jEdit 5.5 User's Guide

The jEdit all-volunteer developer team

jEdit 5.5 User's Guide

The jEdit all-volunteer developer team

Legal Notice

Part 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 originally written by Slava Pestov and is maintained by the jEdit core development team.

Chapter 1. Conventions

Several conventions are used throughout jEdit's user interface and this manual. They will be described here. Macintosh users should note how their modifier keys map to the terms used in the manual.

View>**Scrolling**>**Scroll to Current Line**	The **Scroll to Current Line** command contained in the **Scrolling** submenu of the **View** menu.
Edit>**Go to Line...**	Menu items that end with ellipsis (...) display dialog boxes.
`C`	The primary modifier key in jEdit. On MacOS X, this is actually the key known as "Command". On most other keyboards, this key is labelled "Control".
`A`	The secondary modifier key in jEdit. On MacOS X, this is actually the key labelled "Control". On most other keyboards, this key is labelled "Alt".
`S`	The standard "Shift" key.
`C+o`	Refers to pressing and holding the `Control` key, pressing and releasing `O`, and finally releasing the `Control` key.
`C+e C+j`	Refers to holding down `Control`, pressing `E`, pressing `J`, and releasing `Control`.
Default buttons	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.
`Alt`-key mnemonics	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. The "F10" key can also be pressed to put focus on the menu bar, it has the same functionality as the Alt key in Windows. Note that this functionality is not available on MacOS X with the "MacOS Adaptive" look and feel. See the section called "The Appearance Pane" for information on changing the look and feel.
Right mouse button	Used in jEdit to show context-sensitive menus. If you have a one button Macintosh mouse, a `Control`-click has the same effect.
Middle mouse button	Used by the quick copy feature (see the section called "Quick Copy"). True 3-button mice are rare these days. If you have a wheel mouse, press down on the wheel without rolling it. On a Macintosh with a one-button mouse, `Option`-click. On other platforms without a three-button mouse, `Alt`-click.

Chapter 2. Starting jEdit

Exactly how jEdit is started depends on the operating system. For example, on Unix you can run "jedit" at the command line, or select jEdit from a menu; on Windows, you can double-click on the jEdit icon or select it from the **Start** menu.

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 you find yourself launching and exiting jEdit a lot, the startup time can get a bit bothersome. 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 increased memory usage.

When running on MacOS X, the **-background** command-line switch is active by default, so that jEdit conforms to the platform convention that programs should stay open until the **Quit** command is explicitly invoked by the user, even if all windows are closed. To disable background mode on MacOS X, use the **-nobackground** switch.

For more information about command line switches that control the server feature, see the section called "Command Line Usage".

jEdit remembers open buffers, views and split window configurations between editing sessions, so you can get back to work immediately after starting jEdit. This feature can be disabled in the **General** pane of the **Utilities**>**Options** dialog box see the section called "The General Pane".

> **The edit server and security**
>
> Since Java does not provide any interprocess communication facility other than TCP/IP, jEdit takes extra precautions to prevent remote attacks.
>
> Not only does the edit 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.

Command Line Usage

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

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

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

Command-line switches begin with a "-". Some take a parameter. A file whose name begins with "-" can be opened like so:

```
$ jedit -- -myfile
```

Miscellaneous Options

Option	Effect
-log=_level_	Set the minimum log level to an integer between 1 and 9. Default is 7. Has no effect when connecting to another instance via the edit server.
-usage	Show a brief command line usage message without starting jEdit. This message is also shown if an invalid switch was specified.
-version	Show the version number without starting jEdit.
-nosplash	Don't show the splash screen on startup.
--	Specifies the end of command-line processing. Further parameters are treated as file names, even if they begin with a dash.

Configuration Options

Option	Effect
-plugins	Enable loading of plugins. Has no effect when connecting to another instance via the edit server. See Chapter 9, *Installing and Using Plugins*.
-noplugins	Disable loading of plugins. Has no effect when connecting to another instance via the edit server.
-restore	Restore previously open files on startup. This is the default. This feature can also be set permanently in the **General** pane of the **Utilities**> **Options** dialog box; see the section called "The General Pane".
-norestore	Do not restore previously open files on startup.
-run=_script_	Run the specified BeanShell script. There can only be one of these parameters on the command line. See the section called "Running Scripts from the Command Line".
-settings=_dir_	Store 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. See the section called "The jEdit Settings Directory".
-nosettings	Start jEdit without loading user-specific settings.
-startupscripts	Run startup scripts. This is the default. Has no effect when connecting to another instance via the edit server. See the section called "Startup Scripts".
-nostartupscripts	Disable startup scripts. Has no effect when connecting to another instance via the edit server.

Edit Server Options

See Chapter 2, *Starting jEdit* for a brief description of the edit server.

Option	Effect
-background	Run jEdit in background mode. In background mode, the edit server will continue listening for client connections even after all views are closed. Has no effect when connecting to another instance via the edit server.
-nobackground	Disable background mode. This is the default. Has no effect when connecting to another instance via the edit server.

Option	Effect
-gui	Open an initial view. This is the default. Has no effect when connecting to another instance via the edit server.
-nogui	Do 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. Has no effect when connecting to another instance via the edit server.
-newplainview	Opens the specified files in a new plain view. For more information about views, see the section called "Multiple Views".
-newview	Opens the specified files in a new view.
-reuseview	Opens the specified files in an existing view.
-quit	Exits the currently running editor instance.
-server	Store the server port info in the file named server inside the settings directory.
-server=name	Store the server port info in the file named *name*. File names for this parameter are relative to the settings directory.
-noserver	Do not attempt to connect to a running edit server, and do not start one either.
-wait	Keeps the client open until the user closes the specified buffer in the server instance. Does nothing if passed to the initial jEdit instance. Use this switch if jEdit is being invoked by another program as an external editor; otherwise the client will exit immediately and the invoking program will assume you have finished editing the given file.

Java Virtual Machine Options

To enable AntiAliasing in the TextArea, see the section called "The Text Area Pane".

It is possible to pass command line options to the Java Virtual Machine (JVM). These options can change certain things about how Java runs, such as the maximum heap size, or whether antialiasing is used in certain places.

For operating systems such as Linux where jEdit is started via a shell script, you can easily edit the jedit script and place JVM arguments in the correct place. If you are using the **-jar** command line option with the java command to run jEdit (which is how the default shell scripts do it), remember that the **-jar** parameter must be the last java option, followed immediately by the path to jedit.jar and then any jEdit command line options.

On a Windows install that uses jEdit.exe, the JVM options are located in a separate file, called jEdit.l4j.ini. Create or edit this file in the same directory as jEdit.exe and place one JVM option per line.

On Mac OS X, the jEdit.app bundle gets JVM options from a file called Contents/Info.plist, which can be edited with a text editor.

There is no complete list of options to java, since it can vary from one platform to another. Some of can be found by typing the commands java -? or man java. Common JVM options that are used with jEdit and work on all platforms are:

Option	Effect
-Dawt.useSystemAAFontSettings=on	Antialias the text in AWT components.

Option	Effect
`-Dswing.aatext=true`	Antialias the text in Swing components.
`-Djedit.home=/path/to/jedit`	Sets/overrides the java System property `jedit.home` to be the path to the jEdit install. This tells jEdit where to find its site properties, default keymaps, macros, edit modes, and documentation. You can override this setting to create a custom install that is shared by multiple users. See the section called "Site Properties" for more information.
`-mx768m`	Sets maximum heap size to 768 megabytes. Adjust this value depending on your own personal needs / plugins. On at least one platform, `-Xmx768m` works when `-mx768m` does not (or vice-versa).

Chapter 3. jEdit Basics

Interface Overview

A *View* is the jEdit term for an editor main window. It is possible to have multiple views open at once, and each View can be split into multiple panes. jEdit remembers the state of open views between editing sessions.

An open file is referred to as a *Buffer*. Unlike some editors where each buffer gets its own View, jEdit completely separates the two concepts. A buffer might be visible in several views, or none at all.

A *TextArea* is an editor for a buffer. An *EditPane* contains a TextArea plus optional buffer switcher. A View contains one EditPane by default, and additional panes are created whenever the View is split.

The drop-down buffer switcher list at the top of each EditPane shows a BufferSet, or a set of open buffers (see the section called "Buffer Sets and closing buffers"). Selecting a buffer on this list will make it visible in the TextArea. 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.

With the new Tango icon theme, these symbols are slightly different, a red square is shown for buffers with unsaved changes, a lock is shown for read-only buffers, and a yellow square is shown for new buffers which don't yet exist on disk.

As with most other graphical applications, there is a tool bar at the top of the View which provides quick access to frequently-used commands. Also, clicking the TextArea with the right mouse button displays a popup menu which also facilitates quick access to various commands. Both the tool bar and the right-click menu can be completely customized to suit your tastes in the **Utilities**>**Options** dialog box; see the section called "The Context Menu Pane" and the section called "The Tool Bar Pane".

Most of the View is taken up by the TextArea. If you've ever used a graphical user interface before, the TextArea will be instantly familiar. Text can be inserted simply by typing. More details on text insertion and deletion can be found in the section called "Inserting and Deleting Text".

The strip on the left of the TextArea is called a *gutter*. The gutter displays marker and register locations, as well as folding arrows; it will also display line numbers if the **View**>**Line Numbers** (shortcut: `C+e C+t`) command is invoked. Note this menu toggle action has the side-effect of changing the persistent jEdit properties for the Gutter, which can also be set from the **Gutter** pane of the **Utilities**>**Options** dialog box.

The gutter is divided into two sections. Right-clicking on the left side gives you a context menu, while right-clicking on the right side (where line numbers might be) toggles a marker at that position. Text can be selected by left-clicking and dragging on right side of the gutter, over the range of lines you wish to select.

Multiple Views

As documented at the beginning of this chapter, multiple Views (main windows) can be open at once.

View>**New View** creates a new View, or main window.

View>**New Plain View** creates a new View but without any tool bars or dockable windows. This can be used to open a small 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 2, *Starting jEdit* for information about starting jEdit in background mode.

View>**Split Horizontally** (shortcut: `C+2`) splits the View into two TextAreas, placed above each other.

View>**Split Vertically** (shortcut: `C+3`) splits the View into two TextAreas, placed next to each other.

Macros>**Interface**> **Splitpane Grow** grows the size of the currently focused TextArea.

View>**Unsplit Current** (shortcut: `C+0`) removes the split containing the current TextArea only.

View>**Unsplit All** (shortcut: `C+1`) removes all splits from the View.

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

View>**Go to Previous Text Area** (shortcut: `A+PAGE_UP`) shifts keyboard focus to the previous TextArea.

View>**Go to Next Text Area** (shortcut: `A+PAGE_DOWN`) shifts keyboard focus to the next TextArea.

Switching Buffers

Each EditPane has an optional drop-down BufferSwitcher at the top. The BufferSwitcher shows the current buffer and can also be used to switch the current buffer, using menu item commands and their keyboard shortcuts.

View>**Go to Previous Buffer** (keyboard shortcut: `C+PAGE_UP`) switches to the previous buffer in the list.

View>**Go to Next Buffer** (keyboard shortcut: `C+PAGE_DOWN`) switches to the next buffer in the list.

View>**Go to Recent Buffer** (keyboard shortcut: `C+BACK_QUOTE`) flips between the two most recently edited buffers.

View>**Show Buffer Switcher** (keyboard shortcut: `A+BACK_QUOTE`) has the same effect as clicking on the buffer switcher combo box.

If you prefer an alternative graphical paradigm for switching buffers, take a look at one of these plugins:

• BufferList

• BufferSelector

• BufferTabs

If you decide to use one of these plugins, you can hide the popup menu buffer switcher in the **View** pane of the **Utilities**>**Options** dialog box.

A number of plugins that implement fast keyboard-based buffer switching are available as well:

• FastOpen

• OpenIt

- SwitchBuffer

Buffer Sets and closing buffers

The *Buffer Sets* feature helps keep the buffer lists local and manageable when using jEdit in a multiple-View and multiple-EditPane environment.

As mentioned in the previous section, each Edit Pane can show a Buffer Switcher, which displays the contents of a Buffer Set. In jEdit 4.2, all EditPane buffer switchers showed the same BufferSet: a global list of all buffers that were opened from any EditPane in any View. When using many Views and EditPanes, this resulted in large lists of buffers, and made the next/previous buffer actions useless with many Views, EditPanes and Buffers.

Since jEdit 4.3, it is possible to have more narrow scopes for the Buffer Sets of an Edit Pane. This makes the 'next-buffer' and 'previous-buffer' actions switch between buffers that are local to the view or pane.

The three *BufferSet scopes* are:

1. **Global**: Includes all buffers open from any EditPane.

2. **View**: EditPanes in the same View share the same BufferSet. Opening a buffer in one View will not affect the other views.

3. **EditPane**: Each EditPane can have its own independent BufferSet.

File > **Close** removes the current buffer from the EditPane's BufferSet only. If it was the last BufferSet to contain that buffer, the buffer is also closed.

The **File** > **Close (global)** action closes the buffer in all EditPanes, as the jEdit 4.2 **File** > **Close** action did before.

When **Exclusive Buffersets** are enabled, any time a buffer is visited in one EditPane, it should be automatically closed in other EditPanes which use a disjoint (non-intersecting) BufferSet.

Close Others will clear the BufferSet of the current EditPane by performing a **Close** on all items except those buffers which are displayed in another active EditPane.

Switching Bufferset Scopes

Bufferset Scope can be set from **Utilities** >**Options** > **View** > **BufferSet scope:**.

The statusbar shows you which *BufferSet scope* is active (look for the letter "G", "E" or "V"). Double-clicking on that will allow you to change the scope without going into global options.

The BufferSet Scope can also be changed from **View>Buffer Sets** > **(Global|View|EditPane) Buffer Set**. A change to the bufferset scope affects all editpanes immediately.

Sorting of Buffer Sets and Buffer Switchers

Buffer Switchers and Buffer Sets can be sorted independently. Both can be sorted by name or by path. Both can be left unsorted. Possibly the most useful combination is to sort the Buffer Switchers and leave the Buffer Sets unsorted. In this case, it is easy to find the buffer to work with in the Buffer Switcher since the buffer names are sorted alphabetically, and leaving the Buffer Set unsorted means the "go to previous" and "go to next" actions follow the order in which the buffers were last used rather than by name. However, some users prefer Buffer Sets to also be sorted by name or path, so jEdit supports multiple sorting methods.

Window Docking Layouts

A docking layout is similar to an Eclipse "Perspective" in that it describes a set of dockable windows that are visible to the user at any given time, hiding the rest.

Various jEdit and plugin windows can be docked into the View for convenience. Dockable windows have a popup button in their top-left corner. Clicking this button displays a menu with commands for docking the window in one of four sides of the View.

On each side of the TextArea where there are docked windows, a strip of buttons is shown. There is a button for activating each docked window, as well as a close box and a popup menu button, which when clicked shows a menu for moving or undocking the currently selected window. The popup menu also contains a command for opening a new floating instance of the current window.

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

After you have customized the layout of your dockables and wish to save it for export/import, use the actions **View** - **Docking** - **Save/Load Docking Layout**.

It is possible to configure a Dockable layout for just one or a handful of edit modes. This makes it possible to save or load your dockable layout with the same keyboard shortcut (or automatically) based on the edit mode of your current buffer.

It is also possible to save/load a layout for a particular edit mode. The loading and saving can be done automatically, as configured in the global options docking pane when the mode of the buffer changes, or manually in response to invoking **View** - **Docking** - **Save/Load Docking Layout for current mode.**

Dockable windows can be further configured in the **Docking** pane of the **Utilities**>**Global Options** dialog box. See the section called "The Docking Pane" for details.

For keyboard/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. For example, this can be used to view two different directories side-by-side in two file system browser windows.

  A new floating instance can also be opened from the dockable window's popup menu.

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 the section called "The Global Options Dialog Box".

The Status Bar

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

- Caret position information:

  - The offset of the caret from the beginning of the file

- 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).

- The percent offset of the caret from the start of the file. This is based on the line number of the caret and the total number of lines in the file, so this is the same as the relative position of the right scroll bar in the main text area.

