Threads.bsh`

  Displays in a tree format all running Java threads of the current Java Virtual Machine.

- `Write_HyperSearch_Results.bsh`

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

F.5. Text Macros

These macros generate various forms of formatted text.

- `Add_Prefix_and_Suffix.bsh`

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

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

- `Color_Picker.bsh`

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

- `Duplicate_Line.bsh`

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

- `Insert_Date.bsh`

  Inserts the current date and time in the current buffer.

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

- `Insert_Tag.bsh`

  Inserts a balanced pair of HTML/SGML/XML markup tags as supplied in a input dialog. The tags will surround any selected text.

- `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. See Section 14.1.4.

- `Toggle_Line_Comment.bsh`

  Toggles line comments, alternately inserting and deleting them at the beginning of each selected line. If there is no selection, the macro operates on the current line.

  A “line comment” is a token that designates the entire contents of a line as commented text; it does not use or require a closing token. If the editing mode does not provide for line comments (for example, text or XML modes), the macro will display an error message.

Appendix G. jEditLauncher for Windows

G.1. Introduction

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

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

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

G.2. Starting jEdit

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

jedit.exe supports five command-line options. Except for the **`/1`** option, if any of these options are invoked correctly, the application will not load files or execute jEdit.

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

Using the dialog, you can change parameters specifying the executable for the Java application loader (either `java.exe` or `javaw.exe`), the location of the jEdit archive file, `jedit.jar`, and command line options for both.

- The input fields for Java options and jEdit options are separate. If you insert an option in the wrong place it will not be properly executed.
- If the **`-jar`** option is not used with the Java application loader the principal jEdit class of `org.gjt.sp.jedit.jEdit` is set as fixed data.
- The working directory for the Java interpreter's process can also be specified.

A read-only window at the bottom of the dialog displays the full command line that jEditLauncher will invoke.

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

Note that due to the design of jEditLauncher, platform-independent command line options handled by jEdit itself (such as **`-background`** and **`-norestore`**) must be entered in the "Set jEdit Parameters" dialog box, and cannot be specified on the **jedit.exe** command line directly. For information about platform-independent command line options, see Section 1.4.

- The option **`/1`** is intended for use in circumstances where a single file name is passed to jEdit for opening, and quotation marks cannot be used to delimit file names containing whitespace. The launcher reads the entire command line following the **`/1`** options as a single file path, regardless of the presence of whitespace, and passes the resulting string as a single file name parameter to jEdit.

  This option allows jEdit to be used with version 5 or greater of Internet Explorer as an alternate text editor or as the target of the View Source command. Included with the jEditLauncher distribution is a file named `jEdit_IE.reg.txt` containing an example of a Window registry file that you can use to register jEdit as a HTML editor with Internet Explorer. Instructions for the file's use are included in the text.

  The use of the **`/1`** option with multiple file names or other parameters will lead to program errors or unpredictable results.

- The option **`/i`** is not mentioned in the help window for `jedit.exe`. It is intended primarily to be used in conjunction with jEdit's Java installer, but it can also be used to install or reinstall jEditLauncher manually. When accompanied by a second parameter specifying the directory where your preferred Java interpreter is located,

jEditLauncher will install itself and set a reasonable initial set of command line parameters for executing jEdit. You can change these parameters later by running `jedinit.exe` or `jedit.exe` with the **`/p`** option.

- The option **`/u`** will cause jEditLauncher to be uninstalled by removing its registry entries. This option does not delete any jEditLauncher or jEdit files.

G.3. The Context Menu Handler

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

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

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

G.4. Using jEdit and jEditLauncher as a Diff Utility

As noted above, you can create a graphical diff display comparing the contents of two text files by selecting the two files in an Explorer window, right-clicking to produce a context menu, and selecting the Show diff in jEdit menu item. The utility `jedidiff.exe` allows you to perform this operation from a command line. The command takes the two files to be compared as parameters; they should be delimited by quotation marks if their paths contain whitespace.

G.5. Uninstalling jEdit and jEditLauncher

There are three ways to uninstall jEdit and jEditLauncher.

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

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

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

G.6. The jEditLauncher Interface

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

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

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

Properties

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

Functions

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

G.7. Scripting Examples

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

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

```
' Example VBScript using jEditLauncher interface
dim launcher
set launcher = CreateObject("JEdit.JEditLauncher")
a = Array("I:\Source Code Files\shellext\jeditshell\*.h", _
 "I:\Source Code Files\shellext\jeditshell\*.cpp")
MsgBox "The server authorization code is " + _
 FormatNumber(launcher.ServerKey, 0, 0, 0, 0) + ".", _
```

```
 vbOKOnly + vbInformation, "jEditLauncher"
launcher.openFiles(a)
myScript = "jEdit.newFile(view); textArea.setSelectedText(" _
  & CHR(34) _
  & "Welcome to jEditLauncher." _
  & CHR(34) & ");"
launcher.evalScript(myScript)


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


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


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

G.8. jEditLauncher Logging

The jEditLauncher package has a logging facility that is separate from jEdit's Activity Log to provide a record of events occurring outside the Java virtual machine environment in which jEdit operates. The logging facility maintains two log files: `jelaunch.log` for events relating to starting jEdit, loading files and running scripts, and `install.log` for jEditLauncher installation activity. Both files are maintained in the directory in which jEdit is installed. They are cumulative from session to session, but may be manually deleted at any time without affecting program execution.

G.9. Legal Notice

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

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

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

II. Writing Edit Modes

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

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

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

Chapter 9. Mode Definition Syntax

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.

Editing a mode or a mode catalog file within jEdit will cause the changes to take effect immediately. If you edit modes using another application, the changes will take effect after the Utilities>Reload Edit Modes command is invoked.

9.1. An XML Primer

A very simple XML file (which also happens to be an edit mode) looks like so:

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

<!DOCTYPE MODE SYSTEM "xmode.dtd">

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

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

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

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

To insert a special character such as < or > literally in XML (for example, inside an attribute value), you must write it as an *entity*. An entity consists of the character's symbolic name enclosed with "&" and ";". The most frequently used entities are:

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

- `&` - The ampersand (&) character

For example, the following will cause a syntax error:

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

Instead, you must write:

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

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

9.2. The Preamble and MODE tag

Each mode definition must begin with the following:

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

Each mode definition must also contain exactly one `MODE` tag. All other tags (`PROPS`, `RULES`) must be placed inside the `MODE` tag. The `MODE` tag does not have any defined attributes. Here is an example:

```
<MODE>
    ... mode definition goes here ...
</MODE>
```

9.3. The PROPS Tag

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

All buffer-local properties listed in Section 6.2 may be given values in edit modes.

The following mode properties specify commenting strings:

- `commentEnd` - the comment end string, used by the Range Comment command.
- `commentStart` - the comment start string, used by the Range Comment command.
- `lineComment` - the line comment string, used by the Line Comment command.

When performing auto indent, a number of mode properties determine the resulting indent level:

- The line and the one before it are scanned for brackets listed in the `indentCloseBrackets` and `indentOpenBrackets` properties. Opening brackets in the previous line increase indent.

  If `lineUpClosingBracket` is set to `true`, then closing brackets on the current line will line up with the line containing the matching opening bracket. For example, in Java mode `lineUpClosingBracket` is set to `true`, resulting in brackets being indented like so:

  ```
  {
      // Code
      {
          // More code
      }
  }
  ```

  If `lineUpClosingBracket` is set to `false`, the line *after* a closing bracket will be lined up with the line containing the matching opening bracket. For example, in Lisp mode `lineUpClosingBracket` is set to `false`, resulting in brackets being indented like so:

  ```
  (foo 'a-parameter
      (crazy-p)
      (bar baz ()))
  (print "hello world")
  ```

- If the previous line contains no opening brackets, or if the `doubleBracketIndent` property is set to `true`, the previous line is checked against the regular expressions in the `indentNextLine` and `indentNextLines` properties. If the previous line matches the former, the indent of the current line is increased and the subsequent line is shifted back again. If the previous line matches the latter, the indent of the current and subsequent lines is increased.

  In Java mode, for example, the `indentNextLine` property is set to match control structures such as “if”, “else”, “while”, and so on.

  The `doubleBracketIndent` property, if set to the default of `false`, results in code indented like so:

  ```
  while(objects.hasNext())
  {
      Object next = objects.hasNext();
      if(next instanceof Paintable)
          next.paint(g);
  }
  ```

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

  ```
  while(objects.hasNext())
      {
          Object next = objects.hasNext();
  ```

```
            if(next instanceof Paintable)
                next.paint(g);
        }
```

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

```
<PROPS>
    <PROPERTY NAME="commentStart" VALUE="/*" />
    <PROPERTY NAME="commentEnd" VALUE="*/" />
    <PROPERTY NAME="lineComment" VALUE="//" />
    <PROPERTY NAME="wordBreakChars" VALUE=",+-=&lt;&gt;/?^&amp;*" />