  Double-clicking on the caret location indicator displays the **Edit**>**Go to Line** dialog box; see the section called "Working With Lines".

- 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:

  - the section called "The Buffer Options Dialog Box"

  - the section called "Edit Modes"

  - the section called "Folding"

  - the section called "Character Encodings"

- 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 the section called "Wrapping Long Lines".

  - Multiple selection - see the section called "Multiple Selection".

  - Rectangular selection - see the section called "Rectangular Selection".

  - Overwrite mode - see the section called "Inserting and Deleting Text".

  - Line separator - see the section called "Line Separators".

  - Buffer Set Scope - see the section called "Buffer Sets and closing buffers".

- A **Task Monitor** widget, which spins with activity when a background task is running, and also lists how many tasks are running. Clicking on this will open the Task Monitor dockable (the section called "Task Monitor, and background I/O tasks").

- 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 order and visibility of each of the above items can be controlled in the **Status Bar** pane of the **Utilities**>**Options** dialog box; see the section called "The Status Bar Pane".

The Action Bar

The action bar allows almost any editor feature to be accessed from the keyboard.

Utilities>**Action Bar** (shortcut: C+ENTER) displays the action bar at the bottom of the View and gives it keyboard focus. The action bar remembers previously entered strings; see Appendix C, *History Text Fields* for details.

To use the action bar, input a command and press `Enter`. To see a list of all actions, evaluate the BeanShell expression `jEdit.getActionNames()` (in a BeanShell interpreter, not in the action bar).

Action invocations

Each menu item and tool bar button is bound to an *action*. To find out the name of an action, invoke the menu item or click the tool bar button, and look in the action bar's history.

If a substring or an action name is entered, pressing `Tab` shows a popup listing matching actions. An action can be selected using the `Up` and `Down` arrow keys, or by entering more characters of its name.

Pressing `Enter` with an incomplete substring invokes the action that would be first in the completion popup's list.

For example, entering **d-o** will invoke **combined-options**, which has the same effect as invoking **Utilities**> **Options**.

Buffer-local properties

Entering **buffer.property=value** sets the value of the buffer-local property named **property** to **value**. Buffer-local properties are documented in the section called "Buffer-Local Properties".

For example, entering **buffer.tabSize=4** changes the current buffer's tab size to 4.

See the section called "Buffer-Local Properties" for information about buffer-local properties.

Global properties

Entering **property=value** sets the value of the global property named **property** to **value**. This feature is primarily intended to help plugin developers, since the properties jEdit uses to store its settings are not currently documented.

Command repetition

To repeat a command multiple times, enter a number in the action bar, then press the key-combination that invokes the command. For example, "C+ENTER 1 4 C+d" will delete 14 lines; "C +ENTER 9 #" will insert "#########" in the buffer. Note: The space characters in these examples should not be typed; they are only here to visually separate the keys to be typed.

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 4. Working With Files

Creating New Files

File>**New** (shortcut: C+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.

Opening Files

File>**Open** (shortcut: C+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 the section called "The File System Browser (FSB)".

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**>**Options** dialog box; see the section called "The General Pane".

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 the section called "The File System Browser (FSB)").

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 the section called "The Buffer Options Dialog Box" for details.

Saving Files

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

File>**Save** (shortcut: C+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 a different location but does not rename the buffer, and does not 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: C+e C+s) saves all open buffers to disk, asking for confirmation first. The confirmation dialog can be disabled in the **General** pane of the **Utilities**>**Options** dialog box.

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 in the same directory 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 this creates a new i-node so while jEdit retains file permissions, the owner and group of the file are reset, and if it is a hard link the link is broken. The "two-stage save" feature can be disabled in the **General** pane of the **Utilities**>**Options** dialog box; see the section called "The General Pane".

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, with their respective file names, enclosed in hash ("#") characters. For example, `/home/user/project/program.c` will be autosaved to `home/user/project/#program.c#`.

If you wish autosave files to be stored in an alternate location, you can specify the **Autosave Directory** as a **Saving and Backup** option.

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.

If **Autosave of Untitled Buffers** is enabled, then jEdit will allow you to exit without asking you to save dirty untitled buffers, and will silently restore them for you on next jEdit startup.

The autosave interval can be changed in the **Autosave and Backup** pane of the **Utilities**>**Options** dialog box; see the section called "The Saving and Backup Pane".

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 the **Saving and Backup** pane of the **Utilities**>**Global Options** see the section called "The Saving and Backup Pane".

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 (with the full path to the original file under it). Otherwise, they are saved in the same directory as the original file.

- 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.

- 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.

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.

There are several ways to change a buffer's line separator:

- In the **Utilities**>**Buffer Options** dialog box. See the section called "The Buffer Options Dialog Box".

- By clicking the line separator indicator in the status bar. See the section called "The Status Bar".

- From the keyboard, if a keyboard shortcut has been assigned to the **Toggle Line Separator** command in the **Shortcuts** pane of the **Utilities**> **Options** dialog box. By default, this command does not have a keyboard shortcut.

By default, new files are saved with your operating system's native line separator. This can be changed in the **Encodings** pane of the **Utilities**>**Options** dialog box; see the section called "The Encodings Pane". Note that changing this setting has no effect on existing files.

Character Encodings

A character encoding is a mapping from a set of characters to their on-disk representation. jEdit can use any encoding supported by the Java platform.

Buffers in memory are always stored in UTF-16 encoding, which means each character is mapped to an integer between 0 and 65535. UTF-16 is the native encoding supported by Java, and has a large enough range of characters to support most modern languages.

When a buffer is loaded, it is converted from its on-disk representation to UTF-16 using a specified encoding.

The default encoding, used to load files for which no other encoding is specified, can be set in the **Encodings** pane of the **Utilities**> **Options** dialog box; see the section called "The Encodings Pane". Unless you change this setting, it will be your operating system's native encoding, for example MacRoman on the MacOS, windows-1252 on Windows, and ISO-8859-1 on Unix.

An encoding can be explicitly set when opening a file in the file system browser's **Commands**>**Encoding** menu.

Note that there is no general way to auto-detect the encoding used by a file, however jEdit supports "encoding detectors", of which there are some provided in the core, and others may be provided by plugins through the services api. From the encodings option pane the section called "The Encodings Pane", you can customize which ones are used, and the order they are tried. Here are some of the encoding detectors recognized by jEdit:

- **BOM**: UTF-16 and UTF-8Y files are auto-detected, because they begin with a certain fixed character sequence. Note that plain UTF-8 does not mandate a specific header, and thus cannot be auto-detected, unless the file in question is an XML file.

- **XML-PI**: Encodings used in XML files with an XML PI like the following are auto-detected:

  ```
  <?xml version="1.0" encoding="UTF-8">
  ```

- **html**: Encodings specified in HTML files with a `content=` attribute in a `meta` element may be auto-detected:

  ```
  <html><head><meta http-equiv="Content-Type" content="text/html; charset=utf-8'
  ```

- **python**: Python has its own way of specifying encoding at the top of a file.

  ```
  # -*- coding: utf-8 -*-
  ```

- **buffer-local-property**: Enable buffer-local properties' syntax (see the section called "Buffer-Local Properties") at the top of the file to specify encoding.

  ```
  #  :encoding=ISO-8859-1:
  ```

The encoding that will be used to save the current buffer is shown in the status bar, and can be changed in the **Utilities**>**Buffer Options** dialog box. Note that changing this setting has no effect on the buffer's contents; if you opened a file with the wrong encoding and got garbage, you will need to reload it. **File**>**Reload with Encoding** is an easy way.

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.

Commonly Used Encodings

While the world is slowly converging on UTF-8 and UTF-16 encodings for storing text, a wide range of older encodings are still in widespread use and Java supports most of them.

The simplest character encoding still in use 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. Each ASCII character consists of 7 bits, there is a limit of 128 distinct characters, which makes it unsuitable for anything other than English text. jEdit will load and save files as ASCII if the `US-ASCII` encoding is used.

Because ASCII is unsuitable for international use, most operating systems use an 8-bit extension of ASCII, with the first 128 values mapped to the ASCII characters, and the rest used to encode accents, umlauts, and various more esoteric 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 `windows-1252`. In the Unix world, the `8859_1` character encoding has found widespread usage.

On Windows, various other encodings, referred to as *code pages* and identified by number, are used to store non-English text. The corresponding Java encoding name is `windows-` followed by the code page number, for example `windows-850`.

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

The File System Browser (FSB)

Utilities>**File System Browser** displays the file system browser. By default, the file system browser is shown in a floating window. This window can be docked using the commands in its top-left corner popup menu; see the section called "Window Docking Layouts".

The FSB can be customized in the **Utilities**>**Options** dialog box; see the section called "The File System Browser Panes".

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, *History Text Fields*. 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**>**Options** dialog box.

The **Path:** Text Box can be used to navigate to a specific directory. Environment variables are expanded here, allowing for both `$VARNAME` or `%VARNAME%` syntax.

`A+Up` is a keyboard shortcut that brings you to the parent directory.

`A+Left` and `A+Right` navigate back and forward through the visited directory stacks, in a Netscape/Konqueror/IE like fashion.

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, *Glob Patterns* 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.

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.

The Commands Menu

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

- **Parent Directory** - moves up in the directory hierarchy. The Alt+Left arrow keyboard shortcut achieves the same thing.

- **Reload Directory** - reloads the file list from disk. F5 does this also.

- **Root Directory** - on Unix, goes to the root directory (/). On Windows and MacOS X, lists all mounted drives and network shares. The forward slash (/) achieves this too.

- **Home Directory** - displays your home directory. Keyboard shortcut: ~

- **Directory of Current Buffer** - displays the directory containing the currently active buffer. Shortcut: -

- **New File** (Ctrl+N) - 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 the section called "Search and Replace" 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 the section called "Character Encodings".

The Plugins Menu

Clicking the **Plugins** button displays a menu containing plugin commands. For information about plugins, see Chapter 9, *Installing and Using Plugins*.

The Favorites Menu

Clicking the **Favorites** button displays a menu showing all files and directories in the favorites list. The **Add to Favorites** item adds the currently selected file to the favorites list. If nothing is selected, the current directory is added. To remove a file from the favorites, invoke **Edit Favorites**, which will show the favorites list in the file system view, then select **Delete** from the right-click menu of the entry you want to remove.

Keyboard Shortcuts

Completion behaves differently in file dialogs than in the stand-alone file system browser window.

In the file dialog, keyboard input goes in the file name field by default. Pressing `Enter` opens the file or directory path that is either fully or partially entered in the file name field. Typing the first few characters of a file's name selects that file. If the file name field is empty and nothing is selected, / lists the root directory on Unix and the list of drives on Windows. There are two handy abbreviations that may be used in file paths: ~ expands to the home directory, and - expands to the current buffer's directory.

For example, to open a file `/home/slava/jEdit/doc/TODO.txt`, you might enter `~/j/d/to`.

In the stand-alone file system browser, keyboard input is handled slightly differently. There is no file name field, instead shortcuts are active when the file tree has keyboard focus. Additionally, pressing /, ~ or - always immediately goes to the root, home and current buffer's directory, respectively.

Reloading From Disk

When a view is brought to the foreground, jEdit checks if any open buffers were modified on disk by another application. All affected buffers are listed in a dialog box. By default, buffers without

unsaved changes are automatically reloaded. This feature can be disabled, or changed to prompt if files should be reloaded first, in the **General** pane of the **Utilities**>**Options** dialog box; see the section called "The Global Options Dialog Box".

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.

Task Monitor, and background I/O tasks

To improve responsiveness and perceived performance, jEdit executes all buffer input/output tasks asynchronously. Plugins should do the same. When a task such as this is in progress, the status bar should display the number of running tasks and an icon that spins. If you do not see this, you can add the widget from the the section called "The Status Bar Pane".

The **Utilities**>**Troubleshooting**> **Task Monitor** command displays a window with more detailed status information and progress meters for each task. By default, the **Task Monitor** is shown in a floating window. This window can be docked using the commands in its top-left corner popup menu; see the section called "Window Docking Layouts". Tasks can be aborted in this window, however note that aborting a buffer save can result in data loss.

Printing

jEdit printing is much like you'd expect from any text editor application, you can select the printer, specify specific pages to print, the number of copies, and a number of printer and page options that are supported by the printer such as color printing, duplex printing, and so on. The specific options available are determined by the chosen printer. Print preview is also available, as are several jEdit specific settings such as printing a header and/or footer, line numbers, etc.

The printer dialog is accessed by **File**>**Print** (shortcut: `C+p`) which prints the current buffer. All printing options are set in this dialog. At this time, there is no option to print multiple buffers at once.

The print output can be customized in the printer dialog. In addition to the standard printing options, the following jEdit-specific printing 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.

- 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.

- If folded regions should be printed.

In the Utilities - Global Options - Text Area settings, there is an option to show page break markers. If selected, horizontal lines will be displayed in the text area to indicate where page breaks will be when printing.

There is also an action available in the View menu, to toggle the the page break markers on and off. Page break markers are just what they sound like, they show where page breaks would be for printing. Note that jEdit is not a word processor, so the page breaks are calculated based on printer settings and print font size. It is not possible to set a page break at a specific point in a file.

Closing Files and Exiting jEdit

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

File>**Close Others** closes all buffers except the current buffer. If any have unsaved changes, jEdit will ask if they should be saved first.

File>**Close All** closes all buffers in the current View. 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>**Close (Global)** closes the current buffer in all Views. If it has unsaved changes, jEdit will ask if they should be saved first.

File>**Close All (Global)** (shortcut: `C+e  C+w`) closes all buffers in all Views. 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.

Close Others, **Close All**, and **Close All (Global)** can ask for confirmation before closing. The confirmation dialog can be shown by checking the '"Close All" asks for confirmation' box in Utilities - Global Options - General.

File>**Exit** (shortcut: `C+q`) will completely exit jEdit, prompting if unsaved buffers should be saved first.

Chapter 5. Editing Text

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 screen-full, 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 the section called "Wrapping Long Lines".

`C+HOME` and `C+END` move the caret to the beginning and end of the buffer, respectively.

More advanced caret movement is covered in the section called "Working With Words", the section called "Working With Lines" and the section called "Working With Paragraphs".

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**>**Options**; see the section called "The Shortcuts Pane".

By default, the shortcuts are assigned as follows:

- `HOME` is bound to **Smart Home**.

- `END` is bound to **Smart End**.

- `S+HOME` is bound to **Select to Smart Home Position**.

- `S+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**.

Selecting Text

A *selection* is a block of text marked for further manipulation. Range selections are equivalent to selections in most other text editors; they cover text between two points in a buffer. In addition to the standard text-selection mode, jEdit also allows **rectangular selections** that cover a rectangular area (some text editors refer to these as "column selections"). Furthermore, several chunks of text can be selected and operated on simultaneously.

Range Selection

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: `C+a`) selects the entire buffer.

Edit>**More Selection**>**Select None** (shortcut: `ESCAPE`) deactivates the selection.

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.

Alternatively, invoking **Edit**>**More Selection**>**Rectangular Selection** (shortcut: `A+\`) toggles rectangular selection mode. In rectangular selection mode, dragging the mouse always creates a rectangular selection, and keyboard commands that would normally create a range selection create a rectangular selection instead. A status bar indicator is shown when this mode is enabled.

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.

Inserting text into a rectangular selection repeats the text going down as many times as necessary, and shifts the selection to the right. This makes it behave like a "tall" caret.

Rectangles can be deleted, copied, pasted, and operated on using ordinary editing commands. If necessary, rectangular selections are automatically filled in with whitespace to maintain alignment.

Rectangular selections can extend beyond the end of a line into "virtual space". Furthermore, if keyboard rectangular selection mode is on or if the `Control` key is being held down, clicking beyond the end of a line will insert the appropriate amount of whitespace in order to position the cursor at the clicked location.

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**> **Options** dialog box. For information about soft wrap, see the section called "Wrapping Long Lines".

Multiple Selection

Edit>**More Selection**>**Multiple Selection** (keyboard shortcut: `C+\`) 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 the section called "The Status Bar".

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: `C+e  C+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.

Keyboard Focus

When the cursor disappears, that means the text area no longer has focus, and when you type, your keystrokes are probably going somewhere else, such as a dockable. To ensure that the keyboard is focused in the textarea, you can always use the mouse and click in it, but a more keyboard-friendly way is preferred when you are just about to start typing anyway. For this reason, a number of jEdit's actions have a side-effect of focusing on the text area as well. **View - Toggle Full Screen** is one example. Some others are listed below:

From the **View - Scrolling** submenu:

- **Scroll and Center Caret**

- **Scroll to Line**

From the **View - Docking** submenu:

- **Toggle Docked Areas**

- **Close current docking area**

In general, jEdit is a keyboard-friendly editor that is also mouse-friendly, so you should never be forced to use a mouse to do anything.

Inserting and Deleting Text

Text entered at the keyboard is inserted into the buffer. In overwrite mode, which can be toggled by pressing `INSERT`, one character is deleted from in front of the caret position for every character that is inserted. The caret is drawn as a horizontal line while overwrite mode is active. The status bar also indicates if overwrite mode is active; see the section called "The Status Bar" for details.

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

When inserting text, the `TAB` and `ENTER` keys might not behave entirely like you expect because of various indentation features; see the section called "Tabbing and Indentation" 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 the section called "Working With Words", the section called "Working With Lines" and the section called "Working With Paragraphs".

Undo and Redo

Edit>**Undo** (shortcut: `C+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.

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

By default, the last 100 edits is retained; older edits cannot be undone. The maximum number of undos and whether undos are reset when a buffer is saved can be changed in the **Editing** pane of the **Utilities**> **Options** dialog box; see the section called "The Editing Pane".

Working With Words

`C+LEFT` and `C+RIGHT` move 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: `C+e  w`). A selection that begins and ends on word boundaries can be created by double-clicking and dragging.

`C+BACKSPACE` and `C+DELETE` delete the word before or after the caret, respectively.

Edit>**Complete Word** (shortcut: `C+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, the longest common prefix is inserted into the buffer, and a popup is shown below the caret position listing the completions.

To insert a completion from the list, either select it using the `UP` and `DOWN` keys and press `ENTER`, press a digit to insert one of the first ten completions (1 is the first completion; 9 is the 9th; 0 is the 10th), or click the desired completion with the mouse. 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.

The default word completion uses the visible buffers (buffers being shown in an EditPane) to find completions. The set of possible words can be expanded by enabling the **Global Options - Text Area - Complete words from all open buffers** option. Setting this option will use all open buffers to search for possible completions. Note, this can degrade completion performance if many buffers are open.

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

What's a Word?

The default behavior of the `C+LEFT`, `C+RIGHT`, `C+BACKSPACE` and `C+DELETE` commands is to stop both at the beginning and the end of each word. Normally, a word is a sequence of alphanumerics, but you can add other characters as part of what jEdit considers to be a 'word', set on a global or mode basis from **Global Options - Editing - Extra Word Characters**. In addition, this behavior can be changed by remapping these keystrokes to alternative actions whose names end with **(Eat Whitespace)** in the **Shortcuts** pane of the **Utilities**>**Options** dialog box; see the section called "The Shortcuts Pane".

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: `C+e l`). A selection that begins and ends on line boundaries can be created by triple-clicking and dragging.

Edit>**Go to Line** (shortcut: `C+l`) prompts for a line number and moves the caret there. A relative offset can be used here, if it is prefixed by a + or a - sign. So for example `-5` moves the caret up by 5 lines.

Edit>**More Selection**>**Select Line Range** (shortcut: `C+e C+l`) prompts for two line numbers and selects all text between them.

Edit>**Text**>**Delete Line** (shortcut: `C+d`) deletes the current line.

Edit>**Text**>**Delete to Start Of Line** (shortcut: `CS+BACK_SPACE`) deletes all text from the start of the current line to the caret.

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

Edit>**Text**>**Join Lines** (shortcut: `C+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());
```

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.

`C+UP` and `C+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: `C+e p`) selects the paragraph containing the caret.

Edit>**Text**>**Format Paragraph** (shortcut: `C+e f`) splits and joins lines in the current selection to make it fit within the wrap column position. If nothing is selected, the paragraph containing the caret is formatted instead. See the section called "Wrapping Long Lines" for information about word wrap and changing the wrap column.

Edit>**Text**>**Delete Paragraph** (shortcut: `C+e d`) deletes the paragraph containing the caret.

Wrapping Long Lines

The *word wrap* feature splits lines at word boundaries in order to fit text within a specified wrap margin. A word boundary, for the purposes of word wrap, means whitespace. Long lines without whitespace are currently not wrapped by jEdit. 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 current wrap mode is shown in the status bar; see the section called "The Status Bar". 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**> **Options** dialog box. See the section called "The Editing Pane".

- In the current buffer for the duration of the editing session,

  - By clicking the status bar indicator.

  - In the **Utilities**>**Buffer Options** dialog box. See the section called "The Buffer Options Dialog Box".

  - From the keyboard, if a keyboard shortcut has been assigned to the **Built-in Command** for **Toggle Word Wrap** in the **Shortcuts** pane of **Global Options**. By default, this command does not have a keyboard shortcut, or appear in any menu.

- 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*`:`

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 the margin is set to 0, then the width of the text area window is used to determine where to wrap lines.

If end of line markers are enabled in the **Text Area** pane of the **Utilities**>**Options** dialog box, a special character ("#") is painted at the end of wrapped lines. If you are using a font that can not display it, or wish to use a different character, you can `jEdit.setProperty("view.eolMarkerChar", "#");` from the Console beanshell and you will see it after a restart of jEdit. To enable/disable or change its color, see the section called "The Text Area Pane".

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 the section called "Working With Paragraphs".

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**>**Options** dialog box.

Scrolling

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

- `Shift` - moves the horizontal scrollbar. time.

- `Control` - scrolls a single line at a time.

- `Alt` - moves the caret up and down instead of scrolling.

- `CTRL+SHIFT` - scroll a page at a time.

- `Alt+Shift` - extends the selection up and down instead of scrolling.

Keyboard commands for scrolling the text area are also available.

View>**Scrolling**>**Scroll to Current Line** (shortcut: `C+e C+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: `C+e C+n`) moves the caret to the line in the middle of the screen.

View>**Scrolling**>**Line Scroll Up** (shortcut: `C+QUOTE`) scrolls the text area up by one line.

View>**Scrolling**>**Line Scroll Down** (shortcut: `C+SLASH`) scrolls the text area down by one line.

View>**Scrolling**>**Page Scroll Up** (shortcut: `A+QUOTE`) scrolls the text area up by one screenful.

View>**Scrolling**>**Page Scroll Down** (shortcut: `A+SLASH`) 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.

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. Most other programs can only transfer text to and from the system clipboard; in jEdit, the system clipboard is just another register, with the special name `$`.

The Clipboard

jEdit offers the usual text transfer operations, that operate on the `$` register.

Edit>**Cut** (shortcut: `C+x`) places the selected text in the clipboard and removes it from the buffer.

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

Edit>**Paste** (shortcut: `C+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: `C+e C+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: `C+e C+a`) is the same as **Cut Append** except it does not remove the selection from the buffer.

Quick Copy

The quick copy feature is usually found in Unix text editors. Quick copy is disabled by default, but it can be enabled in the **Mouse** pane of the **Utilities**>**Options** dialog box.

The quick copy feature is accessed using the middle mouse button. If you do not have a three-button mouse, then either `Alt`-click (on Windows and Unix) or `Option`-click (on MacOS X). The quick copy feature enables the following behavior:

- 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.

- 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.

- Holding down `Shift` while clicking the middle mouse button will duplicate text between the caret and the clicked location.

- Holding down `Control` while clicking the middle mouse button on a bracket will insert all text in that bracket's scope at the caret position.

The most recently selected text is stored in the `%` register.

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.

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 cancels the operation.

Note that the content of registers other than the clipboard and quick copy register are automatically saved between jEdit sessions.

Edit>**More Clipboard**>**Cut to Register** (shortcut: `C+r C+x` *key*) stores the selected text in the specified register, removing it from the buffer.

Edit>**More Clipboard**>**Copy to Register** (shortcut: `C+r C+c` *key*) stores the selected text in the specified register, leaving it in the buffer.

Edit>**More Clipboard**>**Cut Append to Register** (shortcut: `C+r C+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: `C+r C+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: `C+r C+v key`) replaces the selection with the contents of the specified register.

The following three commands display dialog boxes instead of prompting for a register name.

Edit>**More Clipboard**>**Paste Previous** (shortcut: `C+e C+v`) displays a dialog box listing the 20 most recently copied and pasted text strings.

Edit>**More Clipboard**>**Paste Deleted** (shortcut: `C+e C+y`) is not really a register command; it displays a dialog box listing the 20 most recently deleted text strings.

Edit>**More Clipboard**>**View Registers** displays a dialog box for viewing register contents, including the clipboard and the quick copy.

Markers

MarkerSets

The MarkerSets plugin is a replacement for the built-in Markers feature of jEdit. Markers saved with MarkerSets properly update when lines are added or removed from a buffer. Furthermore, you can see markers from multiple files in the Marker Sets dockable. We recommend you use that instead of the built-in 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 **General** pane of the **Utilities**>**Options** dialog box; see the section called "The General Pane".

Markers>**Add/Remove Marker** (shortcut: `C+e C+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 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: `C+e C+COMMA`) goes to the marker immediately before the caret position.

Markers>**Go to Next Marker** (shortcut: `C+e C+PERIOD`) goes to the marker immediately after the caret position.

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

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: `C+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: `C+y key`) moves the caret to the location of the marker with the specified shortcut.

Markers>**Select to Marker** (shortcut: `C+u key`) creates a selection from the caret location to the marker with the specified shortcut.

Markers>**Swap Caret and Marker** (shortcut: `C+k key`) moves the caret to the location of the marker with the specified shortcut, and moves the marker to the former caret position. 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 the section called "Interface Overview" for information about the gutter.

Search and Replace

Searching For Text

Search>**Find** (shortcut: `C+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, *History Text Fields* 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 the section called "HyperSearch" 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".

After selecting the **Whole word** check box, searching respects the **Extra word characters** setting from the editing options for recognizing words.

To search for special characters (such as newlines or non-printable characters), inexact sequences of text, or strings that span multiple lines, we use **Regular Expressions**. Selecting the **Regular expressions** check box allows special characters to be used in the search string. Regular expression syntax is described in Appendix E, *Regular Expressions*. If you use (groups) in the search field, you back-reference them with $1 through $9 in the replace field.

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: `C+g`) locates the next occurrence of the most recent search string without displaying the search and replace dialog box.

Search>**Find Previous** (shortcut: `C+h`) locates the previous occurrence of the most recent search string without displaying the search and replace dialog box.

Replacing Text

The replace string text field of the search dialog remembers previously entered strings; see Appendix C, *History Text Fields* 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.

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, *Regular Expressions* 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"

BeanShell Replace

In BeanShell replacement mode, the search string is replaced with the return value of a BeanShell snippet. If you want to use multiple line snippet, enclose your BeanShell in braces. 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, "Writing Macros", 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:

```
<\S+
```

Replacing it with the following BeanShell snippet:

```
_0.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)
```

To replace 3d6 with 3 simulated dice throws:

```
3d6
```

Replacing it with the following BeanShell snippet (note the braces around multi-instruction beanshell code):

```
{r = new java.util.Random(); b = "";
for(i=0;i<3;i++)b+=(r.nextInt(5)+1) + " ";
return b;}
```

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

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.

By default, HyperSearch results are shown in a floating window. This window can be docked using the commands in its top-left corner popup menu; see the section called "Window Docking Layouts".

If the **Multiple results** toggle button 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.

Once the results are shown in the Hypersearch dockable, you can left-click on a result to jump to the position, or right-click to see some "hidden actions". From the top tree-node, for example, you can **Redo Hypersearch**, and **Copy to Clipboard**.

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, *Glob Patterns* 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, *History Text Fields* for details.

When the search and replace dialog box is opened, the directory and file name filter fields are set to their previous values. They can be set to match the current buffer's directory and file name extension by clicking **Synchronize**.

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, the **HyperSearch** check box can be unchecked, for stepping through search results in multiple files one at a time.

Two convenience commands are provided for performing multiple file searches.

Search>**Search in Open Buffers** (shortcut: `C+e C+b`) displays the search dialog box and selects the **All buffers** button.

Search>**Search in Directory** (shortcut: `C+e C+d`) displays the search dialog box and selects the **Directory** button.

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 search. Previously entered strings can be recalled in the search bar with the `Up` and `Down` arrow keys; see Appendix C, *History Text Fields*.

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 **View** pane of the **Utilities**>**Options** dialog box; see the section called "The View Pane".

Search>**Incremental Search Bar** (shortcut: `C+COMMA`) displays the search bar if necessary, and gives it keyboard focus.

Search>**Incremental Search for Word** (shortcut: `A+COMMA`) behaves like the above command except it places the word at the caret in the search string field. If this command is invoked while there is a selection, the selection is placed in the search string field instead.

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 `S+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 8, *Using Macros* for information about macros.

Search>**HyperSearch Bar** (shortcut: `C+PERIOD`) 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: `A+PERIOD`) performs a HyperSearch for the word at the caret. This command does not show the search bar or give it keyboard focus.

Chapter 6. Editing Source Code

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, "Writing Edit Modes".

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.

Mode Selection

File name and first line matching is done using glob patterns similar to those used in Unix shells. Glob patterns associated with edit modes can be changed in the **Editing** pane of the **Utilities**> **Options** dialog box. Note that the glob patterns must match the file name or first line exactly; so to match files whose first line contains `begin`, you must use a first line glob of `*begin*`. See Appendix D, *Glob Patterns* for a description of glob pattern syntax.

The default edit mode for files which do not match any pattern can be set in the **Editing** pane as well.

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 the section called "The Buffer Options Dialog Box". 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:
```

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**> **Options** dialog box; see the section called "The Syntax Highlighting Pane".

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**> **Options** dialog box. See the section called "The Editing Pane".

- In the current buffer for the duration of the editing session in the **Utilities**>**Buffer Options** dialog box. See the section called "The 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: `S+TAB` or `A+LEFT`) removes one level of indent from each selected line, or the current line if there is no selection.

Edit>**Indent**>**Shift Indent Right** (shortcut: `A+RIGHT`) adds one level of indent to 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: `C+e  r`) removes all whitespace from the end of each selected line, or the current line if there is no selection.

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**> **Options** dialog box. See the section called "The Editing Pane".

- In the current buffer for the duration of the editing session in the **Utilities**>**Buffer Options** dialog box. See the section called "The 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>**Indent**>**Spaces to Tabs** converts soft tabs to hard tabs in the current selection, or the entire buffer if nothing is selected.

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

Elastic Tabstops

Elastic tabstops are an alternative way to handle tabstops. Elastic tabstops differ from traditional fixed tabstops because columns in lines above and below the "cell" that is being changed are always kept aligned. As the width of text before a tab character changes, the tabstops on adjacent lines are also changed to fit the widest piece of text in that column. It provides certain explicit benefits like it saves time spent on arranging the code and works seemlessly with variable width fonts.But at the same time it can make the code look unorganized on editors that do not support elastic tabstops.

This feature is known as *elastic tabstops*. Elastic tabstops can be enabled or disabled in one of several ways:

- On a global or mode-specific basis in the **Editing** pane of the **Utilities**>**Options** dialog box. See the section called "The Editing Pane".

- In the current buffer for the duration of the editing session in the **Utilities**>**Buffer Options** dialog box. See the section called "The 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":

  `:elasticTabstops=`*flag*`:`

Note that this feature does not work with *soft tabs*. where tabs are emulated as spaces

Automatic Indent

The auto indent feature inserts the appropriate number of tabs or spaces at the beginning of a line. There are three different indentation schemes to choose from: "full", "simple", and "none". The scheme can be chosen on a global or per-edit mode basis using the **Editing** pane of the **Utilities**>**Options** dialog. It can also be changed for a specific buffer using the **Buffer Options** dialog, or with a buffer-local property. (see the section called "Buffer-Local Properties")

Automatic Indent Scheme: full

In this default scheme, the amount of indentation inserted 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.