    <!-- Auto indent -->
    <PROPERTY NAME="indentOpenBrackets" VALUE="{" />
    <PROPERTY NAME="indentCloseBrackets" VALUE="}" />
    <PROPERTY NAME="indentNextLine"
     VALUE="\s*(((if|while)\s*\(|else\s*|else\s+if\s*\(|for\s*\(.*\))[^{;]*)" />
    <!-- set this to 'true' if you want to use GNU coding style -->
    <PROPERTY NAME="doubleBracketIndent" VALUE="false" />
    <PROPERTY NAME="lineUpClosingBracket" VALUE="true" />
</PROPS>
```

9.4. The RULES Tag

`RULES` tags must be placed inside the `MODE` tag. Each `RULES` tag defines a *ruleset*. A ruleset consists of a number of *parser rules*, with each parser rule specifying how to highlight a specific syntax token. There must be at least one ruleset in each edit mode. There can also be more than one, with different rulesets being used to highlight different parts of a buffer (for example, in HTML mode, one rule set highlights HTML tags, and another highlights inline JavaScript). For information about using more than one ruleset, see Section 9.6.

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.
- `IGNORE_CASE` - if set to `FALSE`, matches will be case sensitive. Otherwise, case will not matter. Default is `TRUE`.
- `NO_WORD_SEP` - any non-alphanumeric character *not* in this list is treated as a word separator for the purposes of syntax highlighting.
- `DEFAULT` - the token type for text which doesn't match any specific rule. Default is `NULL`. See Section 9.15 for a list of token types.
- `HIGHLIGHT_DIGITS`

- `DIGIT_RE` - see below for information about these two attributes.

Here is an example `RULES` tag:

```
<RULES IGNORE_CASE="FALSE" HIGHLIGHT_DIGITS="TRUE">
    ... parser rules go here ...
</RULES>
```

9.4.1. Highlighting Numbers

If the `HIGHLIGHT_DIGITS` attribute is set to `TRUE`, jEdit will attempt to highlight numbers in this ruleset.

Any word consisting entirely of digits (0-9) will be highlighted with the `DIGIT` token type. A word that contains other letters in addition to digits will be highlighted with the `DIGIT` token type only if it matches the regular expression specified in the `DIGIT_RE` attribute. If this attribute is not specified, it will not be highlighted.

Here is an example `DIGIT_RE` regular expression that highlights Java-style numeric literals (normal numbers, hexadecimals prefixed with `0x`, numbers suffixed with various type indicators, and floating point literals containing an exponent):

```
DIGIT_RE="(0x[[:xdigit:]]+|[[:digit:]]+(e[[:digit:]]*)?)[lLdDfF]?"
```

Regular expression syntax is described in Appendix E.

9.4.2. Rule Ordering Requirements

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

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

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

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

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

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

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

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

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

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

9.4.3. Per-Ruleset Properties

The `PROPS` tag (described in Section 9.3) can also be placed inside the `RULES` tag to define ruleset-specific properties. The following properties can be set on a per-ruleset basis:

- `commentEnd` - the comment end string.
- `commentStart` - the comment start string.
- `lineComment` - the line comment string.

This allows different parts of a file to have different comment strings (in the case of HTML, for example, in HTML text and inline JavaScript). For information about the commenting commands, see Section 5.3.

9.5. The TERMINATE Tag

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

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

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

9.6. The SPAN Tag

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

- `TYPE` - The token type to highlight the span with. See Section 9.15 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.
- `AT_WHITESPACE_END` - If set to `TRUE`, the span will only be highlighted if the start sequence is the first non-whitespace text in the line.
- `AT_WORD_START` - If set to `TRUE`, the span will only be highlighted if the start sequence occurs at the beginning of a word.
- `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”.
- `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 not cross line breaks.
- `NO_WORD_BREAK` - If set to `TRUE`, the span will not cross word breaks.

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

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

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

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

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

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

9.7. The SPAN_REGEXP Tag

The SPAN_REGEXP rule is similar to the SPAN rule except the start sequence is taken to be a regular expression. In addition to the attributes supported by the SPAN tag, the HASH_CHAR attribute must be specified. It must be set to the first character that the regular expression matches. Note that this disallows regular expressions which can match more than one character at the start position.

Regular expression syntax is described in Appendix E.

Here is a SPAN_REGEXP rule that highlights constructs placed between <#ftl and >, as long as the <#ftl is followed by a word break:

```
<SPAN_REGEXP TYPE="KEYWORD1" HASH_CHAR="&lt;" DELEGATE="EXPRESSION">
    <BEGIN>&lt;#ftl\&gt;</BEGIN>
    <END>&gt;</END>
</SPAN_REGEXP>
```

9.8. The EOL_SPAN Tag

An EOL_SPAN is similar to a SPAN except that highlighting stops at the end of the line, and no end sequence needs to be specified. The text to match is specified between the opening and closing EOL_SPAN tags. The following attributes are supported:

- TYPE - The token type to highlight the span with. See Section 9.15 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.
- AT_WHITESPACE_END - If set to TRUE, the span will only be highlighted if the sequence is the first non-whitespace text in the line.
- AT_WORD_START - If set to TRUE, the span will only be highlighted if the start sequence occurs at the beginning of a word.
- 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”.

- EXCLUDE_MATCH - If set to TRUE, the start and end sequences will not be highlighted, only the text between them will.

Here is an EOL_SPAN that highlights C++ comments:

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

9.9. The EOL_SPAN_REGEXP Tag

The EOL_SPAN_REGEXP rule is similar to the EOL_SPAN rule except the match sequence is taken to be a regular expression. In addition to the attributes supported by the EOL_SPAN tag, the HASH_CHAR attribute must be specified. It must be set to the first character that the regular expression matches. Note that this disallows regular expressions which can match more than one character at the start position.

Regular expression syntax is described in Appendix E.

9.10. The MARK_PREVIOUS Tag

The MARK_PREVIOUS rule, which must be placed inside a RULES tag, highlights from the end of the previous syntax token to the matched text. The text to match is specified between opening and closing MARK_PREVIOUS tags. The following attributes are supported:

- TYPE - The token type to highlight the text with. See Section 9.15 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.
- AT_WHITESPACE_END - If set to TRUE, the sequence will only be highlighted if it is the first non-whitespace text in the line.
- AT_WORD_START - If set to TRUE, the sequence will only be highlighted if it occurs at the beginning of a word.
- 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_WHITESPACE_END="TRUE"
   EXCLUDE_MATCH="TRUE">:</MARK_PREVIOUS>
```

9.11. The MARK_FOLLOWING Tag

The `MARK_FOLLOWING` rule, which must be placed inside a `RULES` tag, highlights from the start of the match to the next syntax token. The text to match is specified between opening and closing `MARK_FOLLOWING` tags. The following attributes are supported:

- `TYPE` - The token type to highlight the text with. See Section 9.15 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.
- `AT_WHITESPACE_END` - If set to `TRUE`, the sequence will only be highlighted if it is the first non-whitespace text in the line.
- `AT_WORD_START` - If set to `TRUE`, the sequence will only be highlighted if it occurs at the beginning of a word.
- `EXCLUDE_MATCH` - If set to `TRUE`, the match will not be highlighted, only the text after it will.

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

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

9.12. The SEQ Tag

The `SEQ` rule, which must be placed inside a `RULES` tag, highlights fixed sequences of text. The text to highlight is specified between opening and closing `SEQ` tags. The following attributes are supported:

- `TYPE` - the token type to highlight the sequence with. See Section 9.15 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.
- `AT_WHITESPACE_END` - If set to `TRUE`, the sequence will only be highlighted if it is the first non-whitespace text in the line.
- `AT_WORD_START` - If set to `TRUE`, the sequence will only be highlighted if it occurs at the beginning of a word.
- `DELEGATE` - if this attribute is specified, all text after the sequence will be highlighted using this 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”.

The following rules highlight a few Java operators:

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

9.13. The SEQ_REGEXP Tag

The SEQ_REGEXP rule is similar to the SEQ rule except the match sequence is taken to be a regular expression. In addition to the attributes supported by the SEQ tag, the HASH_CHAR attribute must be specified. It must be set to the first character that the regular expression matches. Note that this disallows regular expressions which can match more than one character at the start position.

Here is an example of a SEQ_REGEXP rule that highlights Perl’s matcher constructions such as m/(.+):(\d+):(.+)/:

```
<SEQ_REGEXP TYPE="MARKUP"
    HASH_CHAR="m"
    AT_WORD_START="TRUE"
>m([[:punct:]])(?:.*?[^\\])*?\1[sgiexom]*</SEQ_REGEXP>
```

Regular expression syntax is described in Appendix E.

9.14. The KEYWORDS Tag

The KEYWORDS tag, which must be placed inside a RULES tag and can only appear once, specifies a list of keywords to highlight. Keywords are similar to SEQs, except that SEQs match anywhere in the text, whereas keywords only match whole words. Words are considered to be runs of text separated by non-alphanumeric characters.

The KEYWORDS tag does not define any attributes.

Each child element of the KEYWORDS tag is an element whose name is a token type, and whose content is the keyword to highlight. For example, the following rule highlights the most common Java keywords:

```
<KEYWORDS>
  <KEYWORD1>if</KEYWORD1>
  <KEYWORD1>else</KEYWORD1>
  <KEYWORD3>int</KEYWORD3>
  <KEYWORD3>void</KEYWORD3>
```

```
</KEYWORDS>
```

9.15. 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`
- `KEYWORD1`
- `KEYWORD2`
- `KEYWORD3`
- `LABEL`
- `LITERAL1`
- `LITERAL2`
- `MARKUP`
- `OPERATOR`

Chapter 10. Installing Edit Modes

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

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

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

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

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

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

The `MODE` tag supports the following attributes:

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

Glob pattern syntax is described in Appendix D.

> **Tip:** If an edit mode in the user-specific catalog has the same name as an edit mode in the system catalog, the version in the user-specific catalog will override the system default.

Chapter 11. Updating Edit Modes for jEdit 4.1

In jEdit 4.1, the mode file grammar has been cleaned up somewhat. As a result, some edit modes written for jEdit 4.0 and earlier need to be updated:

- Defining `<WHITESPACE>` rules is no longer necessary and doing so will print warnings to the activity logs.
- The `<KEYWORDS>` tag no longer accepts an `IGNORE_CASE` attribute. Set the `IGNORE_CASE` attribute of the `<RULES>` tag instead.
- The `<END>` tag of the `<SPAN>` rule used to be optional, in which case any occurrence of the start string would cause the remainder of the buffer to be highlighted with the span. In jEdit 4.1, the `<END>` tag can no longer be omitted, however a `<SEQ>` tag with a `DELEGATE` attribute can be used to achieve the same effect as endless span.
- Defining `<SEQ TYPE="NULL">` rules for word separators is no longer necessary. Now, any non-alphanumeric character not appearing in a keyword definition or the ruleset's `NO_WORD_SEP` attribute is considered a word separator.

III. Writing Macros

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

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

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

Chapter 12. Macro Basics

12.1. Introducing BeanShell

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

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

You do not have to know anything about Java to begin writing your own jEdit macros. But if you know how to program in Java, you already know how to write BeanShell scripts. The major strength of using BeanShell with a program written in Java is that it allows the user to customize the program's behavior using the same interfaces designed and used by the program itself. BeanShell can turn a well-designed application into a powerful, extensible toolkit.

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

12.2. Single Execution Macros

As noted in Section 7.3, you can save a BeanShell script of any length as a text file with the `.bsh` extension and run it from the Macros menu. There are three other ways jEdit lets you use BeanShell quickly, without saving a script to storage, on a "one time only" basis. You will find them in the Utilities menu.

Utilities>BeanShell>Evaluate BeanShell Expression displays 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.

Utilities>BeanShell>Evaluate For Selected Lines displays a text input dialog that asks you to type a single line of BeanShell commands. The commands are evaluated for each line of the selection. In addition to the standard set of variables described in Section 12.4, this command defines the following:

- `line` - the line number, from the start of the buffer. The first line is numbered 0.
- `index` - the line number, from the start of the selection. The first line is numbered 0.

- `text` - the text of the line.

Try typing an expression like **`(line + 1) + ": " + text`** in the Evaluate For Selected Lines dialog box. This will add a line number to each selected line beginning with the number **`1`**.

The BeanShell expression you enter will be evaluated and substituted in place of the entire text of a selected line. If you want to leave the line's current text as an element of the modified line, you must include the defined variable **`text`** as part of the BeanShell expression that you enter.

Utilities>BeanShell>Evaluate Selection evaluates the selected text as a BeanShell script and replaces it with the return value of the 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>BeanShell>Evaluate Selection. The selected text will be replaced by the answer, **`3205794`**.

Console plugin

You can also do the same thing using the BeanShell interpreter option of the Console plugin.

12.3. The Mandatory First Example

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

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

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

- An *object* is a collection of data that can be initialized, accessed and manipulated in certain defined ways.
- A *class* is a specification of what data an object contains and what methods can be used to work with the data. A Java application consists of one or more classes (in the case of jEdit ,over 600 classes) written by the programmer that defines the application's behavior. A BeanShell macro uses these classes, along with built-in classes that are supplied with the Java platform, to define its own behavior.
- A *subclass* (or child class) is a class which uses (or "inherits") the data and methods of its parent class along with additions or modifications that alter the subclass's behavior. Classes are typically organized in hierarchies of parent and child classes to organize program code, to define common behavior in shared parent class code, and to specify the types of similar behavior that child classes will perform in their own specific ways.
- A *method* (or function) is a procedure that works with data in a particular object, other data (including other objects) supplied as *parameters*, or both. Methods typically are applied to a particular object which is an *instance* of the class to which the method belongs.
- A *static method* differs from other methods in that it does not deal with the data in a particular object but is included within a class for the sake of convenience.

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

In our one line script, the static method `Macros.message()` has two parameters because that is the way the method is defined in the `Macros` class. You must specify both parameters when you call the function. The first parameter, *`view`*, is a a variable naming the current, active `View` object. Information about pre-defined variables can be found in Section 12.4.

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

If you are a Java programmer, you might wonder about a few things missing from this one line program. There is no class definition, for example. You can think of a BeanShell script as an implicit definition of a `main()` method in an anonymous class. That is in fact

how BeanShell is implemented; the class is derived from a BeanShell class called `XThis`. If you don't find that helpful, just think of a script as one or more blocks of procedural statements conforming to Java syntax rules. You will also get along fine (for the most part) with C or C++ syntax if you leave out anything to do with pointers or memory management - Java and BeanShell do not have pointers and deal with memory management automatically.

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

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

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

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

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

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

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

12.4. Predefined Variables in BeanShell

The following variables are always available for use in BeanShell scripts:

- `buffer` - a `Buffer` object represents the contents of an open text file. The variable `buffer` is predefined as the current, visible buffer being edited.
- `view` - A `View` represents a top-level window, extending Java's `JFrame` class, that contains the various visible components of the program, including the text area, menu bar, toolbar, and any docked windows. The variable `view` is defined as the current, active `View` object.

  This variable has the same value as calling:

  ```
  jEdit.getActiveView()
  ```

- `editPane` - an `EditPane` object contains a text area and buffer switcher. A view can be split to display multiple buffers, each in its own edit pane. Among other things, the `EditPane` class contains methods for selecting the buffer to edit.

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

This variable has the same value as calling:

```
view.getEditPane()
```

- `textArea` - 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.

  This variable has the same value as calling:

  ```
  editPane.getTextArea()
  ```

- `scriptPath` - set to the full path of the script currently being executed.

Note that these variables are set at the beginning of macro execution. If the macro switches views, buffers or edit panes, the variables will be out of date. In that case, you can use the method calls equivalent to the values of the variables.

12.5. Helpful Methods in the Macros Class

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

- `public static void` **`message`**`(Component` *`comp`*`, String` *`message`*`);`
- `public static void` **`error`**`(Component` *`comp`*`, String` *`message`*`);`
- `public static String` **`input`**`(Component` *`comp`*`, String` *`prompt`*`);`
- `public static String` **`input`**`(Component` *`comp`*`, String` *`prompt`*`, String` *`defaultValue`*`);`
- `public static int` **`confirm`**`(Component` *`comp`*`, String` *`prompt`*`, int` *`buttons`*`);`

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

word is the type of the value returned by the method. The keyword `void` is Java's way of saying the the method does not have a return value.

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

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

The `confirm()` method in the `Macros` class is a little more complex. The `buttons` parameter has an `int` type, and the usual way to supply a value is to use one of the predefined values taken from Java's `JOptionPane` class. You can choose among `JOptionPane.YES_NO_OPTION`, `JOptionPane.YES_NO_CANCEL_OPTION`, or `JOptionPane.OK_CANCEL_OPTION`. The return value of the method is also an `int`, and should be tested against the value of other predefined constants: `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.OK_OPTION` or `JOptionPane.CANCEL_OPTION`.

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, "When were you born?",
    "I don't remember, I was very young at the time");

int result = Macros.confirm("Do you really want to learn"
    + " about BeanShell?",JOptionPane.YES_NO_OPTION);
```

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

12.6. BeanShell Dynamic Typing

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

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

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

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

However, if you first declared `result` to be type `String` and and then tried these reassignments, BeanShell would complain. While avoiding explicit type declaration makes writing macro code simpler, using them can act as a check to make sure you are not using the wrong variable type of object at a later point in your script. It also makes it easier (if you are so inclined) to take a BeanShell “prototype” and incorporate it in a Java program.

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

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

12.7. 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);
```

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

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

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

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

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

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

// a line count for the current buffer
String newText = "This file contains "
    + textArea.getLineCount() + " lines.";
```

Here are brief comments on each:

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

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

  Taking the pieces in order:

  - A Java `Date` object is created using the `new` keyword. The empty parenthesis after `Date` signify a call on the *constructor method* of `Date` having no parameters; here, a `Date` is created representing the current date and time.
  - `DateFormat.getDateInstance()` is a static method that creates and returns a `DateFormat` object. As the name implies, `DateFormat` is a Java class that takes `Date` objects and produces readable text. The method `getDateInstance()` returns a `DateFormat` object that parses and formats dates. It will use the default *locale* or text format specified in the user's Java installation.
  - Finally, `DateFormat.format()` is called on the new `DateFormat` object using the `Date` object as a parameter. The result is a `String` containing the date in the default locale.

    - Note that the `Date` class is contained in the `java.util` package, so an explicit import statement is not required. However, `DateFormat` is part of the `java.text` package, which is not automatically imported, so an explicit `import` statement must be used.

- The third example shows three items of note:
    - `getLineCount()` is a method in jEdit's `JEditTextArea` class. It returns an `int` representing the number of lines in the current text buffer. We call it on `textArea`, the pre-defined, current `JEditTextArea` object.
    - The use of the `+` operator (which can be chained, as here) appends objects and string literals to return a single, concatenated `String`.

Chapter 13. A Dialog-Based Macro

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

13.1. Use of the Macro

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

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

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

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

13.2. Listing of the Macro

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

```
// beginning of Add_Prefix_and_Suffix.bsh
```

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

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

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

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

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

    // locate the dialog in the center of the
    // editing pane and make it visible (see Section 13.3.6)
```

```
    dialog.pack();
    dialog.setLocationRelativeTo(view);
    dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    dialog.setVisible(true);

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

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

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

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

// end of Add_Prefix_and_Suffix.bsh
```

13.3. Analysis of the Macro

13.3.1. Import Statements

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

This macro makes use of classes in the `javax.swing.border` package, which is not automatically imported. As we mentioned previously (see Section 12.3), jEdit's implementation of BeanShell causes a number of classes to be automatically imported. Classes that are not automatically imported must be identified by a full qualified name or be the subject of an `import` statement.

13.3.2. Create the Dialog

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

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

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

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

components inside. We then attach the `JPanel` as the dialog's content pane, replacing the dialog's home-grown version.

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

13.3.3. Create the Text Fields

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

Next we shall create a smaller panel containing two fields for entering the prefix and suffix text and two labels 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. When the component has input focus, the up and down keys scroll through the prior values for the variable.