The automatic indentation can be triggered by: pressing ENTER (this will by default only affect the indentation of the new line), pressing TAB at the beginning of, or inside the leading whitespace of a line, entering one the bracket characters defined in the edit mode, pressing one of the electricKeys for the current edit mode (more details in the section called "The PROPS Tag"), or when causing a hard wrap (see the section called "Wrapping Long Lines").

No matter what automatic indentation scheme is currently active, **Edit** > **Indent** > **Indent Selected Lines** (shortcut: C+i) indents all selected lines, or the current line if there is no selection, as if in the "full" scheme.

Electric keys

Electric keys cause reapplying of the indentation rules to the current line. Thanks to the electric keys the following code fragments are indented properly on-line:

- Java, C: brackets. If indenting brackets are defined for the language, they are implicitly considered electric keys. Thus a closing bracket is placed in its correct position immediately after being typed.

- Java, C: labels. Labels end with a colon and the colon is included in electric keys for these languages. With pressing the colon, the line is reindented and the labels are indented a level to the left.

- Basic: endif. Here f letter is an electric key, that makes the line indented to the left.

In jEdit 4 electric keys worked unconditionally. As of jEdit 5 they trigger reindentation only if the indentation of the line, before pressing a key, is the same as jEdit would indent it using its rules. This allows for specifying more electric keys in mode files, because they don't cause unwanted indentation like they did before. Electric keys including all letters seem to be good solution for basic-like languages.

Automatic Indent Scheme: simple

In this simplified automatic-indentation scheme, only two actions trigger an indentation: pressing ENTER, or causing a hard wrap. Only the new line will be indented, and the amount of indentation will be the same as the previously line.

Automatic Indent Scheme: none

In this automatic indentation scheme, no actions in the text area will trigger a reindentation, and all lines start completely unindented.

Further customization of automatic indentation

The behavior of the ENTER and TAB keys can be configured in the **Shortcuts** pane of the **Utilities**>**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. This is equivalent to **Insert Newline** when using the indentation scheme "none".

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. This is equivalent to **Insert Tab** when not using the "full" automatic indentation scheme.

- **Indent Selected Lines**. This binding will not respect the selected auto indentation scheme.

See the section called "The Shortcuts Pane" for details.

To insert a literal tab or newline without performing indentation, prefix the tab or newline with C+e v. For example, to create a new line without any indentation, type C+e v ENTER.

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: C+e C+c) encloses the selection with comment start and end strings, for example /* and */ in Java mode.

Edit>**Source Code**>**Line Comment** (shortcut: C+e C+k) inserts the line comment string, for example // in Java mode, at the start of each selected line.

Toggling Comments

You might be looking for the actions **Toggle Line Comment** or **Toggle Range Comment**. They are available from the TextTools plugin, not jEdit core.

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 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: `C+]`) or clicking the scope indicator in the gutter moves the caret to the matching bracket.

Edit>**Source**>**Select Code Block** (shortcut: `C+[`) 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: `C+e C+[`) moves the caret to the previous opening bracket.

Edit>**Source**>**Go to Next Bracket** (shortcut: `C+e C+]`) 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**>**Options** dialog box; see the section called "The Global Options Dialog Box".

> **Tip**
>
> 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.

Abbreviations

Abbreviations are invoked by typing a couple of letters and then issuing the **Edit**>**Expand Abbreviation** (keyboard shortcut: `C+;`), which takes the word before the caret as the abbreviation name. If that particular abbreviation was not yet set, a dialog will pop up, and you can enter the text to insert before and after the caret. After the abbreviation is created, it can be viewed or edited from the **Abbreviations** pane of the **Utilities**>**Options** dialog box; see the section called "The Abbreviations Pane".

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 `C+;`. An abbreviation can either be global, in which case it can be used in all edit modes, or specific to a single mode.

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.

Automatic abbreviation expansion can be enabled in the **Abbreviations** pane of the **Utilities**>**Options** dialog box. 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`.

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.

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 9, *Installing and Using Plugins* 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**> **Options** dialog box. See the section called "The Editing Pane".

• In the current buffer for the duration of the editing session in the **Utilities**>**Buffer Options** dialog box. See the section called "The 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.

Collapsing and Expanding Folds

The first line of each fold has a triangle drawn next to it in the gutter (see the section called "Interface Overview" 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 background color that depends on the fold level, 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** (shortcut: `A+BACK_SPACE`) collapses the fold containing the caret.

Folding>**Expand Fold One Level** (shortcut: `A+ENTER`) expands the fold containing the caret. Nested folds will remain collapsed, and the caret will be positioned on the first nested fold (if any).

Folding>**Expand Fold Fully** (shortcut: `AS+ENTER`) expands the fold containing the caret, also expanding any nested folds.

Folding>**Collapse All Folds** (shortcut: `C+e c`) collapses all folds in the buffer.

Folding>**Expand All Folds** (shortcut: `C+e x`) expands all folds in the buffer.

Navigating Around With Folds

Folding>**Go to Parent Fold** (shortcut: `C+e u`) moves the caret to the fold containing the one at the caret position.

Folding>**Go to Previous Fold** (shortcut: `A+UP`) moves the caret to the fold immediately before the caret position.

Folding>**Go to Next Fold** (shortcut: `A+DOWN`) moves the caret to the fold immediately after the caret position.

Miscellaneous Folding Commands

Folding>**Add Explicit Fold** (shortcut: `C+e a`) surrounds the selection with "{{{" and "}}}". If the current buffer's edit mode defines comment strings (see the section called "Commenting Out Code") the explicit fold markers will automatically be commented out as well.

Folding>**Select Fold** (shortcut: `C+e s`) selects all lines within the fold containing the caret. `Control`-clicking a fold expansion triangle in the gutter has the same effect.

Folding>**Expand Folds With Level** (shortcut: `C+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**> **Options** dialog box. See the section called "The Editing Pane".