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

The labels that accompany the text fields are `JLabel` objects from the Java Swing package. The constructor we use for both labels 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` container object named `fieldPanel` that will be nested inside the dialog's content pane that we have already created. In the constructor for `fieldPanel`, we assign a new `GridLayout` with the indicated parameters: four rows, one column, zero spacing between columns (a meaningless element of a grid with only one column, but nevertheless a required parameter) and spacing of six pixels between

rows. The spacing between rows spreads out the four “grid” elements. After the components, the panel and the layout are specified, the components are added to `fieldPanel` top to bottom, one “grid cell” at a time. Finally, the complete `fieldPanel` is added to the dialog’s content pane to occupy the “Center” section of the content pane.

13.3.4. Create the Buttons

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

To create the dialog’s buttons, we follow repeat the “nested container” pattern we used in creating the text fields. First, we create a new, nested panel. This time we use a `BoxLayout` that places components either in a single row or column, depending on the parameter passed to its constructor. This layout object is more flexible than a `GridLayout` in that variable spacing between elements can be specified easily. 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 when the dialog if first displayed. Finally, we place the buttons side by side with a 6 pixel gap between them (for aesthetic reasons), and place the completed `buttonPanel` in the “South” section of the dialog’s content pane.

13.3.5. Register the Action Listeners

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

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

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

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

13.3.6. Make the Dialog Visible

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

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

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

13.3.7. The Action Listener

```
// this method will be called when a button is clicked
// or when ENTER is pressed
void actionPerformed(e)
```

```
{
    if(e.getSource() != cancel)
    {
        processText();
    }
    dialog.dispose();
}
```

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

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

13.3.8. Get the User's Input

```
// this is where the work gets done to insert
// the prefix and suffix
void processText()
{
    prefix = prefixField.getText();
    suffix = suffixField.getText();
    if(prefix.length() == 0 && suffix.length() == 0)
        return;
    prefixField.addCurrentToHistory();
    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()`. We do not need to test the `prefixField` or `suffixField` controls for `null` or empty values because `addCurrentToHistory()` does that internally.

13.3.9. Call jEdit Methods to Manipulate Text

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

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

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

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

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

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

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

Compound edits

Note the `beginCompoundEdit()` and `endCompoundEdit()` calls. These ensure that all edits performed between the two calls can be undone in one step. Normally, jEdit automatically wraps a macro call in these methods; however if the macro shows a non-modal dialog box, as far as jEdit is concerned the macro has finished executing by the time the dialog is shown, since control returns to the event dispatch thread.

If you do not understand this, don't worry; just keep it in mind if your macro needs to show a non-modal dialog box for some reason; Most macros won't.

13.3.10. The Main Routine

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

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

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

Chapter 14. Macro Tips and Techniques

14.1. Getting Input for a Macro

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

14.1.1. Getting a Single Line of Text

As mentioned earlier in Section 12.5, the method `Macros.input()` offers a convenient way to obtain a single line of text input. Here is an example that inserts a pair of HTML markup tags specified by the user.

```
// Insert_Tag.bsh

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

insertTag();

// end Insert_Tag.bsh
```

Here the call to `Macros.input()` seeks the name of the markup tag. This method sets the message box title to a fixed string, “Macro input”, but the specific message Enter name of tag provides all the information necessary. The return value `tag` must be tested

to see if it is null. This would occur if the user presses the Cancel button or closes the dialog window displayed by `Macros.input()`.

14.1.2. Getting Multiple Data Items

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

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

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

```
// excerpt from Write_File_Header.bsh

title = “Write file header”;

currentName = buffer.getName();

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

namePanel = new JPanel(new GridLayout(1, 2));
nameLabel = new JLabel(“Name of file:”, SwingConstants.LEFT);
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 uses a total of seven visible components. Two of them are created behind the scenes by `showConfirmDialog()`, the rest are made by the macro. To arrange them, the script creates an array of `Object` objects and assigns components to each location in the array. This translates to a fixed, top-to-bottom arrangement in the message box created by `showConfirmDialog()`.
- The macro uses `JTextField` objects to obtain most of the input data. The fields `nameField` and `authorField` are created with constructors that take the initial, default text to be displayed in the field as a parameter. When the message box is displayed, the default text will appear and can be altered or deleted by the user.
- The text field `descField` uses an empty string for its initial value. The second parameter in its constructor sets the width of the text field component, expressed as the number of characters of “average” width. When `showConfirmDialog()` prepares the layout of the message box, it sets the width wide enough to accommodate the designated with of `descField`. This technique produces a message box and input text fields that are wide enough for your data with one line of code.
- The displayed message box includes a `JCheckBox` component that determines whether the buffer will be saved to disk immediately after the file header is written. To conserve space in the message box, we want to display the check box to the right of the label Name of file:. To do that, we create a `JPanel` object and populate it with the label and the checkbox in a left-to-right `GridLayout`. The `JPanel` containing the two components is then added to the beginning of `message` array.
- The two visible components created by `showConfirmDialog()` appear at positions 3 and 6 of the `message` array. Only the text is required; they are rendered as text labels.
- There are three invisible components created by `showConfirmDialog()`. Each of them involves a call to `Box.createVerticalStrut()`. The `Box` class is a sophisticated layout class that gives the user great flexibility in sizing and

positioning components. Here we use a `static` method of the `Box` class that produces a vertical *strut*. This is a transparent component whose width expands to fill its parent component (in this case, the message box). The single parameter indicates the height of the strut in pixels. The last call to `createVerticalStrut()` separates the description text field from the OK and Cancel buttons that are automatically added by `showConfirmDialog()`.

- Finally, the call to `showConfirmDialog()` uses defined constants for the option type and the message type. The constants are the same as those used with the `Macros.confirm()` method; see Section 12.5. The option type signifies the use of OK and Cancel buttons. The `QUERY_MESSAGE` message type causes the message box to display a question mark icon.

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

14.1.3. Selecting Input From a List

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

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

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

The following macro fragment illustrates the use of this method.

```
// fragment illustrating use of showInputDialog()
options = new Object[5];
options[0] = "JLabel";
options[1] = "JTextField";
options[2] = "JCheckBox";
options[3] = "HistoryTextField";
options[4} = "-- other --";

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

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

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

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

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

abbrevSets = getActiveSets();

...

// from showAbbrevs() method

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

// end fragments
```

14.1.4. Using a Single Keypress as Input

Some macros may choose to emulate the style of character-based text editors such as emacs or vi. They will require only a single keypress as input that would be handled by

the macro but not displayed on the screen. If the keypress corresponds to a character value, jEdit can pass that value as a parameter to a BeanShell script.

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

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

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

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

```
// Next_Char.bsh

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

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

// end Next_Char.bsh
```

Once again, here are a few comments on the macro's design.

- A `StringBuffer` object is used for efficiency; it obviates multiple creation of fixed-length `String` objects. The parameter to the constructor of `script` specifies the initial size of the buffer that will receive the contents of the child script.

- Besides the quoting of the script code, the formatting of the macro is entirely optional but (hopefully) makes it easier to read.
- It is important that the child script be self-contained. It does not run in the same namespace as the "parent" macro `Next_Char.bsh` and therefore does not share variables, methods, or scripted objects defined in the parent macro.
- Finally, access to the `InputHandler` object used by jEdit is available by calling `getInputHandler()` on the current view.

14.2. Startup Scripts

On startup, jEdit runs any BeanShell scripts located in the `startup` subdirectory of the jEdit installation and user settings directories (see Section 6.4). As with macros, the scripts must have a `.bsh` file name extension. Startup scripts are run near the end of the startup sequence, after plugins, properties and such have been initialized, but before the first view is opened.

Startup scripts can perform initialization tasks that cannot be handled by command line options or ordinary configuration options, such as customizing jEdit's user interface by changing entries in the Java platform's `UIManager` class.

Startup scripts have an additional feature lacking in ordinary macros that can help you further customize jEdit. Variables and methods defined in a startup script are available in all instances of the BeanShell interpreter created in jEdit. This allows you to create a personal library of methods and objects that can be accessed at any time during the editing session in another macro, the BeanShell shell of the Console plugin, or menu items such as Utilities>BeanShell>Evaluate BeanShell Expression.

The startup script routine will run script files in the installation directory first, followed by scripts in the user settings directory. In each case, scripts will be executed in alphabetical order, applied without regard to whether the file name contains upper or lower case characters.

If a startup script throws an exception (because, for example, it attempts to call a method on a `null` object). jEdit will show an error dialog box and move on to the next startup script. If script bugs are causing jEdit to crash or hang on startup, you can use the **`-nostartupscripts`** command line option to disable them for that editing session.

Another important difference between startup scripts and ordinary macros is that startup scripts cannot use the pre-defined variables `view`, `textArea`, `editPane` and `buffer`. This is because they are executed before the initial view is created.

If you are writing a method in a startup script and wish to use one of the above variables, pass parameters of the appropriate type to the method, so that a macro calling them after

startup can supply the appropriate values. For example, a startup script could include a method

```
void doSomethingWithView(View v, String s)  {
    ...
}
```

so that during the editing session another macro can call the method using

```
doSomethingWithView(view, "something");
```

Reloading startup scripts without restarting

It is actually possible to reload startup scripts or load other scripts without restarting jEdit, using a BeanShell statement like the following:

```
BeanShell.runScript(view,path,null,false);
```

For *`path`*, you can substitute any string, or a method call such as `buffer.getPath()`.

14.3. Running Scripts from the Command Line

The **`-run`** command line switch specifies a BeanShell script to run on startup:

```
$ jedit -run=test.bsh
```

Note that just like with startup scripts, the `view`, `textArea`, `editPane` and `buffer` variables are not defined.

If another instance is already running, the script will be run in that instance, and you will be able to use the `jEdit.getLastView()` method to obtain a view. However, if a new instance of jEdit is being started, the script will be run at the same time as all other startup scripts; that is, before the first view is opened.

If your script needs a view instance to operate on, you can use the following code pattern to obtain one, no matter how or when the script is being run:

```
void doSomethingUseful()
{
    void run()
    {
        view = jEdit.getLastView();

        // put actual script body here
```

```
        }

        if(jEdit.getLastView() == null)
            VFSManager.runInAWTThread(this);
        else
            run();
    }

    doSomethingUseful();
```

If the script is being run in a loaded instance, it can be invoked to perform its work immediately. However, if the script is running at startup, before an initial view exists, its operation must be delayed to allow the view object first to be created and displayed. In order to queue the macro's operation, the scripted "closure" named `doSomethingUseful()` implements the `Runnable` interface of the Java platform. That interface contains only a single `run()` method that takes no parameters and has no return value. The macro's implementation of the `run()` method contains the "working" portion of the macro. Then the scripted object, represented by a reference to `this`, is passed to the `runInAWTThread()` method. This schedules the macro's operations for execution after the startup routine is complete.

As this example illustrates, the `runInAWTThread()` method can be used to ensure that a macro will perform operations after other operations have completed. If it is invoked during startup, it schedules the specified `Runnable` object to run after startup is complete. If invoked when jEdit is fully loaded, the `Runnable` object will execute after all pending input/output is complete, or immediately if there are no pending I/O operations. This will delay operations on a new buffer, for example, until after the buffer is loaded and displayed.

14.4. Advanced BeanShell Techniques

BeanShell has a few advanced features that we haven't mentioned yet. They will be discussed in this section.

14.4.1. BeanShell's Convenience Syntax

We noted earlier that BeanShell syntax does not require that variables be declared or defined with their type, and that variables that are not typed when first used can have values of differing types assigned to them. In addition to this "loose" syntax, BeanShell allows a "convenience" syntax for dealing with the properties of JavaBeans. They may be accessed or set as if they were data members. They may also be accessed using the name of the property enclosed in quotation marks and curly brackets. For example, the following statement are all equivalent, assuming `btn` is a `JButton` instance:

```
b.setText("Choose");
b.text = "Choose";
b{"text"} = "Choose";
```

The last form can also be used to access a key-value pair of a `Hashtable` object.

14.4.2. Special BeanShell Keywords

BeanShell uses special keywords to refer to variables or methods defined in the current or an enclosing block's scope:

- The keyword `this` refers to the current scope.
- The keyword `super` refers to the immediately enclosing scope.
- The keyword `global` refers to the top-level scope of the macro script.

The following script illustrates the use of these keywords:

```
a = "top\n";
foo() {
    a = "middle\n";
    bar() {
        a = "bottom\n";
        textArea.setSelectedText(global.a);
        textArea.setSelectedText(super.a);
        // equivalent to textArea.setSelectedText(this.a):
        textArea.setSelectedText(a);
    }

    bar();
}
foo();
```

When the script is run, the following text is inserted in the current buffer:

```
top
middle
bottom
```

14.4.3. Implementing Interfaces

As discussed in the macro example in Chapter 13, scripted objects can implement Java interfaces such as `ActionListener`. Which interfaces may be implemented varies depending upon the version of the Java runtime environment being used. If running under Java 1.1 or 1.2, BeanShell objects can only implement the AWT or Swing event listener interfaces contained in the `java.awt.event` and `javax.swing.event`

packages, along with the `java.lang.Runnable` interface. If BeanShell is running under Java 1.3 or 1.4, which jEdit 4.0 requires, any interface can be implemented.

Frequently it will not be necessary to implement all of the methods of a particular interface in order to specify the behavior of a scripted object. Under Java 1.3 and above, the virtual machine's reflection mechanism will throw an exception for any missing interface methods. This will bring macro execution to a halt unless the exception is trapped and handled. The solution is to implement the `invoke()` method, which is called when an undefined method is invoked on a scripted object. Typically, the implementation of this method will do nothing, as in the following example:

```
invoke(method, args) {}
```

14.5. Debugging Macros

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

14.5.1. Identifying Exceptions

An *exception* is a condition reflecting an error or other unusual result of program execution that requires interruption of normal program flow and some kind of special handling. Java has a rich (and extensible) collection of exception classes which represent such conditions.

jEdit catches exceptions thrown by BeanShell scripts and displays them in a dialog box. In addition, the full traceback is written to the activity log (see Appendix B for more information about the activity log).

There are two broad categories of errors that will result in exceptions:

- *Interpreter errors*, which may arise from typing mistakes like mismatched brackets or missing semicolons, or from BeanShell's failure to find a class corresponding to a particular variable.

  Interpreter errors are usually accompanied by the line number in the script, along with the cause of the error.

- *Execution errors*, which result from runtime exceptions thrown by the Java platform when macro code is executed.

  Some exceptions thrown by the Java platform can often seem cryptic. Nevertheless, examining the contents of the activity log may reveals clues as to the cause of the error.

14.5.2. Using the Activity Log as a Tracing Tool

Sometimes exception tracebacks will say what kind of error occurred but not where it arose in the script. In those cases, you can insert calls that log messages to the activity log in your macro. If the logged messages appear when the macro is run, it means that up to that point the macro is fine; but if an exception is logged first, it means the logging call is located after the cause of the error.

To write a message to the activity log, use the following method of the `Log` class:

- `public static void` **`log`**`(int` *`urgency`*`, Object` *`source`*`, Object` *`message`*`);`

The parameter `urgency` can take one of the following constant values:

- `Log.DEBUG`
- `Log.MESSAGE`
- `Log.NOTICE`
- `Log.WARNING`
- `Log.ERROR`

Note that the `urgency` parameter merely changes the string prefixed to the log message; it does not change the logging behavior in any other way.

The parameter `source` can be either an object or a class instance. When writing log messages from macros, set this parameter to `BeanShell.class` to make macro errors easier to spot in the activity log.

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

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

The corresponding activity log entry might read as follows:

```
[debug] BeanShell: counter = 15
```

Using message dialog boxes as a tracing tool

If you would prefer not having to deal with the activity log, you can use the `Macros.message()` method as a tracing tool. Just insert calls like the following in the macro code:

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

Execution of the macro is halted until the message dialog box is closed. When you have finished debugging the macro, you should delete or comment out the debugging calls to `Macros.message()` in your final source code.

Chapter 15. BeanShell Commands

BeanShell includes a set of *commands*; subroutines that can be called from any script or macro. The following is a summary of those commands which may be useful within jEdit.

> **Note:** Plugins, because they are written in Java and not BeanShell, cannot make use of BeanShell commands.

15.1. Output Commands

- `void` **`cat`**`(String` *`filename`*`);`

  Writes the contents of *`filename`* to the activity log.

- `void` **`javap`**`(String | Object | Class` *`target`*`);`

  Writes the public fields and methods of the specified class to the output stream of the current process. Requires Java 2 version 1.3 or greater.

- `void` **`print`**`(`*`arg`*`);`

  Writes the string value of the argument to the activity log, or if run from the Console plugin, to the current output window. If *`arg`* is an array, `print` runs itself recursively on the array's elements.

15.2. File Management Commands

- `void` **`cd`**`(String` *`dirname`*`);`

  Changes the working directory of the BeanShell interpreter to *`dirname`*.

- `void` **`dir`**`(String` *`dirname`*`);`

  Displays the contents of directory *`dirname`*. The format of the display is similar to the Unix `ls -l` command.

- `void` **`mv`**`(String` *`fromFile`*`, String` *`toFile`*`);`

  Moves the file named by *`fromFile`* to *`toFile`*.

- `File` **`pathToFile`**`(String` *`filename`*`);`

  Create a `File` object corresponding to `filename`. Relative paths are resolved with reference to the BeanShell interpreter's working directory.

- `void` **`pwd`**`();`

  Writes the current working directory of the BeanShell interpreter to the output stream of the current process.

- `void` **`rm`**`(String` *`pathname`*`);`

  Deletes the file name by *`pathname`*.

15.3. Component Commands

- `JFrame` **`frame`**`(Component` *`frame`*`);`

  Displays the component in a top-level `JFrame`, centered and packed. Returns the `JFrame` object.

- `Object` **`load`**`(String` *`filename`*`);`

  Loads and returns a serialized Java object from *`filename`*.

- `void` **`save`**`(Component` *`component`*`, String` *`filename`*`);`

  Saves *`component`* in serialized form to *`filename`*.

- `Font` **`setFont`**`(Component` *`comp`*`, int` *`ptsize`*`);`

  Set the font size of *`component`* to *`ptsize`* and returns the new font.

15.4. Resource Management Commands

- `URL` **`getResource`**`(String` *`path`*`);`

  Returns the resource specified by *`path`*. A absolute path must be used to return any resource available in the current classpath.

15.5. Script Execution Commands

- `Thread` **`bg`**`(String` *`filename`*`);`

  Run the BeanShell script named by *`filename`* in a copy of the existing namespace and in a separate thread. Returns the `Thread` object so created.

- `void` **`exec`**`(String` *`cmdline`*`);`

  Start the external process by calling `Runtime.exec()` on *`cmdline`*. Any output is directed to the output stream of the calling process.

- `Object` **`eval`**`(String` *`expression`*`);`

  Evaluates the string *`expression`* as a BeanShell script in the interpreter's current namespace. Returns the result of the evaluation of `null`.

- `bsh.This` **`run`**`(String` *`filename`*`);`

  Run the BeanShell script named by *`filename`* in a copy of the existing namespace. The return value represent the object context of the script, allowing you to access its variables and methods.