- 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:
  ```

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.

Holding down `Alt` while clicking a fold expansion triangle in the gutter will hide all lines the buffer except those contained in the clicked fold.

Folding>**Narrow Buffer to Fold** (shortcut: `C+e n n`) hides all lines the buffer except those in the fold containing the caret.

Folding>**Narrow Buffer to Selection** (shortcut: `C+e n s`) hides all lines the buffer except those in the selection.

Folding>**Expand All Folds** (shortcut: `C+e x`) shows lines that were hidden as a result of narrowing.

Chapter 7. Customizing jEdit

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 the section called "Line Separators")

- The character encoding (see the section called "Character Encodings")

- If the file should be GZipped on disk (see the section called "Opening Files")

- Whether to show a dialog or auto-reload when this buffer's file is changed on disk.

- The edit mode (see the section called "Edit Modes")

- The fold mode (see the section called "Folding")

- The automatic indentation scheme (see the section called "Automatic Indent")

- The wrap mode and margin (see the section called "Wrapping Long Lines")

- The tab width (see the section called "Tabbing and Indentation")

- The indent width

- If soft tabs should be used (see the section called "Tabbing and Indentation")

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 the buffer is saved and loaded again.

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

Property name	Description
collapseFolds	Folds with a level of this or higher will be collapsed when the buffer is opened. If set to zero, all folds will be expanded initially. See the section called "Folding".

deepIndent	When set to "true", multiple-line expressions delimited by parentheses are aligned like so:
	
	`retVal.x = (int)(horizontalOffset`
	`    + Chunk.offsetToX(info.chunks,`
	`                        offset));`

Property name	Description

	With this setting disabled, the text would look like so:
	
	`retVal.x = (int)(horizontalOffset`
	`    + Chunk.offsetToX(info.chunks,`
	`      offset));`

`folding`	The fold mode; one of "none", "indent", "explicit", or the name of a plugin folding mode. See the section called "Folding".
`indentSize`	The width, in characters, of one indent. Must be an integer greater than 0. See the section called "Tabbing and Indentation".
`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 the section called "Inserting and Deleting Text".
`mode`	The default edit mode for the buffer. See the section called "Edit Modes".
`noTabs`	If set to "true", soft tabs (multiple space characters) will be used instead of "real" tabs. See the section called "Tabbing and Indentation".
`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 the section called "Tabbing and Indentation".
`wordBreakChars`	Characters, in addition to spaces and tabs, at which lines may be split when the word wrap mode is set to "hard". See the section called "Wrapping Long Lines".
`wrap`	The word wrap mode; one of "none", "soft", or "hard". See the section called "Wrapping Long Lines".
`autoIndent`	The automatic indentation scheme; one of "none", "full", or "simple". See the section called "Automatic Indent".

You may see `:encoding=XXX:` in a file as it is a buffer-local property and specifying the character encoding for the file. But it is not really a buffer-local property, and behaves differently. It is detected by `buffer-local-property` detector only if the detector is selected in encoding options. Thus, it works only at loading, and it must appear near the top of the file. See the section called "Character Encodings".

The Global Options Dialog Box

Utilities>**Options** displays the options dialog. It has 2 tabs, the first is **Global Options**. This tab contains several options panes, each containing a set of related options. Use the list on the left splitter to switch between panes. Only panes created by jEdit are described here; **Plugin Options** panes are created and documented by the plugins themselves.

The General Pane

The **General** pane contains various settings, such as the number of recent files to remember, when to check for changed files, if the recent file list should be sorted, what current locale to use, if caret positions or markers in buffers should be saved, if previously open files or split configurations should be restored on startup, and so on.

If Open Buffers Are Changed On Disk... .   If **Do Nothing** is selected, then modifications from jEdit will silently clobber changes made from other processes during saves. Don't use this option unless you know what you are doing! Also, changing this option here only affects newly opened buffers, not the ones that are currently open. You can also change this setting for individual buffers from Buffer Options. the section called "The Buffer Options Dialog Box"

Check for changed buffers upon... .   This option allows you choose additional times that jEdit checks for changed files on disk. For slow or remote file systems, removing unnecessary file status checks might improve performance. Regardless of the choice here, files are still checked before save, unless **Do Nothing** is also selected for the previous option.

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 labelled "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 the section called "Positional Parameters" for information about positional parameters in abbreviations.

The Appearance Pane

The **Appearance** pane can be used to change the appearance of user interface controls such as buttons, labels and menus. It can also be used to change the icon set, or look and feel, enable/disable the splash screen or system tray, and other appearance tweaks. You can also set the number of items retained in history text fields, see Appendix C, *History Text Fields*.

The Context Menu Pane

The **Context Menu** option pane edits the text area's right-click context menu. See the section called "Multiple Views".

The Docking Pane

The **Docking** option pane shows a list of available dockables, and allows you to specify docking locations for each of them. Another way to specify docking locations is to use the popup menus associated with each dockable window.

It is possible to configure jEdit to automatically load and/or save **Docking Layouts** (similar to eclipse perspectives) based on the edit mode of your current buffer through the checkboxes in this pane. See the section called "Window Docking Layouts".

jEdit also supports alternate docking frameworks. If the appropriate plugins are installed (Currently only MyDoggy is available), you can change docking frameworks from here.

The Editing Pane

The **Editing** option pane contains 3 tabs where settings such as the tab size, syntax highlighting and soft tabs on a global or mode-specific basis are made.

The **Mode Settings** tab allows adjusting specific settings per mode. Changing these options 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, "Writing Edit Modes". Some of these options can be further overridden on an individual file basis through the use of buffer-local properties.

The `File name glob` and `First line glob` text fields let you specify a glob pattern that paths and first lines of buffers will be matched against to determine the edit mode. See Appendix D, *Glob Patterns* for information about glob patterns.

The `Extra Word Characters` allows you to set the `noLineSep` buffer property on a mode-wide basis, allowing you to define what is considered part of a word when double-clicking on it in the text area.

The `Deep Indent` option instructs jEdit to indent subsequent lines so that they line up with the open bracket on the previous line.

The **Edit Modes** tab provides a setting to choose the default edit mode, the edit modes to display in the various 'mode' choosers, and the ability to manually add a new edit mode. Since there are now over 200 modes that jEdit recognizes, having the ability to reduce the number of choices in the 'mode' choosers to those needed by a user is a nice feature.

This tab also provides a way to easily add and delete user modes.

The **Undo Settings** tab allows setting the number of undo and whether to reset the undo history on save.

The Encodings Pane

This option pane offers users of jEdit many flexible options for defining how Encodings are handled in jEdit. See the section called "Character Encodings" for the basics.

The default line separator character (see the section called "Line Separators") can be set from here.

Use autodetection when possible is an option you can switch on or off.

The `List of Encoding Autodetector Names` can be used to control what encoding detections are used on each file when it is loaded. The order they appear in this list determines the order of detectors that are tried. There are some detectors which are available with jEdit core:

- `BOM`: detects Byte Order Mark.

- `XML-PI`: detects encoding declaration in XML Processing Instruction.

- `html`: detects charset description in HTML META element.

- `python`: detects various encoding declaration accepted in Python. This accepts encoding declarations for GNU Emacs or Bram Moolenaar's VIM.

- `buffer-local-property`: detects same syntax described at the section called "Buffer-Local Properties" for property name "encoding". Note that unlike other buffer-local properties, this one will not work unless it is at the top of the file, and this appears in the list of encoding detectors.

Others can be defined in plugins as services and added to this space-separated list. See EncodingDetector for details on how to offer additional encoding autodetector.

The `List of Fallback Encodings` is used when a file fails to open in the default encoding, and the Encoding Autodetectors also fail. The list order here determines the order of encodings that are tried. Each is separated by a space. This is especially handy when doing directory searches through files of different encodings. We suggest using `UTF-8` as either your default or one of the fallback encodings.

While jEdit allows you to edit files in a variety of different encodings, the average user switches between only 2 or 3. In other parts of jEdit, where the list of encodings is displayed in a combobox (such as the buffer options) or a menu (such as **File - Reload with Encoding** submenu) it may be desirable to display only a subset of available encodings, those that are in common local use. The

Encodings checkbox list allows the user to select the subset of supported encodings to display in other GUI components that list all of the encodings.

The Gutter Pane

The **Gutter** option pane contains settings to customize the appearance of the gutter. You can customize values such as "minimal number of digits to reserve for line numbers", and "fold style". See the section called "Interface Overview".

The Large Files Pane

This option pane allows the user to customize how jEdit should handle large files. There is a radio-button with 4 choices, allowing you to pop up a dialog asking the user each time, or automatically lower the level of syntax highlighting and disable soft-wrap to speed things things up (or use less battery).

The jEdit properties `longLineLimit` and `largeBufferSize` can be customized to change which buffers jEdit considers 'large'.

The Mouse Pane

The **Mouse** option pane contains settings for toggling drag and drop of text, as well as gutter mouse click behavior.

The only option that may not be self-explanatory is the **Double-Click drag joins non-alphanumeric characters**. This option means that double-click will select a region that includes the non-alphabetical characters, as defined for the current mode. The actual set of characters can be defined for an indiviual file via buffer-local properties (`noWordSep`) or on a mode-wide basis from the Editing option pane (`Extra Word Characters`).

The Plugin Manager Pane

The **Plugin Manager** pane contains a chooser for the desired download mirror, as well as various settings such as the directory where plugins are to be installed. In addition, you can set the time in minutes that the pluginlist can be cached from jedit.org, helping to reduce the server load. See Chapter 9, *Installing and Using Plugins*.

If the option **Disable Obsolete Plugins** is checked, then plugins that were released on Plugin Manager will be checked against the plugins you have installed, for those with a maximum jEdit version that is lower than the one you are running. Plugins are marked with a maximum jEdit version when they are found to be broken or somehow incompatible with a given jEdit release. Until an update is made available for such a plugin on Plugin Manager, these plugins are automatically unloaded and marked unsupported. This should improve the stability of jEdit.

If you re-enable a plugin that was disabled this way, it will remain loaded until the next time the plugin list is checked - whenever the user selects the **Update** or **Install** tab from Plugin Manager. If you un-check this option, then plugins will not be automatically disabled in this way.

The Printing Pane

As of jEdit 5.3, all printing options have been moved to the printer dialog. See the section called "Printing".

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 downloading plugins.

The Saving and Backup Pane

The **Saving and Backup** option pane contains settings for dirty and/or untitled buffers, confirmation dialog option after **Close All**, autosave, and backup features. See the section called "Autosave and Crash Recovery" and the section called "Backups" for information about those features.

The Shortcuts Pane

The **Shortcuts** option pane associates keyboard shortcuts with commands. Each command can have up to two shortcuts associated with it, and each shortcut can be a single or multiple key sequence.

jEdit 5 organizes shortcuts into **Keymaps**. Each *keymap* is a named set of keyboard shortcut mappings. Default keymaps are found in jEdit's `keymaps` folder, and user customized keymaps are are stored in the user settings' `keymaps` folder. The user can switch between keymaps with the first combobox on this pane.

The `imported.props` keymap is automatically created and selected when jEdit needs to initially create a `keymaps` user settings folder. At this point, jEdit imports the existing shortcuts and places them into `imported`. This makes it easy to bring in shortcuts from properties files that were customized with jEdit 4.5 or earlier.

If a keymap of the same name exists in the defaults and the user settings directory, the user version is the one that is used in favor of the default. To take an existing keymap and customize it, select it, click **duplicate** and you will be asked for the name of the new keymap. A copy of that keymap will be saved in the user settings `keymaps` directory. At this point, this keymap will be selected and will determine where new shortcut properties are stored. To remove all customizations and restore a default keymap, click **reset**.

The combo box below the **Keymap** selector selects the **Action Set** to edit. *Action Sets* exist for the set of all built-in commands, the commands of each plugin, and the set of macros. Some plugins (ProjectViewer, Console and SideKick) offer additional action sets of dynamically-generated actions. Here, you can also select **All** to see all actions, and an additional 4th column appears in the table, indicating the Action Set each action comes from.

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

The Status Bar Pane

The **Status Bar**, its API, and its corresponding option pane contains settings to customize which widgets are in the status bar, their order, and what separators exist between them. Also, you can disable it completely, for regular and/or plain views. See the section called "The Status Bar".

From the `Options` tab, you can customize information about the caret display in the lower left corner.

Selecting the `Widgets` tab of this option pane shows you what widgets on the right, and their order. You can add or remove widgets and separators/labels here.

The Syntax Highlighting Pane

The **Syntax Highlighting** pane can be used to customize the fonts and colors for syntax highlighting. See the section called "Syntax Highlighting".

The Text Area Pane

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

You can configure the **Text Font**, antialias settings, colors, cursor style, highlight matching, and word-completion settings from here.

Fractional Font Metrics is an old option that helps with certain versions of Java, but usually not in combination with subpixel antialiasing.

Additional Fonts with font substitution if checked, shows a list of **Preferred fonts**, as well as the following option. Fonts added to this list will determine the order jEdit searches for glyphs that may be missing from your chosen **Text Font**.

If the **Font Substitution: Search all system fonts** option is checked, *all of the installed fonts* are searched for glyphs, after the preferred list is searched. If this option is checked, no fonts need to be added to preferred fonts list. You probably don't want to un-check either of these options unless you want to test a system with limited fonts.

The Tool Bar Pane

The **Tool Bar** option pane lets you edit the tool bar, or disable it completely. See the section called "Multiple Views".

The View Pane

The **View** option pane lets you change various settings related to the editor main window appearance, including the arrangement of dockable windows, whether the search bar and buffer switcher are visible, and whether menu, toolbar, and statusbar are visible in full-screen mode. See the section called "Multiple Views".

You can choose the default *bufferset scope* here, as well as whether/how you want buffersets to be sorted in buffer switchers. See the section called "Buffer Sets and closing buffers" for more details.

If **Abbreviate paths with environment variables when possible** is checked, you will notice that jEdit displays abbreviated versions of file paths when it can, using `$VARIABLE/name.ext` or `%VARIABLE%\name.ext` syntax, depending on your platform. Abbreviating is used in the File System Browser, as well as in the window title, and in plugins, to save horizontal space. Reverse-expansions also work as you would expect them to, with both syntaxes recognized on all platforms.

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 the section called "The File System Browser (FSB)" for more information.

The jEdit Settings Directory

jEdit stores settings, keymaps, 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 [1]. It is printed to the activity log (**Utilities**>**Troubleshooting**>**Activity Log**). For example:

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

Another way to find the location of your settings directory is to use the "Utilities" menu, then the "Settings Directory" menu item. The first item in the pullout menu is the location of your settings directory.

---

[1] On Linux, it is `~/.jedit`. On Windows, you will find it in `%APPDATA%\jEdit`. On the Mac, it is `~/Library/jEdit`.

From Console or the File System Browser, you can use an environment variable, $JEDIT_SETTINGS, to refer to that location.

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.

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 the section called "Abbreviations".

- activity.log - a plain text file which contains the full activity log. See Appendix B, *The Activity Log*.

- history - a plain text file which stores history lists, used by history text fields and the **Edit**>**Paste Previous** command. See the section called "Transferring Text" and Appendix C, *History Text Fields*.

- jars - this directory contains plugins. See Chapter 9, *Installing and Using Plugins*.

- jars-cache - this directory contains plugin cache files which decrease the time to start jEdit. They are automatically updated when plugins are installed or updated.

- keymaps - this directory contains collections of named keyboard shortcuts which can be defined from the Shortcuts Option Pane (see the section called "The Shortcuts Pane").

- killring.xml - stores recently deleted text. See the section called "Transferring Text".

- macros - this directory contains macros. See Chapter 8, *Using Macros*.

- modes - this directory contains custom edit modes. See Part II, "Writing Edit Modes".

- perspective.xml - an XML file that stores the list of open buffers and views used to maintain editor state between sessions.

- 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 9, *Installing and Using Plugins*.

- pluginMgr-Cached.xml.gz - this contains a cached copy of the last XML plugin list downloaded from plugin central. If you delete this file, a new one will be created next time you try to install a plugin via Plugin Manager.

- printspec - a binary file that stores printing settings.

- properties - a plain text file that stores the majority of jEdit's and its plugins settings. For more information see the section called "The jEdit properties file".

- 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.

- registers.xml - an XML file that stores register contents. See the section called "General Register Commands" for more information about registers.

- 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 2, *Starting jEdit*.

- `settings-backup` - this directory contains numbered backups of all automatically-written settings files.

- `startup` - This directory contains startup scripts in beanshell or other plugin-supported scripting languages. They are run at the time jEdit starts, after the `startup` scripts in the jEdit install directory have been run. See the section called "Startup Scripts"

The jEdit properties file

The jEdit `properties` file uses the Java properties syntax to store key/value pairs. All of the values are stored as strings, but are interpreted as other types (such as integer or boolean) by plugins at runtime.

Do not edit this file while jEdit is running. If you do, it is possible that your changes (either your edits, or jEdit settings changes) may get lost.

Site Properties

You may also put properties files in the `properties` directory under the jEdit home directory (NOT the jedit settings directory). You can locate the jEdit home directory by going to the Utilities menu directory, then the **jEdit Home Directory** menu item, and the first item in the pullout menu will be the location of the jEdit home directory. This is intended for site-wide settings. This lets you keep custom user properties separate from the jEdit site-wide properties, so they are easier to find, edit, and move between machines. Note that your custom properties files must have ".props" as the file name extension.

Prior to jEdit 5.0, this was also where site-wide keyboard shortcuts were placed, but now you can define custom sets of shortcuts as keymap files. These files can be cloned and edited from the Shortcuts Option Pane, or edited by hand. To place them in a system-wide location, copy them into the `keymaps` folder under the jedit home directory.

Site properties files are read in alphabetically by file name. This means that if you have a property with the same name in more than one file, the value for that property will be the value found in the last file that was read.

You can edit these files inside jEdit - changes made to these files will not be re-read until the next time jEdit is started.

Chapter 8. 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, "Writing Macros".

> **Other scripting languages**
>
> A number of jEdit plugins provide support for writing scripts in alternative programming languages, like Python and Prolog. Consult the documentation for the appropriate plugins for more information.

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: `C+m C+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: `C+m C+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 the section called "How jEdit Organizes Macros" 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: `C+m C+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: `C+m C+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.

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, *Macros Included With jEdit*.

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**>**Options** dialog; see the section called "The Global Options Dialog Box".

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 the section called "The jEdit Settings Directory" 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 9. 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, "Writing Plugins".

The Plugin Manager

Plugins>**Plugin Manager** displays the plugin manager window. It consists of three tabs: Manage, Update and Install. The Manage tab 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**. This will display a confirmation dialog box first.

To view plugin documentation, select a plugin and click **Help**. Note that plugin documentation can also be accessed by invoking **Help**>**jEdit Help**.

After you have tuned jEdit to your liking and want to install the same set of plugins onto another host, or another user's profile, you can export your currently installed plugin list as an xml file, known as a **PluginSet**. The **Save** rollover button allows you to save the list of installed and loaded plugins to an XML file. See the section called "Plugin Sets" for more information.

Plugins>**Plugin Options** displays a dialog box for changing plugin settings.

Installing and Updating 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** tab 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.

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 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.

If a previously saved PluginSet was selected, it will automatically be loaded whenever the Install tab is created, and you will see the filename in the hovertip of the **choose** rolloverbutton, as well as all of the plugins in that set already checked for you.

You can clear the active PluginSet with the **clear** button next to it, or choose a different PluginSet xml file with the **choose** button. See the section called "Plugin Sets" for more information.

Once you have specified plugins to install, click **Install** to begin the download process.

By default, the plugin manager does not download plugin source code, and installs the downloaded plugins in the `jars` subdirectory of the user-specific settings directory. These settings can be changed in **Plugin Manager** pane of the **Utilities**>**Options** dialog box; see the section called "The Plugin Manager Pane".