- `void` **`source`**`(String` *`filename`*`);`

  Evaluates the contents of *`filename`* as a BeanShell script in the interpreter's current namespace.

15.6. BeanShell Object Management Commands

- `bind(bsh.This ths, bsh.Namespace namespace);`

  Binds the scripted object *`ths`* to *`namespace`*.

- `void clear();`

  Clear all variables, methods, and imports from this namespace. If this namespace is the root, it will be reset to the default imports.

- `bsh.This extend(bsh.This object);`

  Creates a new BeanShell `This` scripted object that is a child of the parameter *`object`*.

- `bsh.This object();`

  Creates a new BeanShell `This` scripted object which can hold data members. You can use this to create an object for storing miscellaneous crufties, like so:

  ```
  crufties = object();
  crufties.foo = "hello world";
  crufties.counter = 5;
  ...
  ```

- `setNameSpace(bsh.Namespace namespace);`

  Set the namespace of the current scope to *`namespace`*.

- `bsh.This super(String scopename);`

  Returns a reference to the BeanShell `This` object representing the enclosing method scope specified by *`scopename`*. This method work similar to the `super` keyword but can refer to enclosing scope at higher levels in a hierarchy of scopes.

- `void unset(String name);`

  Removes the variable named by *`name`* from the current interpreter namespace. This has the effect of "undefining" the variable.

15.7. Other Commands

- `void` **`debug`**`();`

  Toggles BeanShell's internal debug reporting to the output stream of the current process.

- **`getSourceFileInfo`**`();`

  Returns the name of the file or other source from which the BeanShell interpreter is reading.

IV. Writing Plugins

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

Like jEdit itself, plugins are written primarily in Java. While this guide assumes some working knowledge of the language, you are not required to be a Java wizard. If you can write a useful application of any size in Java, you can write a plugin.

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

Chapter 16. Introducing the Plugin API

The *jEdit Plugin API* provides a framework for hosting plugin applications without imposing any requirements on the design or function of the plugin itself. You could write a application that performs spell checking, displays a clock or plays chess and turn it into a jEdit plugin. There are currently over 50 released plugins for jEdit. While none of them play chess, they perform a wide variety of editing and file management tasks.

A detailed listing of available plugins is available at `plugins.jedit.org`. You can also find beta versions of new plugins in the "Downloads" area of `community.jedit.org`.

Using the "Plugin Manager" feature of jEdit, users with an Internet connection can check for new or updated plugins and install and remove them without leaving jEdit. See Chapter 8 for details.

Requirements for "plugging in" to jEdit are as follows:

- This plugin must supply information about itself, such as its name, version, author, and compatibility with versions of jEdit.
- The plugin must provide for activating, displaying and deactivating itself upon direction from jEdit, typically in response to user input.
- The plugin may define *actions*, both explicitly with an action definition file, or implicitly by providing dockable windows. Actions are small blocks of BeanShell code that jEdit will perform on behalf of the plugin upon user request. They provide the "glue" between user input and specific plugin routines.

  By convention, plugins display their available actions in submenus of jEdit's Plugins menu; each menu item corresponds to an action. The user can also assign actions to keyboard shortcuts, toolbar buttons or entries in the text area's right-click menu.

- The plugin may, but need not, provide a user interface.

  If the plugin has a visible interface, it can be shown in any object derived from one of Java top-level container classes: `JWindow`, `JDialog`, or `JFrame`. jEdit also provides a dockable window API, which allows plugin windows derived from the `JComponent` class to be docked into views or shown in top-level frames, at the user's request.

  Plugins can also act directly upon jEdit's text area. They can add graphical elements to the text display (like error highlighting in the case of the ErrorList plugin) or decorations surrounding the text area (like the JDiff plugin's summary views).

- Plugins may provide a range of options that the user can modify to alter their configuration.

  If a plugin provides configuration options in accordance with the plugin API, jEdit will make them available in the Global Options dialog box.

- While it is not required, plugins are encouraged to provide documentation.

As noted, many of these features are optional; it is possible to write a plugin that does not provide actions, configuration options, or dockable windows. The majority of plugins, however, provide most of these services.

Plugins and different jEdit versions

As jEdit continues to evolve and improve, elements of the plugin API may change with a new jEdit release.

On occasion an API change will break code used by plugins, although efforts are made to maintain or deprecate plugin-related code on a transitional basis. While the majority of plugins are unaffected by most changes and will continue working, it is a good idea to monitor the jEdit change log, the mailing lists and `community.jedit.org` for API changes so that you can update your plugin if necessary.

Chapter 17. Implementing a Simple Plugin

There are many applications for the leading operating systems that provide a "scratch-pad" or "sticky note" facility for the desktop display. A similar type of facility operating within the jEdit display would be a convenience. The use of dockable windows would allow the notepad to be displayed or hidden with a single mouse click or keypress (if a keyboard shortcut were defined). The contents of the notepad could be saved at program exit (or, if earlier, deactivation of the plugin) and retrieved at program startup or plugin activation.

We will keep the capabilities of this plugin modest, but a few other features would be worthwhile. The user should be able to write the contents of the notepad to storage on demand. It should also be possible to choose the name and location of the file that will be used to hold the notepad text. This would allow the user to load other files into the notepad display. The path of the notepad file should be displayed in the plugin window, but will give the user the option to hide the file name. Finally, there should be an action by which a single click or keypress would cause the contents of the notepad to be written to the new text buffer for further processing.

The full source code for QuickNotepad is contained in jEdit's source code distribution. We will provide excerpts in this discussion where it is helpful to illustrate specific points. You are invited to obtain the source code for further study or to use as a starting point for your own plugin.

17.1. How Plugins are Loaded

We will discuss the implementation of the QuickNotepad plugin, along with the jEdit APIs it makes use of. But first, we describe how plugins are loaded.

As part of its startup routine, jEdit's `main` method calls various methods to load and initialize plugins.

Plugin loading occurs after jEdit has loaded application properties, any user-supplied properties, and the application's set of actions that will be available from jEdit's menu bar (as well as the toolbar and keyboard shortcuts).

Plugin loading occurs before jEdit opens the initial view or loads any files for editing. It also occurs before jEdit runs any startup scripts.

Plugins are loaded from files with the `.jar` filename extension located in the `jars` subdirectories of the jEdit installation and user settings directories (see Section 6.4).

For each JAR archive file it finds, jEdit scans its entries and performs the following tasks:

- Adds to a collection maintained by jEdit a new object of type `EditPlugin.JAR`. This is a data structure holding the name of the JAR archive file, a reference to the `JARClassLoader`, and a collection of plugins found in the archive file.
- Loads any properties defined in files ending with the extension `.props` that are contained in the archive. See Section 17.4.
- Reads action definitions from any file named `actions.xml` in the archive (the file need not be at the top level). See Section 17.5.
- Parses and loads the contents of any file named `dockables.xml` in the archive (the file need not be at the top level). This file contains BeanShell code for creating docking or floating windows that will contain the visible components of the plugin. Not all plugins define dockable windows, but those that do need a `dockables.xml` file. See Section 17.6.
- Checks for a class name with a name ending with `Plugin.class`.

  Such a class is known as a *plugin core class* and must extend jEdit's abstract `EditPlugin` class. The initialization routine checks the plugin's properties to see if it is subject to any dependencies. For example, a plugin may require that the version of the Java runtime environment or of jEdit itself be equal to or above some threshold version. A plugin can also require the presence of another plugin.

  If any dependency is not satisfied, the loader marks the plugin as "broken" and logs an error message.

After scanning the plugin JAR file and loading any resources, a new instance of the plugin core class is created and added to the collection maintained by the appropriate `EditPlugin.JAR`. jEdit then calls the `start()` method of the plugin core class. The `start()` method can perform initialization of the object's data members. Because this method is defined as an empty "no-op" in the `EditPlugin` abstract class, a plugin need not provide an implementation if no unique initialization is required.

17.2. The QuickNotepadPlugin Class

The major issues encountered when writing a plugin core class arise from the developer's decisions on what features the plugin will make available. These issues have implications for other plugin elements as well.

- Will the plugin provide for actions that the user can trigger using jEdit's menu items, toolbar buttons and keyboard shortcuts?
- Will the plugin have its own visible interface?
- Will the plugin have settings that the user can configure?

- Will the plugin respond to any messages reflecting changes in the host application's state?

Recall that the plugin core class must extend `EditPlugin`. In QuickNotepad's plugin core class, there are no special initialization or shutdown chores to perform, so we will not need a `start()` or `stop()` method.

The resulting plugin core class is lightweight and straightforward to implement:

-
```
public class QuickNotepadPlugin extends EditPlugin {
    public static final String NAME = "quicknotepad";
    public static final String MENU = "quicknotepad.menu";
    public static final String PROPERTY_PREFIX
        = "plugin.QuickNotepadPlugin.";
    public static final String OPTION_PREFIX
        = "options.quicknotepad.";
```

  First we define a few static `String` data members to enforce consistent syntax for the name of properties we will use throughout the plugin.

-
```
    public void createMenuItems(Vector menuItems) {
        menuItems.addElement(GUIUtilities.loadMenu(MENU));
    }
```

  This implementation of the `EditPlugin.createMenuItems()` method is very typical. It uses a jEdit utility function to create the menu, taking the list of actions from the `quicknotepad` property, and the label from `quotenotepad.label`.

  If the plugin only had a single menu item (for example, an item activating a dockable window), we would call `GUIUtilities.loadMenuItem()` instead of `GUIUtilities.loadMenu()`.

-
```
    public void createOptionPanes(OptionsDialog od) {
        od.addOptionPane(new QuickNotepadOptionPane());
    }


}
```

This implementation of the `EditPlugin.createOptionPanes()` method adds a new instance of `QuickNotepadOptionPane` to the given instance of the Global Options dialog box.