The **Update** tab of the plugin manager is very similar to the **Install** tab. It lists plugins for which updated versions are available. It will also offer to delete any obsolete plugins.

> **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**>**Options** dialog box; see the section called "The Proxy Servers Pane".

Plugin Sets

A **PluginSet** is a collection of plugins, represented as an XML file. These XML files can be created from the **save** button of the Manage tab of the Plugin Manager. Saving a PluginSet remembers all of the currently loaded plugins.

When a PluginSet has been saved, it becomes the "default pluginset", which means that if you unload/uninstall plugins from that set and go back to the Install tab, you should see them selected for download again. To clear this setting, click on the **clear** button in the Install tab.

It is posisble to Choose/Open a PluginSet from the Manage or the Install tab. The behavior of choosing a PluginSet depends on which tab you are on when you choose it. From the Manage tab, it unloads plugins that are loaded but not in the list. From the Install tab, it selects plugins from that list that are not loaded, marking them for download from Plugin Central.

When choosing a PluginSet, the path can be given as a remote URL. This helps teachers and sysadmins direct the students/slaves to a standard set of plugins that are required for the course/job.

Appendix A. Keyboard Shortcuts

This appendix documents the "jEdit" keymap of keyboard shortcuts. Keymaps can be created and customized to suit your taste in the **Shortcuts** pane of the **Utilities**>**Options** dialog box; see the section called "The Shortcuts Pane".

Files

For details, see the section called "Switching Buffers", the section called "Multiple Views" and Chapter 4, *Working With Files*.

`C+n`	New file.
`C+o`	Open file.
`C+w`	Close buffer.
`C+e C+w`	Close all buffers.
`C+s`	Save buffer.
`C+e C+s`	Save all buffers.
`C+p`	Print buffer.
`C+PAGE_UP`	Go to previous buffer.
`C+PAGE_DOWN`	Go to next buffer.
`C+\``	Go to recent buffer.
`A+\``	Show buffer switcher.
`C+q`	Exit jEdit.

Views

For details, see the section called "Multiple Views".

`C+e C+t`	Turn gutter (line numbering) on and off.
`C+0`	Remove split containing current text area only.
`C+1`	Remove all splits.
`C+2`	Split view horizontally.
`C+3`	Split view vertically.
`A+PAGE_UP`	Send keyboard focus to previous text area.
`A+PAGE_DOWN`	Send keyboard focus to next text area.
`C+e UP; LEFT; DOWN; RIGHT`	Send keyboard focus to top; bottom; left; right docking area.
`C+e C+\``	Close currently focused docking area.

Action Bar

For details, see the section called "The Action Bar".

`C+ENTER`	Display the action bar and give it keyboard focus.
`C+SPACE`	Repeat last editor action.

Moving the Caret

For details, see the section called "Moving The Caret", the section called "Working With Words", the section called "Working With Lines", the section called "Working With Paragraphs" and the section called "Bracket Matching".

`Arrow`	Move caret one character or line.
`C+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).
`C+HOME`	Beginning of buffer.
`C+END`	End of buffer.
`C+]`	Go to matching bracket.
`C+e [; ]`	Go to previous; next bracket.
`C+l`	Go to line.

Selecting Text

For details, see the section called "Selecting Text", the section called "Working With Words", the section called "Working With Lines", the section called "Working With Paragraphs" and the section called "Bracket Matching".

`S+Arrow`	Extend selection by one character or line.
`CS+Arrow`	Extend selection by one word or paragraph.
`S+PAGE_UP`; `S+PAGE_DOWN`	Extend selection by one screenful.
`S+HOME`	Extend selection to first non-whitespace character of line, beginning of line, first visible line (repeated presses).
`S+END`	Extend selection to last non-whitespace character of line, end of line, last visible line (repeated presses).
`CS+HOME`	Extend selection to beginning of buffer.
`CS+END`	Extend selection to end of buffer.
`C+[`	Select code block.
`C+e w; l; p`	Select word; line; paragraph.
`C+e C+l`	Select line range.
`C+a`	Select all.
`ESCAPE`	Select none.
`A+\`	Switch between range and rectangular selection mode.
`C+\`	Switch between single and multiple selection mode.
`C+e i`	Invert selection.

Scrolling

For details, see the section called "Multiple Views".

`C+e C+j`	Ensure current line is visible, and send focus to the text area.
`C+e C+n`	Center caret on screen.
`C+'; C+/`	Scroll up; down one line.
`A+'; A+/`	Scroll up; down one page.

Text Editing

For details, see the section called "Undo and Redo", the section called "Inserting and Deleting Text", the section called "Working With Words", the section called "Working With Lines" and the section called "Working With Paragraphs".

`C+z`	Undo.
`C+e C+z`	Redo.
`BACK_SPACE; DELETE`	Delete character before; after caret.
`C+BACK_SPACE; C+DELETE`	Delete word before; after caret.
`C+d; C+e d`	Delete line; paragraph.
`CS+BACK_SPACE; CS+DELETE`	Delete from caret to beginning; end of line.
`C+e r`	Remove trailing whitespace from the current line (or all selected lines).
`C+j`	Join lines.
`C+b`	Complete word.
`C+e f`	Format paragraph (or selection).

Clipboard and Registers

For details, see the section called "Transferring Text".

`C+x` or `S+DELETE`	Cut selected text to clipboard.
`C+c` or `C+INSERT`	Copy selected text to clipboard.
`C+e C+u`	Append selected text to clipboard, removing it from the buffer.
`C+e C+a`	Append selected text to clipboard, leaving it in the buffer.
`C+v` or `S+INSERT`	Paste clipboard contents.
`C+e C+p`	Vertically paste clipboard contents.
`C+r C+x` *key*	Cut selected text to register *key*.
`C+r C+c` *key*	Copy selected text to register *key*.
`C+r C+u` *key*	Append selected text to register *key*, removing it from the buffer.
`C+r C+a` *key*	Append selected text to register *key*, leaving it in the buffer.
`C+r C+v` *key*	Paste contents of register *key*.
`C+r C+p` *key*	Vertically paste contents of register *key*.
`C+e C+v`	Paste previous.
`C+e C+y`	Paste deleted.

Markers

For details, see the section called "Markers".

`C+e C+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.
`C+t key`	Add marker with shortcut *key*.
`C+y key`	Go to marker with shortcut *key*.
`C+u key`	Select to marker with shortcut *key*.
`C+k key`	Go to marker with shortcut *key*, and move the marker to the previous caret position.
`C+e C+,;C+e C+.`	Move caret to previous; next marker.

Search and Replace

For details, see the section called "Search and Replace".

`C+f`	Open search and replace dialog box.
`C+g`	Find next.
`C+h`	Find previous.
`C+e C+b`	Search in open buffers.
`C+e C+d`	Search in directory.
`C+e C+r`	Replace in selection.
`C+e C+g`	Replace in selection and find next.
`C+,`	Incremental search bar.
`A+,`	HyperSearch bar.
`C+.`	Incremental search for word under the caret.
`A+.`	HyperSearch for word under the caret.
`C+e C+i`	Toggle ignore case.
`C+e C+x`	Toggle regular expressions.

Source Code Editing

For details, see the section called "Abbreviations", the section called "Tabbing and Indentation" and the section called "Commenting Out Code".

`C+;`	Expand abbreviation.
`A+LEFT; A+RIGHT`	Shift current line (or all selected lines) left; right.
`S+TAB; TAB`	Shift selected lines left; right. Note that pressing `TAB` with no selection active will insert a tab character at the caret position.
`C+i`	Indent current line (or all selected lines).
`C+e C+c`	Range comment selection.
`C+e C+k`	Line comment selection.

Folding and Narrowing

For details, see the section called "Folding" and the section called "Narrowing".

`A+BACK_SPACE`	Collapse fold containing caret.
`A+ENTER`	Expand fold containing caret one level only.

`AS+ENTER`	Expand fold containing caret fully.
`C+e x`	Expand all folds.
`C+e a`	Add explicit fold.
`C+e s`	Select fold.
`C+e ENTER` *key*	Expand folds with level less than *key*, collapse all others.
`C+e n n`	Narrow to fold.
`C+e n s`	Narrow to selection.
`A+UP`; `A+DOWN`	Moves caret to previous; next fold.
`C+e u`	Moves caret to the parent fold of the one containing the caret.

Macros

For details, see Chapter 8, *Using Macros*.

`C+m C+r`	Record macro.
`C+m C+m`	Record temporary macro.
`C+m C+s`	Stop recording.
`C+m C+p`	Run temporary macro.

Alternative Shortcuts

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

`A+j`; `A+l`	Move caret to previous, next character.
`A+i`; `A+k`	Move caret up, down one line.
`A+q`; `A+a`	Move caret up, down one screenful.
`A+z`	First non-whitespace character of line, beginning of line, first visible line (repeated presses).
`A+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 activity log is shown in a floating window. This window can be docked using the commands in its top-left corner popup menu; see the section called "Window Docking Layouts".

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.

The **Settings** button in the Activity Log window shows a dialog that lets you adjust the output colors, filter the messages by type, and lets you set the maximum number of lines to display. Note that larger numbers will decrease the overall performance of jEdit since these lines are kept in memory.

In the **Settings** pane there is also a debugging option, **Beep on output**. It allows for catching problems right after they show up. Each error message entry is accompanied by a system beep. Lower priority entries may be alerted this way, if the `log` option is used, see the section called "Command Line Usage".

Appendix C. History Text Fields

The text fields in many jEdit components, such as the file system browser, incremental search bar, and action bar, all remember the last 20 entered strings by default. The number of strings to remember can be changed in the **Appearance** pane of the **Utilities**>**Options** dialog box; see the section called "The Appearance Pane".

Pressing UP recalls previous strings. Pressing DOWN after recalling previous strings recalls later strings.

Pressing S+UP or S+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. Selecting the first item, "previously entered strings:" pops up a dialog that lets you change previously entered strings. Holding down Shift while clicking will display a menu of all previously entered strings that begin with the text already entered.

Search and Replace fields

In jEdit 4.3, the search/replace history fields are multi-line textareas, so they no longer use the same single-line history textfield described above. The multiline history textarea behaves a little differently: UP and DOWN arrows go up and down a line in the textarea, instead of through the previously entered strings. PageUp and PageDown are used instead to select history strings, and there is no arrow combo button, although right-click will still show you the history as a context menu.

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

Since we use `java.util.regex` patterns to implement globs, this means that in addition to the above, a number of "character class metacharacters" may be used. Keep in mind, their usefulness is limited since the regex quantifier metacharacters (asterisk, questionmark, and curly brackets) are redefined to mean something else in filename glob language, and the regex quantifiers are not available in glob language.

- `\w` matches any alphanumeric character or underscore

- `\s` matches a space or horizontal tab

- `\S` matches a printable non-whitespace.

- `\d` matches a decimal 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".

- `*.{c,cpp,h,hpp,cxx,hxx}` - all C or C++ files.

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

Using regexes instead of globs

Sometimes it is desirable to use a regular expression instead of a glob for specifying file sets. This is because regular expressions are more powerful than globs and can provide the user with more specific filename matching criteria. To avoid the glob-to-regex transformation, prefix your pattern with the string `(re)`, which will tell jEdit to not translate the following pattern into a regex (since it already is one). For example:

`(re).*\.(h|c(c|pp)?)`   Matches *.c, *.cpp, *.h, *.cc

If you need to match files that begin with the glob-translate-disable prefix `(re)`, you can escape it with a leading backslash and the metacharacters will be translated into globs as before.

Appendix E. Regular Expressions

jEdit uses regular expressions from java.util.regex.Pattern to implement inexact search and replace. Click there to see a complete reference guide to all supported meta-characters.

A regular expression consists of a string where some characters are given special meaning with regard to pattern matching.

Inside XML files

Inside XML files (such as jEdit mode files), it is important that you escape XML special characters, such as &, <, >, etc. You can use the XML plugin's "characters to entities" to perform this mapping.

Inside Java / beanshell / properties files

Java strings are always parsed by java before they are processed by the regular expression engine, so you must make sure that backslashes are escaped by an extra backslash (\\)

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 non-word break

- \b matches at a word boundary

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

- \xNN matches hexadecimal character code NN

- \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

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.

On regex search

There are some known issues with the `java.util.regex` library, as it stands in Java. In particular, it is possible to create regular expressions that hang the JVM, or cause stack overflow errors, which was not as easy to accomplish using the legacy `gnu.regexp` library. If you find that `gnu.regexp`, used in jEdit 4.2 and earlier, is more suitable for your search/replace needs, you can try the **XSearch plugin**, which still uses it and can provide a replacement to the built-in search dialog.

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.

C/C++ macros

These macros are useful for C/C++ programming.

- `Include_Guard.bsh`

  Inserts conditional preprocessor directives around a header file, to prevent it from being included multiple times.

  The name of the generated preprocessor macro is based on the buffer's name.

- `Toggle_Header_Source`

  Toggles between the header and the implementation file. Works for .c, .cxx, and .cpp extensions.

Clipboard Macros

These macros copy or cut text to the clipboard.

- `Copy_Lines_Containing.bsh`

  Copies all lines from the current buffer, containing a user-supplied string, to the clipboard.

- `Cut_Lines_Containing.bsh`

  Cuts all lines from the current buffer, containing a user-supplied string, to the clipboard.

- `Copy_Selection_or_Line.bsh`

  If no text is selected, the current line is copied to the clipboard, otherwise the selected text is copied to the clipboard. Some editors have this has the default copy behavior. To achieve the same effect in jEdit, bind this macro to `C+c` in the **Shortcuts** pane of the **Utilities**> **Options** dialog box.

- `Cut_Selection_or_Line.bsh`

  If no text is selected, the current line is cut to the clipboard, otherwise the selected text is cut to the clipboard. Some editors have this has the default cut behavior. To achieve the same effect in jEdit, bind this macro to `C+x` in the **Shortcuts** pane of the **Utilities**> **Options** dialog box.

- `Copy_Visible_Lines.bsh`

  Copies the visible lines from the current buffer to the Clipboard. Lines that are not visible becuase they are folded are not copied.

- `Paste_Indent.bsh`

Pastes the content of the clipboard and indents it.

Editing Macros

These macros automate various text editing tasks.

- `Duplicate_Lines_Above.bsh`

  Duplicates current/selected line(s) upward.

- `Duplicate_Lines_Below.bsh`

  Duplicates current/selected line(s) downward.

- `Emacs_Ctrl-K.bsh`

  Cuts and appends text, from the cursor to the end of the line, into the copy buffer.

- `Emacs_Next_Line.bsh`

  Moves the cursor to the next line, centering the current line in the middle of the text area if the cursor is at the bottom of the text area.

- `Emacs_Previous_Line.bsh`

  Moves the cursor to the previous line, centering the current line in the middle of the text area if the cursor is at the top of the text area.

- `Go_to_Column.bsh`

  Prompts the user for a column position on the current line, then moves the caret there.

- `Greedy_Backspace.bsh`

  If buffer is using soft tabs, this macro will backspace to the previous tab stop, if all characters between the caret and the tab stop are spaces. In all other cases a single character is removed.

- `Greedy_Delete.bsh`

  If a buffer is using soft tabs, this macro will delete tabSize number of spaces, if all the characters between the caret and the next tab stop are spaces. In all other cases a single character is deleted.

- `Greedy_Left.bsh`

  If a buffer is using soft tabs, this macro will move the caret tabSize spaces to the left, if all the characters between the caret and the previous tab stop are all spaces. In all other cases, the caret is moved a single character to the left.

- `Greedy_Right.bsh`

  If a buffer is using soft tabs, this macro will move the caret tabSize spaces to the right, if all the characters between the caret and the next tab stop are all spaces. In all other cases, the caret is moved a single character to the right.

- `Keywords_to_Upper_Case.bsh`

  Converts all keywords in the current buffer to upper case.

- `Mode_Switcher.bsh`

  Displays a modal dialog with the current buffer's mode in a text field, allowing one to change the mode by typing in its name.

ENTER selects the current mode; if the text is not a valid mode, the dialog still dismisses, but a warning is logged to the activity log. ESACPE closes the dialog with no further action. TAB attempts to auto-complete the mode name. Pressing TAB repeatedly cycles through the possible completions. SHIFT-TAB cycles through the completions in reverse.

- Move_Line_Down.bsh

  Moves the current line down one, with automatic indentation.

- Move_Line_Up.bsh

  Moves the current line up one, with automatic indentation.

- Open_Line_Above.bsh

  Adds a new blank line before the current/selected line(s).

- Open_Line_Below.bsh

  Adds a new blank line after the current/selected line(s).

- Toggle_Fold.bsh

  Toggles visibility of current fold.

  This is especially useful for fold toggling via keyboard.

File Management Macros

These macros automate the opening and closing of files.

- Browse_Buffer_Directory.bsh

  Opens a the current buffer's directory in the file system browser.

- Browse_Directory.bsh

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

- Buffer_Switcher.bsh

  Displays a modal dialog listing all open buffers, allowing one to switch to and/or close buffers. ENTER switches to a buffer and closes the dialog, DELETE closes a buffer, SPACE switches to a buffer but does not close the dialog.

- Close_All_Except_Active.bsh

  Closes all files except the current buffer.

  Prompts the user to save any buffer containing unsaved changes.

- Copy_Path_to_Clipboad.bsh

  Copies the current buffer's path to the clipboard.

- Copy_Name_to_Clipboad.bsh

  Copies the current buffer's filename to the clipboard.

- Duplicate_Buffer.bsh

  Duplicates the current buffer into a new one.

- `Delete_Current.bsh`

  Deletes the current buffer's file on disk, but doesn't close the buffer.

- `Glob_Close.bsh`

  Closes all open buffers matching a given glob pattern.

- `Insert_Selection.bsh`

  Assumes the current selection is file path and tries replaces the selection with the contents of the file. Does nothing if no text is selected or the selection spans multiple lines.

- `Next_Dirty_Buffer.bsh`

  Switches to the next dirty buffer, if there is one.

- `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. Current VFS browser directory is also tried as a parent of the filename, but only as a local path.

- `Open_Selection_In_Desktop.bsh`

  Opens the file named by the current buffer's selected text using `Desktop`. That is opens the file using operating system's default application. If a link is selected, it is browsed instead, using default web browser. If no selection is active, the path under caret is used.

- `Send_Buffer_To_Next_Split.bsh`

  If bufferset scope is set to EditPane, the current buffer is added to the next Editpane's bufferset.

- `Toggle_ReadOnly.bsh`

  Toggles a local file's read-only flag. Uses platform-specific commands, so it only works on Windows, Unix and MacOS X.

User Interface Macros

Description.

- `Decrease_Font_Size.bsh`

  Decreases the font size in the gutter and text area by 1 point.

- `Increase_Font_Size.bsh`

  Increases the font size in the gutter and text area by 1 point.

- `Open_Context_Menu.bsh`

  Opens the text area context menu just below and to the right of the caret.

- `Reset_TextArea.bsh`

  Performs a split and an unsplit of the current TextArea. Useful for those occasions when your textarea is corrupt (painting the incorrect characters on the screen).

- `Splitpane_Grow.bsh`

When inside a split EditPane, this macro moves the splitter away from the cursor, effectively increasing the size of the currently active split pane.

- `Toggle_Bottom_Docking_Area.bsh`

    Expands or collapses the bottom docking area, depending on it's current state.

- `Toggle_Left_Docking_Area.bsh`

    Expands or collapses the left docking area, depending on it's current state.

- `Toggle_Right_Docking_Area.bsh`

    Expands or collapses the right docking area, depending on it's current state.

- `Toggle_Top_Docking_Area.bsh`

    Expands or collapses the top docking area, depending on it's current state.

Java Code Macros

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

- `Create_Constructor.bsh`

    Inserts constructor for the class at the current caret position.

- `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.

- `Java_File_Save.bsh`

Acts as a wrapper script to the Save As action. If the buffer is a new file, it scans the first 250 lines for a Java class or interface declaration. On finding one, it extracts the appropriate filename to be used in the Save As dialog.

- `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`

  Create and display API documentation for the current buffer.

  The macro includes various configuration variables you can change; see the comment at the beginning of the macro source for details.

Miscellaneous Macros

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

- `Buffer_to_HyperSearch_Results.bsh`

  Reads HyperSearch results from a buffer that was previously created by the HyperSearch_Results_to_Buffer macro and possibly filtered manually, and imports them into the HyperSearch Results dockable.

- `Debug_BufferSets.bsh`

  Display int and hex values for the character at the caret, in the status bar.

- `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_Actions.bsh`

  Displays a list of all the actions known to jEdit categorised by their action set.

  This macro can be a useful reference if you want to use the jEdit 4.2 action bar.

- `Display_Character_Code.bsh`

  Display int and hex values for the character at the caret, in the status bar.

- `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.

- `HyperSearch_Results_to_Buffer.bsh`

  Writes HyperSeach results to a new buffer.

  This buffer can be re-imported to the HyperSearch Results dockable by the Buffer_to_HyperSearch_Results macro.

- `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.

Property Macros

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

- `Create_Plugin_Announcement.bsh`

  Creates an announcement for the Plugin Central Submission Tracker based on the plugins *.props and description.html files.

- `Insert_Buffer_Properties.bsh`

  Inserts buffer-local properties into the current buffer.

  If the buffer's mode has a line comment defined, or comment start and end defined, the inserted properties will be commented out.

- `jEdit_Properties.bsh`

  Writes an unsorted list of jEdit 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.

- `System_Properties.bsh`

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

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 "#".

- `Compose_Tag.bsh`

  The selection is taken as tag name and replaced with a full xml tag.

- `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 an input dialog. The tags will surround any selected text.

- `Line_Filter.bsh`

  Filters lines of the current buffer due to a provided regular expression. The resulting set of lines can be either removed from the buffer or written to a new buffer.

  The filter works on a multiline selection (if there is one) otherwise on the whole buffer. The resulting set of lines includes those lines that either match or not match the regular expression.

- `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 the section called "Using a Single Keypress as Input".

- `Reverse_Lines.bsh`

  Reverses the selected lines or the entire buffer if no lines are selected. Does not support Rectangular Selections.

- `Single_Space_Buffer.bsh`

  Removes every second line, if they are all blank.

Part 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 the section called "Edit Modes".

This part of the user's guide was written by Slava Pestov and is maintained by the jEdit core development team.

Chapter 10. 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** > **Troubleshooting** > **Reload Edit Modes** command is invoked.

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.

Validation and Errors

Most XML file formats have a formal grammar specified in either DTD, XSD or RNG. In the example above, we can see that the `DOCTYPE`, or formal grammar for jEdit mode files is described in `xmode.dtd`, which happens to come from jEdit's source code. If you install the XML plugin, and while editing a mode file in jEdit, go to **Plugins - XML - Parse as XML**, you should see a structure tree in Sidekick, and you will also see errors (if there are any) in ErrorList, if the document does not conform to the proper XML syntax or the document's formal grammar. In addition, the XML plugin provides completion tips for elements and attributes. All of these things can help immensely especially when learning XML.

It is highly recommended that you check your XML files for validation errors before submitting them to the community.

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 within "&" and ";". The most frequently used entities are:

- `<` - The less-than (<) character

- `>` - 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.

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>
```