17.3. The EditBus

Plugins register `EBComponent` instances with the `EditBus` to receive messages reflecting changes in jEdit's state.

The message classes derived from `EBMessage` cover the opening and closing of the application, changes in the status of buffers and views, changes in user settings, as well as changes in the state of other program features. A full list of messages can be found in the `org.gjt.sp.jedit.msg` package.

`EBComponent`s are added and removed with the `EditBus.addToBus()` and `EditBus.removeFromBus()` methods.

Typically, the `EBComponent.handleMessage()` method is implemented with one or more `if` blocks that test whether the message is an instance of a derived message class in which the component has an interest.

```
if(msg instanceof BufferUpdate) {
    // a buffer's state has changed!
}
else if(msg instanceof ViewUpdate) {
    // a view's state has changed!
}
// ... and so on
```

If a plugin core class will respond to EditBus messages, it can be derived from `EBPlugin`, in which case no explicit `addToBus()` call is necessary. Otherwise, `EditPlugin` will suffice as a plugin base class. Note that QuickNotepad uses the latter.

17.4. The Property File

jEdit maintains a list of "properties", which are name/value pairs used to store human-readable strings, user settings, and various other forms of meta-data. During startup, jEdit loads the default set of properties, followed by plugin properties stored in plugin JAR files, finally followed by user properties.

Some properties are used by the plugin API itself. Others are accessed by the plugin using methods in the `jEdit` class.

Property files contained in plugin JARs must end with the filename extension `.props`, and have a very simple syntax, which the following example illustrates:

```
# Lines starting with '#' are ignored.
name=value
another.name=another value
long.property=Long property value, split over \
    several lines
escape.property=Newlines and tabs can be inserted \
    using the \t and \n escapes
backslash.property=A backslash can be inserted by writing \\.
```

Now we look at the `QuickNotepad.props` file which contains properties for the QuickNotepad plugin. The first type of property data is information about the plugin itself; these are the only properties that must be specified in order for the plugin to load:

```
# general plugin information
plugin.QuickNotepadPlugin.name=QuickNotepad
plugin.QuickNotepadPlugin.author=John Gellene
plugin.QuickNotepadPlugin.version=4.1
plugin.QuickNotepadPlugin.docs=QuickNotepad.html
plugin.QuickNotepadPlugin.depend.0=jedit 04.00.01.00
```

These properties are described in detail in the documentation for the `EditPlugin` class and do not require further discussion here.

Next in the file comes a property that sets the title of the plugin's dockable window. Dockable windows are discussed in detail in Section 17.6.

```
# dockable window name
quicknotepad.title=QuickNotepad
```

Next, we see menu item labels for the plugin's actions. Actions are discussed in detail in Section 17.5.

```
# action labels
quicknotepad.label=QuickNotepad
quicknotepad.choose-file.label=Choose notepad file
quicknotepad.save-file.label=Save notepad file
quicknotepad.copy-to-buffer.label=Copy notepad to buffer
```

Next, the plugin's menu is defined. See Section 17.2.

```
# application menu items
quicknotepad.menu.label=QuickNotepad
quicknotepad.menu=quicknotepad - quicknotepad.choose-file \
  quicknotepad.save-file quicknotepad.copy-to-buffer
```

We have created a small toolbar as a component of QuickNotepad, so file names for the button icons follow:

```
# plugin toolbar buttons
quicknotepad.choose-file.icon=Open.png
quicknotepad.save-file.icon=Save.png
quicknotepad.copy-to-buffer.icon=Edit.png
```

The menu item labels corresponding to these icons will also serve as tooltip text.

Finally, the properties file set forth the labels and settings used by the option pane:

```
# Option pane labels
options.quicknotepad.label=QuickNotepad
options.quicknotepad.file=File:
options.quicknotepad.choose-file=Choose
options.quicknotepad.choose-file.title=Choose a notepad file
options.quicknotepad.choose-font=Font:
options.quicknotepad.show-filepath.title=Display notepad file path

# Initial default font settings
options.quicknotepad.show-filepath=true
options.quicknotepad.font=Monospaced
options.quicknotepad.fontstyle=0
options.quicknotepad.fontsize=14

# Setting not defined but supplied for completeness
options.quicknotepad.filepath=
```

17.5. The Action Catalog

Actions define procedures that can be bound to a menu item, a toolbar button or a keyboard shortcut. Actions are short scripts written in BeanShell, jEdit's macro scripting language. These scripts either direct the action themselves, delegate to a method in one of the plugin's classes that encapsulates the action, or do a little of both. The scripts are usually short; elaborate action protocols are usually contained in compiled code, rather than an interpreted macro script, to speed execution.

Actions are defined by creating an XML file entitled `actions.xml` and placing it in the plugin JAR file.

The `actions.xml` file from the QuickNotepad plugin looks as follows:

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

<!DOCTYPE ACTIONS SYSTEM "actions.dtd">
```

```
<ACTIONS>
    <ACTION NAME="quicknotepad.choose-file">
        <CODE>
            view.getDockableWindowManager()
              .getDockable(QuickNotepadPlugin.NAME).chooseFile();
        </CODE>
    </ACTION>

    <ACTION NAME="quicknotepad.save-file">
        <CODE>
            view.getDockableWindowManager()
              .getDockable(QuickNotepadPlugin.NAME).saveFile();
        </CODE>
    </ACTION>

    <ACTION NAME="quicknotepad.copy-to-buffer">
        <CODE>
            view.getDockableWindowManager()
              .getDockable(QuickNotepadPlugin.NAME).copyToBuffer();
        </CODE>
    </ACTION>
</ACTIONS>
```

This file defines three actions. They use the current view's `DockableWindowManager` object and the method `getDockable()` to find the QuickNotepad plugin window and call the desired method.

When an action is invoked, the BeanShell scripts address the plugin through static methods, or if instance data is needed, the current `View`, its `DockableWindowManager`, and the plugin object return by the `getDockable()` method.

If you are unfamiliar with BeanShell code, you may nevertheless notice that the code statements bear a strong resemblance to Java code, with one exception: the variable `view` is never assigned any value.

For complete answers to this and other BeanShell mysteries, see Part III in *jEdit 4.1 User's Guide*; two observations will suffice here. First, the variable `view` is predefined by jEdit's implementation of BeanShell to refer to the current `View` object. Second, the BeanShell scripting language is based upon Java syntax, but allows variables to be typed at run time, so explicit types for variables need not be declared.

A formal description of each element of the `actions.xml` file can be found in the documentation of the `ActionSet` class.

17.6. The Dockable Window Catalog

The jEdit plugin API uses BeanShell to create the top-level visible container of a plugin's interface. The BeanShell code is contained in a file named `dockables.xml`. It

usually is quite short, providing only a single BeanShell expression used to create a visible plugin window.

The following example from the QuickNotepad plugin illustrates the requirements of the data file:

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

<!DOCTYPE DOCKABLES SYSTEM "dockables.dtd">

<DOCKABLES>
 <DOCKABLE NAME="quicknotepad">
  new QuickNotepad(view, position);
 </DOCKABLE>
</DOCKABLES>
```

In this example, the `<DOCKABLE>` element has a single attribute, the dockable window's identifier. This attribute is used to key a property where the window title is stored; see Section 17.4.

The contents of the `<DOCKABLE>` element itself is a BeanShell expression that constructs a new `QuickNotepad` object. The `view` and `position` are predefined by the plugin API as the view in which the plugin window will reside, and the docking position of the plugin.

A formal description of each element of the `dockables.xml` file can be found in the documentation of the `DockableWindowManager` class.

17.7. The QuickNotepad Class

Here is where most of the features of the plugin will be implemented. To work with the dockable window API, the top level window will be a `JPanel`. The visible components reflect a simple layout. Inside the top-level panel we will place a scroll pane with a text area. Above the scroll pane we will place a panel containing a small tool bar and a label displaying the path of the current notepad file.

We have identified three user actions that need implementation here: `chooseFile()`, `saveFile()`, and `copyToBuffer()`. As noted earlier, we also want the text area to change its appearance in immediate response to a change in user options settings. In order to do that, the window class must respond to a `PropertiesChanged` message from the EditBus.

Unlike the `EBPlugin` class, the `EBComponent` interface does not deal with the component's actual subscribing and unsubscribing to the EditBus. To accomplish this, we use a pair of methods inherited from the Java platform's `JComponent` class that are called when the window is made visible, and when it is hidden. These two methods,

`addNotify()` and `removeNotify()`, are overridden to add and remove the visible window from the list of EditBus subscribers.

We will provide for two minor features when the notepad is displayed in the floating window. First, when a floating plugin window is created, we will give the notepad text area input focus. Second, when the notepad if floating and has input focus, we will have the **Escape** key dismiss the notepad window. An `AncestorListener` and a `KeyListener` will implement these details.