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 the section called "Buffer-Local Properties" may be given values in edit modes.

`contextInsensitive` - If true, the property indicates that a line can always be highlighted without taking care of the previous line. If activated, the syntax parsing will be much faster.

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.

  There are corresponding `unindentThisLine` and `unindentNextLines` properties which are checked also, for doing the reverse-indent operation on lines that match certain regular expressions.

  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);
    }
```

- `electricKeys`: characters listed here, when typed on a line, cause the current line to be re-indented. Notice that by default, pressing "Enter" does not re-indent the current line, only the new line. To get this behavior, add the newline character to `electricKeys` in the xml-escaped form

- `ignoreWhitespace`: Ignore whitespace lines. This property is on (`true`) by default. Python language sets this to `false` because of the special treatment of whitespaces. Note this example:

```
def fun1:
    a = 1
    b = 2

def fun2:
```

Pressing `C+i` (**Indent Lines** command) on the `fun2` line would usually indent this line and make it even with the `b = 2` line. But with switched off `ignoreWhitespace` setting the line will stay the way it was indented manually. `ignoreWhitespace=false` setting prevents any forward indentation after a whitespace line.

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="unalignedOpenBrackets" VALUE="(" />
    <PROPERTY NAME="unalignedCloseBrackets" VALUE=")" />
    <PROPERTY NAME="indentNextLine"
     VALUE="\s*(((if|while)\s*\(|else\s*|else\s+if\s*\(|for\s*\(.*\))[^{;]*)" /:
    <PROPERTY NAME="unindentThisLine"
     VALUE="^.*(default:\s*|case.*:.*)$" />
    <PROPERTY NAME="electricKeys" VALUE=":" />
    <!-- set this to 'true' if you want to use GNU coding style -->
    <PROPERTY NAME="doubleBracketIndent" VALUE="false" />
    <PROPERTY NAME="lineUpClosingBracket" VALUE="true" />
</PROPS>
```

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 the section called "The SPAN Tag".

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`.

- `ESCAPE` - specifies a character sequence for escaping literals. The first character following the escape sequence is not considered as input for syntax highlighting, thus being highlighted with default token for the rule set.

- `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 the section called "Token Types" 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>
```

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). Note that newlines have been inserted here for readability.

```
DIGIT_RE="(0[lL]?|[1-9]\d{0,9}(\d{0,9}[lL])?
|0[xX]\p{XDigit}{1,8}(\p{XDigit}{0,8}[lL])?
|0[0-7]{1,11}([0-7]{0,11}[lL])?|([0-9]+\.[0-9]*
|\.[0-9]+)([eE][+-]?[0-9]+)?[fFdD]?|[0-9]+([eE][+-]?[0-9]+[fFdD]?
|([eE][+-]?[0-9]+)?[fFdD]))"
```

Regular expression syntax is described in Appendix E, *Regular Expressions*.

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.

Per-Ruleset Properties

The PROPS tag (described in the section called "The PROPS Tag") 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 the section called "Commenting Out Code".

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.

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 the section called "Token Types" 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".

- `MATCH_TYPE` - Controls how the start and end of the sequence will be highlighted. See the section called "The MATCH_TYPE Attribute" for more information.

- `ESCAPE` - specifies a character sequence for escaping characters. The first character following the escape sequence is not considered as input for syntax highlighting, thus being highlighted with rule's token.

- `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.

Note that the `AT_LINE_START`, `AT_WHITESPACE_END` and `AT_WORD_START` attributes can also be used on the `END` element.

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>
```

The SPAN_REGEXP Tag

The `SPAN_REGEXP` rule is similar to the `SPAN` rule except the start sequence and optionally the end sequence are taken to be regular expressions. In addition to the attributes supported by the `SPAN` tag, the following attributes are supported:

- `HASH_CHAR` - a literal string which must be at the start of a regular expression.

- `HASH_CHARS` - a list of possible literal characters, one of which must match at the start of the regular expression.

`HASH_CHAR` and `HASH_CHARS` attributes are both optional, but you may only specify one, not both. If both are specified, `HASH_CHARS` is ignored and an error is shown. Whenever possible, use a literal prefix to specify a `SPAN_REGEXP`. If the starting prefix is always the same, use `HASH_CHAR` and provide as much prefix as possible. Only in rare cases would you omit both attributes, such as the case where there is no other reliable way to get the highlighting you need, for example, with comments in the Cobol programming language.

In addition, the `END` subtag supports the attribute `REGEXP`, which if set to `TRUE`, tells the highlighter to interpret the `END` text as a regular expression as well.

The regular expression match cannot span more than one line. Any text matched by groups in the BEGIN regular expression is substituted in the END string. See below for an example of where this is useful.

Regular expression syntax is described in Appendix E, *Regular Expressions*.

Here is a SPAN_REGEXP rule that highlights "read-ins" in shell scripts:

```
<SPAN_REGEXP HASH_CHAR="&lt;" TYPE="LITERAL1" DELEGATE="LITERAL">
    <BEGIN><![CDATA[<<[\p{Space}'"]*([\p{Alnum}_]+)[\p{Space}'"]*]]></BEGIN>
    <END>$1</END>
</SPAN_REGEXP>
```

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\b</BEGIN>
    <END>&gt;</END>
</SPAN_REGEXP>
```

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 the section called "Token Types" 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".

- MATCH_TYPE - Controls how the start of the sequence will be highlighted. See the section called "The MATCH_TYPE Attribute" for more information.

Here is an EOL_SPAN that highlights C++ comments:

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

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 following attributes are supported:

- HASH_CHAR - a literal string which must be at the start of a regular expression.

- `HASH_CHARS` - a list of possible literal characters, one of which must match at the start of the regular expression.

`HASH_CHAR` and `HASH_CHARS` attributes are both optional, but you may only specify one, not both. If both are specified, `HASH_CHARS` is ignored and an error is shown. Whenever possible, use a literal prefix to specify a `EOL_SPAN_REGEXP`. If the starting prefix is always the same, use `HASH_CHAR` and provide as much prefix as possible. Only in rare cases would you omit both attributes, such as the case where there is no other reliable way to get the highlighting you need, for example, with comments in the Cobol programming language.

The regular expression match cannot span more than one line.

Regular expression syntax is described in Appendix E, *Regular Expressions*.

Here is an `EOL_SPAN_REGEXP` that highlights MS-DOS batch file comments, which start with `REM`, followed by any whitespace character, and extend until the end of the line:

```
<EOL_SPAN_REGEXP AT_WHITESPACE_END="TRUE" HASH_CHAR="REM" TYPE="COMMENT1">REM\s<
```

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 the section called "Token Types" 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.

- `MATCH_TYPE` - Controls how the matched region will be highlighted. See the section called "The MATCH_TYPE Attribute" for more information.

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

```
<MARK_PREVIOUS AT_WHITESPACE_END="TRUE"
    MATCH_TYPE="DEFAULT">:</MARK_PREVIOUS>
```

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 the section called "Token Types" 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.

- `MATCH_TYPE` - Controls how the matched region will be highlighted. See the section called "The MATCH_TYPE Attribute" for more information.

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

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

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 the section called "Token Types" 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>
```

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 following attributes are supported:

- `HASH_CHAR` - a literal string which must be at the start of a regular expression.

- `HASH_CHARS` - a list of possible literal characters, one of which must match at the start of the regular expression.

`HASH_CHAR` and `HASH_CHARS` attributes are both optional, but you may only specify one, not both. If both are specified, `HASH_CHARS` is ignored and an error is shown. Whenever possible, use a literal prefix to specify a `SEQ_REGEXP`. If the starting prefix is always the same, use `HASH_CHAR` and provide as much prefix as possible. Only in rare cases would you omit both attributes, such as the case where there is no other reliable way to get the highlighting you need, for example, with comments in the Cobol programming language.

The regular expression match cannot span more than one line.

Regular expression syntax is described in Appendix E, *Regular Expressions*.

NOTE: c-style character escaping for literals (such as the tab char: \t) do not work as attribute values in XML. Use the XML character entity instead. For example: 	 instead of \t.

Here is a SEQ_REGEXP rule from moin.xml that uses the HASH_CHARS attribute, to describe a keyword (wikiword) that can start with any uppercase letter and contain lower case letters and at least one uppercase letter in the middle.

```
<SEQ_REGEXP HASH_CHARS="ABCDEFGHIJKLMNOPQRSTUVWXYZ" AT_WORD_START="TRUE"
TYPE="KEYWORD2">[A-Z][a-z]+[A-Z][a-zA-Z]+</SEQ_REGEXP>
```

The IMPORT Tag

The IMPORT tag, which must be placed inside a RULES tag, loads all rules defined in a given ruleset into the current ruleset; in other words, it has the same effect as copying and pasting the imported ruleset.

The only required attribute DELEGATE must be set to the name of a ruleset. To import a ruleset defined in the current mode, just specify its name. To import 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".

One quirk is that the definition of the imported ruleset is not copied to the location of the IMPORT tag, but rather to the end of the containing ruleset. This has implications with rule-ordering; see the section called "Rule Ordering Requirements".

Here is an example from the PHP mode, which extends the inline JavaScript highlighting to support embedded PHP:

```
<RULES SET="JAVASCRIPT+PHP">

<SPAN TYPE="MARKUP" DELEGATE="php::PHP">
    <BEGIN>&lt;?php</BEGIN>
    <END>?&gt;</END>
</SPAN>

<SPAN TYPE="MARKUP" DELEGATE="php::PHP">
    <BEGIN>&lt;?</BEGIN>
    <END>?&gt;</END>
</SPAN>

<SPAN TYPE="MARKUP" DELEGATE="php::PHP">
    <BEGIN>&lt;%=</BEGIN>
    <END>%&gt;</END>
</SPAN>

<IMPORT DELEGATE="javascript::MAIN"/>
</RULES>
```

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>
```

Token Types

The various token types are used to syntax highlight particular words in a language. This makes code easier to read. There is a wide latitude in the usage of the token types, and really it depends on the specifics of the language as to which token represents which type. Some examples are given below, but these are just guidelines, not hard and fast rules.

Many languages include constructs from other languages. One common example is html files can include javascript and css blocks. Several of the mode tags support a DELEGATE attribute, which will allow a section of text to be passed to a different mode for highlighting. The html mode delegates to the javascript mode for javascript blocks and to the css mode for style blocks. Use of the DELEGATE attribute is highly encouraged when appropriate since it makes writing modes easier, reduces duplication, and promotes visual consistency across languages.

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`

- `COMMENT3`

- `COMMENT4`

  jEdit supports four different types of comment tokens. Generally, comments are programmer-readable constructs that are ignored by compilers and interpreters. As an example, the lisp mode defines four comment types:

  <EOL_SPAN TYPE="COMMENT4">;;;;</EOL_SPAN>

  <EOL_SPAN TYPE="COMMENT3">;;;</EOL_SPAN>

  <EOL_SPAN TYPE="COMMENT2">;;</EOL_SPAN>

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

- `FUNCTION`

  The function token is intended to identify functions, methods, procedures, routines, or named subprograms.

- `DIGIT`

The digit token is to identify numbers.

- `INVALID`

  The invalid type is to indicate that particular words are not to be used, for example, the java mode defined both "goto" and "const" as invalid words. These are words that are defined by the language, but are not to be used.

- `KEYWORD1`

- `KEYWORD2`

- `KEYWORD3`

- `KEYWORD4`

  Keywords are used to identify well-defined words within a language. Some languages naturally divide keywords into groups, for example, the pascal mode identifies "for" as a KEYWORD1, "private" as a KEYWORD2, and "int" as a KEYWORD3.

- `LABEL`

  A label is generally a named position within a source, for example, the ada mode defined a label as <<foo>>.

- `LITERAL1`

- `LITERAL2`

- `LITERAL3`

- `LITERAL4`

  Literals are usually, but not always, uninterpreted strings, for example, "foo" or 'bar'. There are a wide variety of usages of literals in the mode files.

- `MARKUP`

  The markup token is generally used in the various "markup" languages, such as xml and html. Markup is used for those elements that are not specified as words belonging to the language. For example, in html, <body> would be considered a keyword, where <foo> would be considered markup.

- `OPERATOR`

  Common examples of operators are the math symbols, such as '+', '-', and so on.

The MATCH_TYPE Attribute

The `MATCH_TYPE` attribute is used by some of the rules to control how the region matched by the rule will be highlighted.

For example, when using a `MARK_PREVIOUS` rule to highlight a function call of the form `fcall()`, the following rule could be used:

```
<MARK_PREVIOUS TYPE="FUNCTION" MATCH_TYPE="OPERATOR">(</MARK_PREVIOUS>
```

This would cause `fcall` to be highlighted as `FUNCTION`, and `(` to be highlighted as `OPERATOR`. In this case, to maintain bracket matching working, a `SEQ` rule would have to be added to match `)` and mark it as `OPERATOR`.

The `MATCH_TYPE` attribute value can be any of the valid token types, or the following special values:

- `RULE`: this is the default value. It tells the syntax system to use the same token type as the TYPE attribute of the rule. This is equivalent to `EXCLUDE_MATCH="FALSE"` in 4.2 and earlier mode files.

- `CONTEXT`: using this value tells the syntax system to mark the matched region using the default token type for the current rule set. In 4.2 and earlier mode files, this was specified by `EXCLUDE_MATCH="TRUE"`.

Chapter 11. Installing Edit Modes

The easiest way to install a new mode locally is to use the Editing dialog found under Utilities - Global Options. At the bottom of this dialog is an "Add Mode" button. Clicking this button shows a dialog to add a mode to jEdit. Simply fill in the blanks and the mode file will be put in the proper place and the catalog file will be updated appropriately. Modes added this way can be removed by selecting the mode in the Editing dialog, then clicking the "Delete Mode" button. This button will only appear for modes that exist in the user settings directory. Core modes, that is, those modes distributed with jEdit cannot be deleted this way. The details of adding a mode by hand are below.

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 the section called "The jEdit Settings Directory".

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. This can also specify full paths, if the glob pattern contains a path separator character. `FILE_NAME_GLOB` can be specified in the `modes/catalog` file, or the mode file itself. See the `FILE_NAME_GLOB` for `apacheconf.xml` in `modes/catalog` for an example of full path filename globbing.

- `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, *Glob Patterns*.

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.

When a buffer is opened, jEdit must choose an edit mode for that buffer. It checks conditions in this order to decide which edit mode to use:

1. the filename is an exact match for the `FILE_NAME_GLOB`.

2. the filename matches the `FILE_NAME_GLOB` and the first line of the file matches the `FIRST_LINE_GLOB`

3. the filename matches the `FILE_NAME_GLOB`

4. the first line of the file matches the `FIRST_LINE_GLOB`

Chapter 12. Updating Edit Modes

From jEdit 4.2 to 4.4

1. All regular expressions in mode files were rewritten to use java.util.regex instead of gnu.regexp.

2. `HASH_CHAR` handling of `xxx_REGEXP` elements has been updated, as explained in the section called "The SPAN_REGEXP Tag".

3. The `EXCLUDE_MATCH` attribute got superseded by `MATCH_TYPE`. The attribute values translate from `TRUE` to `CONTEXT` and from `FALSE` to `RULE`, respectively. For more information see the section called "The MATCH_TYPE Attribute".

4. `NO_ESCAPE` is now deprecated and ignored by the parsing engine. `ESCAPE` is now a valid attribute for `SPAN` and `SPAN_REGEXP` rules.

Part 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.

Chapter 13. Macro Basics

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 8, *Using Macros*.

Single Execution Macros

As noted in the section called "How jEdit Organizes Macros", 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 the section called "Predefined Variables in BeanShell", 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.

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 variable naming the current, active View object.
Information about pre-defined variables can be found in the section called "Predefined Variables in
BeanShell".

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
```

Predefined Variables in BeanShell

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

- `buffer` - a Buffer object represents the contents of the currently visible open text file.