Here is the listing for the data members, the constructor, and the implementation of the `EBComponent` interface:

```
public class QuickNotepad extends JPanel
    implements EBComponent
{
    private String filename;
    private String defaultFilename;
    private View view;
    private boolean floating;

    private QuickNotepadTextArea textArea;
    private QuickNotepadToolPanel toolPanel;

    //
    // Constructor
    //

    public QuickNotepad(View view, String position)
    {
        super(new BorderLayout());

        this.view = view;
        this.floating = position.equals(
            DockableWindowManager.FLOATING);

        this.filename = jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "filepath");
        if(this.filename == null || this.filename.length() == 0)
        {
            this.filename = new String(jEdit.getSettingsDirectory()
                + File.separator + "qn.txt");
            jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
                + "filepath",this.filename);
        }
        this.defaultFilename = new String(this.filename);

        this.toolPanel = new QuickNotepadToolPanel(this);
        add(BorderLayout.NORTH, this.toolPanel);
```

```
        if(floating)
            this.setPreferredSize(new Dimension(500, 250));

        textArea = new QuickNotepadTextArea();
        textArea.setFont(QuickNotepadOptionPane.makeFont());
        textArea.addKeyListener(new KeyHandler());
        textArea.addAncestorListener(new AncestorHandler());
        JScrollPane pane = new JScrollPane(textArea);
        add(BorderLayout.CENTER, pane);

        readFile();
    }

    //
    // Attribute methods
    //

    // for toolBar display
    public String getFilename()
    {
        return filename;
    }

    //
    // EBComponent implementation
    //

    public void handleMessage(EBMessage message)
    {
        if (message instanceof PropertiesChanged)
        {
            propertiesChanged();
        }
    }


    private void propertiesChanged()
    {
        String propertyFilename = jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX + "filepath");
        if(!defaultFilename.equals(propertyFilename))
        {
            saveFile();
            toolPanel.propertiesChanged();
            defaultFilename = propertyFilename.clone();
            filename = defaultFilename.clone();
            readFile();
        }
        Font newFont = QuickNotepadOptionPane.makeFont();
        if(!newFont.equals(textArea.getFont()))
        {
```

```
            textArea.setFont(newFont);
            textArea.invalidate();
        }
    }

    // These JComponent methods provide the appropriate points
    // to subscribe and unsubscribe this object to the EditBus

    public void addNotify()
    {
        super.addNotify();
        EditBus.addToBus(this);
    }


    public void removeNotify()
    {
        saveFile();
        super.removeNotify();
        EditBus.removeFromBus(this);
    }

    ...

}
```

This listing refers to a `QuickNotebookTextArea` object. It is currently implemented as a `JTextArea` with word wrap and tab sizes hard-coded. Placing the object in a separate class will simply future modifications.

17.8. The QuickNotepadToolBar Class

There is nothing remarkable about the toolbar panel that is placed inside the `QuickNotepad` object. The constructor shows the continued use of items from the plugin's properties file.

```
public class QuickNotepadToolPanel extends JPanel
{
    private QuickNotepad pad;
    private JLabel label;

    public QuickNotepadToolPanel(QuickNotepad qnpad)
    {
        pad = qnpad;
        JToolBar toolBar = new JToolBar();
        toolBar.setFloatable(false);

        toolBar.add(makeCustomButton("quicknotepad.choose-file",
```

```
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.chooseFile();
                    }
                }));
            toolBar.add(makeCustomButton("quicknotepad.save-file",
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.saveFile();
                    }
                }));
            toolBar.add(makeCustomButton("quicknotepad.copy-to-buffer",
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        QuickNotepadToolPanel.this.pad.copyToBuffer();
                    }
                }));
            label = new JLabel(pad.getFilename(),
                SwingConstants.RIGHT);
            label.setForeground(Color.black);
            label.setVisible(jEdit.getProperty(
                QuickNotepadPlugin.OPTION_PREFIX
                + "show-filepath").equals("true"));
            this.setLayout(new BorderLayout(10, 0));
            this.add(BorderLayout.WEST, toolBar);
            this.add(BorderLayout.CENTER, label);
            this.setBorder(BorderFactory.createEmptyBorder(0, 0, 3, 10));
        }

        ...

    }
```

The method `makeCustomButton()` provides uniform attributes for the three toolbar buttons corresponding to three of the plugin's use actions. The menu titles for the user actions serve double duty as tooltip text for the buttons. There is also a `propertiesChanged()` method for the toolbar that sets the text and visibility of the label containing the notepad file path.

17.9. The QuickNotepadOptionPane Class

Using the default implementation provided by `AbstractOptionPane` reduces the preparation of an option pane to two principal tasks: writing a `_init()` method to layout and initialize the pane, and writing a `_save()` method to commit any settings changed by user input. If a button on the option pane should trigger another dialog, such as a `JFileChooser` or jEdit's own enhanced `VFSFileChooserDialog`, the option pane will also have to implement the `ActionListener` interface to display additional components.

The QuickNotepad plugin has only three options to set: the path name of the file that will store the notepad text, the visibility of the path name on the tool bar, and the notepad's display font. Using the shortcut methods of the plugin API, the implementation of `_init()` looks like this:

```
public class QuickNotepadOptionPane extends AbstractOptionPane
      implements ActionListener
{
    private JTextField pathName;
    private JButton pickPath;
    private FontSelector font;

    ...

    public void _init()
    {
        showPath = new JCheckBox(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "show-filepath.title"),
        jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX +  "show-filepath")
            .equals("true"));
        addComponent(showPath);

        pathName = new JTextField(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "filepath"));
        JButton pickPath = new JButton(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX
            + "choose-file"));
        pickPath.addActionListener(this);

        JPanel pathPanel = new JPanel(new BorderLayout(0, 0));
        pathPanel.add(pathName, BorderLayout.CENTER);
        pathPanel.add(pickPath, BorderLayout.EAST);

        addComponent(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX + "file"),
            pathPanel);

        font = new FontSelector(makeFont());
        addComponent(jEdit.getProperty(
            QuickNotepadPlugin.OPTION_PREFIX + "choose-font"),
            font);
    }

    ...

}
```

Here we adopt the vertical arrangement offered by use of the `addComponent()` method with one embellishment. We want the first "row" of the option pane to contain a text field with the current notepad file path and a button that will trigger a file chooser dialog when pressed. To place both of them on the same line (along with an identifying label for the file option), we create a `JPanel` to contain both components and pass the configured panel to `addComponent()`.

The `_init()` method uses properties from the plugin's property file to provide the names of label for the components placed in the option pane. It also uses a property whose name begins with `PROPERTY_PREFIX` as a persistent data item - the path of the current notepad file. The elements of the notepad's font are also extracted from properties using a static method of the option pane class.

The `_save()` method extracts data from the user input components and assigns them to the plugin's properties. The implementation is straightforward:

```
public void _save()
{
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "filepath", pathName.getText());
    Font _font = font.getFont();

    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "font", _font.getFamily());
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "fontsize", String.valueOf(_font.getSize()));
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "fontstyle", String.valueOf(_font.getStyle()));
    jEdit.setProperty(QuickNotepadPlugin.OPTION_PREFIX
        + "show-filepath", String.valueOf(showPath.isSelected()));
}
```

The class has only two other methods, one to display a file chooser dialog in response to user action, and the other to construct a `Font` object from the plugin's font properties. They do not require discussion here.

17.10. Plugin Documentation

While not required by the plugin API, a help file is an essential element of any plugin written for public release. A single web page is often all that is required. There are no specific requirements on layout, but because of the design of jEdit's help viewer, the use of frames should be avoided. Topics that would be useful include the following:

- a description of the purpose of the plugin;

- an explanation of the type of input the user can supply through its visible interface (such as mouse action or text entry in controls);
- a listing of available user actions that can be taken when the plugin does not have input focus;
- a summary of configuration options;
- information on development of the plugin (such as a change log, a list of “to do” items, and contact information for the plugin’s author); and
- licensing information, including acknowledgments for any library software used by the plugin.

The location of the plugin’s help file is stored in the `plugin.QuickNotepad.docs` property; see Section 17.4.

17.11. Compiling the Plugin

We have already outlined the contents of the user action catalog, the properties file and the documentation file in our earlier discussion. The final step is to compile the source file and build the archive file that will hold the class files and the plugin’s other resources.

Publicly released plugins include with their source a makefile in XML format for the Ant utility. The format for this file requires few changes from plugin to plugin. Here is the version of `build.xml` used by QuickNotepad and many other plugins:

```
<project name="QuickNotepad" default="dist" basedir=".">

  <property name="jedit.install.dir"  value="../.."/>
  <property name="jar.name"  value="QuickNotepad.jar"/>

  <property name="install.dir"  value=".."/>


  <path id="project.class.path">
    <pathelement location="${jedit.install.dir}/jedit.jar"/>
    <pathelement location="."/>
  </path>


  <target name="compile">
    <javac
      srcdir="."
      deprecation="on"
      includeJavaRuntime="yes"
    >
      <classpath refid="project.class.path"/>
    </javac>
```

```
    </target>


    <target name="dist" depends="compile">
      <mkdir dir="${install.dir}"/>
      <jar jarfile="${install.dir}/${jar.name}">
        <fileset dir=".">
          <include name="**/*.class"/>
          <include name="**/*.props"/>
          <include name="**/*.html"/>
          <include name="actions.xml"/>
          <include name="dockables.xml"/>
        </fileset>
      </jar>
    </target>
  </project>
```

For a full discussion of the `Ant` file format and command syntax, you should consult the `Ant documentation site`. Modifying this makefile for a different plugin will likely only require three changes:

- the name of the plugin;
- the choice of compiler (made by inserting and deleting the comment character `'#'`); and
- the classpath variables for `jedit.jar` any plugins this one depends on.

If you have reached this point in the text, you are probably serious about writing a plugin for jEdit. Good luck with your efforts, and thank you for contributing to the jEdit project.

Chapter 18. Plugin Tips and Techniques

18.1. Bundling Additional Class Libraries

Recall that any class whose name ends with `Plugin.class` is called a plugin core class. JAR files with no plugin core classes are also loaded by jEdit; the classes they contain are made available to other plugins. Many plugins that rely on third-party class libraries ship them as separate JAR files. The libraries will be available inside the jEdit environment but are not part of a general classpath or library collection when running other Java applications.

A plugin that bundles extra JAR files must list them in the `plugin.`*`class name`*`.jars` property. See the documentation for the `EditPlugin` class for details.