- `view` - A View represents the current top-level editor 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.

  This variable has the same value as the return value of:

  `jEdit.getActiveView()`

- `editPane` - an EditPane object contains a text area and buffer switcher. A view can be split to display edit panes. 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 the return value of:

  `view.getEditPane()`

- `textArea` - a JEditTextArea is the visible component that displays the current buffer.

  This variable has the same value as the return value of:

  `editPane.getTextArea()`

- `wm` - a DockableWindowManager is the visible component that manages dockable windows in the current view. This class is discussed in detail in Part IV, "Writing Plugins". This object is useful for writing macros that interface with, open, or close plugin windows.

  This variable has the same value the return value of:

  `view.getDockableWindowManager()`

- `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 variable values will be out of date. In that case, you can use the equivalent method calls.

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(view, "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.

Note that macros are not limited to using these methods for presenting a user interface. In fact, full-blown user interfaces using the Java Swing APIs are also possible, and will be covered later on in Chapter 14, *A Dialog-Based Macro*.

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 */
```

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 14. A Dialog-Based Macro

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

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);
```

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 the section called "Import Statements")
import javax.swing.border.*;

// main routine
void prefixSuffixDialog()
{
    // create dialog object (see the section called "Create the Dialog")
    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 the section called "Create 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");

// add a panel containing the buttons (see the section called "Create the Bu
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 the section called "Register the Action
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 the section called "Make the Dialo
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 the section called "The Action Listener")
void actionPerformed(e)
{
    if(e.getSource() != cancel)
    {
        processText();
    }
    dialog.dispose();
}

// this is where the work gets done to insert
// the prefix and suffix (see the section called "Get the User's Input")
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 the section called "Call jEdit Methods to Man
        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 the section called "The Main Routine")
prefixSuffixDialog();

// end of Add_Prefix_and_Suffix.bsh
```

Analysis of the Macro

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 the section called "The Mandatory First Example"), 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.

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.

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.

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.

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.

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.

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. this technique is discussed in the next chapter; see the section called "Implementing Classes and Interfaces".

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.

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.

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 15. Macro Tips and Techniques

Getting Input for a Macro

The dialog-based macro discussed in Chapter 14, *A Dialog-Based Macro* 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.

Getting a Single Line of Text

As mentioned earlier in the section called "Helpful Methods in the Macros Class", 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()`.

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 the section called "Helpful Methods in the Macros Class". 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()`).

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
```

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.

Startup Scripts

On startup, jEdit runs any BeanShell scripts located in the `startup` subdirectory of the jEdit installation and user settings directories (see the section called "The jEdit Settings Directory"). 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()`.

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.

Advanced BeanShell Techniques

BeanShell has a few advanced features that we haven't mentioned yet. They will be discussed in this section.

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.

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
```

Implementing Classes and Interfaces

As discussed in the macro example in Chapter 14, *A Dialog-Based Macro*, scripted objects can implicitly implement Java interfaces such as `ActionListener`. For example:

```
myRunnable() {
    run() {
        System.out.println("Hello world!");
    }

    return this;
}

Runnable r = myRunnable();
new Thread(r).start();
```

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. To prevent BeanShell from throwing exceptions for missing interface methods, 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) {}
```

In addition to the implicit interface definitions described above, BeanShell permits full-blown classes to be defined. Indeed, almost any Java class definition should work in BeanShell:

```
class Cons {
    // Long-live LISP!
    Object car;
    Object cdr;

    rplaca(Object car) {
        this.car = car;
```

```
    }

    rplacd(Object cdr) {
        this.cdr = cdr;
    }
}
```

Debugging Macros

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

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, *The Activity Log* 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.

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``);`

See the documentation for the Log class for information about the method's parameters.

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 16. 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

Java classes in plugins cannot make use of BeanShell commands directly. However, these commands can be called from BeanShell code that is part of a plugin, for example the snippets in `actions.xml`, or any BeanShell scripts shipped with the plugin and loaded on startup.

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.

- 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.

File Management Commands

- void **cd**(String *dirname*);

  Changes the working directory of the BeanShell interpreter to *dirname*.

- void **cp**(String *fromFile*, String *toFile*);

  Copy *fromFile* to *toFile*.

- 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**(void);

  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*.

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.

Resource Management Commands

- URL **getResource**(String *path*);

  Returns the resource specified by *path*. An absolute path must be used to return any resource available in the current classpath.

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`.

- org.gjt.sp.jedit.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 **setAccessibility**(boolean *flag*);

  If *flag* is `true`, BeanShell scripts are allowed to change and modify private variables, and call private methods. The default is `false`.

- void **setStrictJava**(boolean *flag*);

If *flag* is `true`, BeanShell scripts must follow a much more strict, Java-like syntax, and are not able to use the convenience features described in the section called "BeanShell's Convenience Syntax".

- void **source**(String *filename*);

Evaluates the contents of *filename* as a BeanShell script in the interpreter's current namespace.

BeanShell Object Management Commands

- **bind**(org.gjt.sp.jedit.bsh.This *ths*, org.gjt.sp.jedit.bsh.Namespace *namespace*);

Binds the scripted object *ths* to *namespace*.

- void **clear**(void);

Clear all variables, methods, and imports from this namespace. If this namespace is the root, it will be reset to the default imports.

- org.gjt.sp.jedit.bsh.This **extend**(org.gjt.sp.jedit.bsh.This *object*);

Creates a new BeanShell `This` scripted object that is a child of the parameter *object*.

- void **importObject**(Object *object*);

Import an object into this namespace. This is somewhat similar to Java 1.5 static class imports, except you can import the methods and fields of a Java object instance into a BeanShell namespace, for example:

```
Map map = new HashMap();
    importObject( map );
    put("foo", "bar");
    print( get("foo") ); // "bar"
```

- org.gjt.sp.jedit.bsh.This **object**(void);

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**(org.gjt.sp.jedit.bsh.Namespace *namespace*);

Set the namespace of the current scope to *namespace*.

- org.gjt.sp.jedit.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.

Other Commands

- `void` **`debug`**`(void);`

  Toggles BeanShell's internal debug reporting to the output stream of the current process.

- **`getSourceFileInfo`**`(void);`

  Returns the name of the file or other source from which the BeanShell interpreter is reading.

Part 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.

Where applicable, this section will also explain how jEdit's source code is similar. Therefore, this is also a good introduction to to jEdit development.

Chapter 17. 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 an 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 9, *Installing and Using Plugins* 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[1]. Make sure you can continue to use both your plugin and the editor after it has been reloaded.

- Each Plugin has an ActionSet defined by jEdit, which is added to the main ActionContext. The ActionSet is a container for EditAction instances. The plugin may define *actions* in a number of ways. One way is explicitly, with an action definition file known as `actions.xml`. Another is implicitly, by defining dockable windows in `dockables.xml`.

  Most EditActions 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. Plugin authors do not define specific shortcuts - the user can/ will assign EditActions to keyboard shortcuts, toolbar buttons, or entries in the text area's Context menu (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). These plugins are dependent on the JEditTextArea class, which is currently getting refactored.

- 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.

---

[1]You should test your plugin by loading and unloading it from both the Plugin Manager, as well as the **Activator Plugin**.

• 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 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, and join the `jedit-devel` mailing list, to keep updated on changes and bug reports, so that you will know when your plugin needs to be updated. If you allow the source code to be managed by the jEdit project, then other plugin developers can help fix things when they break more easily.

Chapter 18. 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.

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.

Additionally, plugins using the jEdit 4.2 plugin API can be loaded and unloaded at any time. This is a great help when developing your own plugins -- there is no need to restart the editor after making changes (see the section called "Reloading the Plugin").

Plugins are loaded from files with the `.jar` filename extension located in the `jars` subdirectories of the jEdit installation and user settings directories (see the section called "The jEdit Settings Directory").

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 `PluginJAR`. 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 the section called "The Property Files".

- Reads action definitions from any file named `actions.xml` in the archive (the file need not be at the top level). See the section called "The Actions.xml Catalog".

- 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 the section called "The dockables.xml Window Catalog".

- 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 `PluginJAR`. 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.

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?

• Should the plugin do something special when it gets focus?

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 OPTION_PREFIX = "options.quicknotepad.";
}
```

The class has been simplified since 4.1, and all we defined here were a couple of `String` data members to enforce consistent syntax for the name of properties we will use throughout the plugin.

• These names are used in `actions.xml` for each of the menu choices. This file is discussed in more detail in the section called "The Actions.xml Catalog". Each action is a beanshell script.

```
<!DOCTYPE ACTIONS SYSTEM "actions.dtd">
<ACTIONS>
  <ACTION NAME="quicknotepad.choose-file">
    <CODE>
      wm.addDockableWindow(QuickNotepadPlugin.NAME);
```

```
          wm.getDockableWindow(QuickNotepadPlugin.NAME).chooseFile();
       </CODE>
     </ACTION>

     <ACTION NAME="quicknotepad.save-file">
       <CODE>
         wm.addDockableWindow(QuickNotepadPlugin.NAME);
         wm.getDockableWindow(QuickNotepadPlugin.NAME).saveFile();
       </CODE>
     </ACTION>

     <ACTION NAME="quicknotepad.copy-to-buffer">
       <CODE>
         wm.addDockableWindow(QuickNotepadPlugin.NAME);
         wm.getDockableWindow(QuickNotepadPlugin.NAME).copyToBuffer();
       </CODE>
     </ACTION>
   </ACTIONS>
```

- The names also come up in the properties file, `QuickNotePad.props` file. The properties define option panes and strings used by the plugin. It is explained in more detail in the section called "The Property Files" and the `EditPlugin` API docs.

```
# jEdit only needs to load the plugin the first time the user accesses it
# the presence of this property also tells jEdit the plugin is using the new A
plugin.QuickNotepadPlugin.activate=defer

# Even if you don't store additional files, this is a good idea to set:
plugin.QuickNotepadPlugin.usePluginHome=true

# Required for all plugins:
plugin.QuickNotepadPlugin.name=QuickNotepad
plugin.QuickNotepadPlugin.author=John Gellene

# version number == jEdit version number
plugin.QuickNotepadPlugin.version=4.5

# online help
plugin.QuickNotepadPlugin.docs=index.html

# we only have one dependency, jEdit 4.5
# See jEdit.getBuild() to understand version numbering scheme.
plugin.QuickNotepadPlugin.depend.0=jedit 4.05.99.00

# quicknotepad's plugin menu - a list of actions or separators
plugin.QuickNotepadPlugin.menu=quicknotepad \
  - \
  quicknotepad.choose-file \
  quicknotepad.save-file \
  quicknotepad.copy-to-buffer

# action labels for actions supplied by dockables.xml
quicknotepad.label=QuickNotepad

# action labels for actions supplied by actions.xml
quicknotepad.choose-file.label=Choose notepad file
quicknotepad.save-file.label=Save notepad file
```

```
quicknotepad.copy-to-buffer.label=Copy notepad to buffer

# plugin option pane
plugin.QuickNotepadPlugin.option-pane=quicknotepad

# Option pane activation BeanShell snippet
options.quicknotepad.code=new QuickNotepadOptionPane();

# 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

# window title
quicknotepad.title=QuickNotepad

# window toolbar buttons
quicknotepad.choose-file.icon=Open.png
quicknotepad.save-file.icon=Save.png
quicknotepad.copy-to-buffer.icon=CopyToBuffer.png

# default 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=
```

The Property Files

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. Others are accessed by the scripts used by plugin packagers [1].

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 \\.
```

---

[1] See the **Macros/Properties/Create Plugin Announcement** macro for an example.

Now we look at a fragment from the `QuickNotepad.props` file [2] 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.activate=defer
plugin.QuickNotepadPlugin.usePluginHome=true
plugin.QuickNotepadPlugin.name=QuickNotepad
plugin.QuickNotepadPlugin.author=John Gellene
plugin.QuickNotepadPlugin.version=4.5
plugin.QuickNotepadPlugin.docs=QuickNotepad.html
# depends on jEdit 4.5
plugin.QuickNotepadPlugin.depend.0=jedit 04.05.99.00
plugin.QuickNotepadPlugin.description=A demo jEdit plugin that provides a notep
plugin.QuickNotepadPlugin.longdescription=description.html
```

These properties are each 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 the section called "The dockables.xml Window Catalog".

```
# dockable window name
quicknotepad.title=QuickNotepad
```

Next, we see menu item labels for the plugin's actions. All of these but the first are defined in `actions.xml` file, and that is because the dockable itself has its own actions. Actions are discussed further in the section called "The Actions.xml Catalog".

```
# action labels
# Dockable label
quicknotepad.label=QuickNotepad
# Additional strings extracted from the plugin java source
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 the section called "The QuickNotepadPlugin Class".

```
# 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
```

---

[2]Examine the actual file for a more complete example

```
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=
```

PropertySideKick

There is a SideKick for Property files, provided in the JavaSideKick plugin. This gives you a compact and sorted tree view of property files.

Localization Files

In addition to property files ending in `.props`, you will find property files with the names like `lang_de.properties`. Each of these files provides localized strings for a particular locale. In the example above, it is for the German locale. These files are loaded by jEdit automatically when that locale is in use. They need to have a different filename extension from the other property files so they can be treated differently from the regular properties.

The EditBus

jEdit (and some plugins) generate several kinds of messages to alert plugins and other components of jedit-specific events. The message classes, all 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.

For example, the ViewUpdate messages are all related to the jEdit View, or the top-level window. If the user creates multiple Views, a plugin may need to know when they are created or destroyed, so it would monitor ViewUpdate messages.

BufferUpdate messages are all related to jEdit buffers. They let plugins know when a buffer has become dirty, when it is about to be closed, after it is closed, created, loaded, or saved. Each of these messages are described in further detail in the API docs.

As another example, The Navigator plugin monitors an EBMessage of the kind BufferChanging. The BufferChanging event provides Navigator enough advance notice to save the TextArea's caret just before the current EditPane changes its active Buffer. The `BufferChanged` event, another `EditPaneUpdate` message, is thrown shortly afterward. This is not used by Navigator, but it is used by SideKick to determine when it is time to reparse the buffer.

Plugins register `EBComponent` instances with the `EditBus` to receive messages reflecting changes in jEdit's state.

EBComponents 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.

Using the Activity Log to see the EditBus

To determine precisely which EditBus messages are being sent by jEdit or the plugins, start up jEdit with an additional argument, `-log=5`. You can set an even lower log level to see further details (the default is 7). With a log level of 5 or lower, the Activity Log will include [notice]s, which will show us exactly which EditBus message is sent and when. See Appendix B, *The Activity Log* for more details.

The Actions.xml Catalog

In jEdit as well as in Plugins, actions define procedures that can be bound to a menu item, a toolbar button or a keyboard shortcut. Most plugin Actions [3] 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:

```
<ACTIONS>
  <ACTION NAME="quicknotepad.choose-file">
    <CODE>
      wm.addDockableWindow(QuickNotepadPlugin.NAME);
      wm.getDockableWindow(QuickNotepadPlugin.NAME).chooseFile();
    </CODE>
  </ACTION>

  <ACTION NAME="quicknotepad.save-file">
    <CODE>
      wm.addDockableWindow(QuickNotepadPlugin.NAME);
      wm.getDockableWindow(QuickNotepadPlugin.NAME).saveFile();
    </CODE>
  </ACTION>

  <ACTION NAME="quicknotepad.copy-to-buffer">
    <CODE>
      wm.addDockableWindow(QuickNotepadPlugin.NAME);
      wm.getDockableWindow(QuickNotepadPlugin.NAME).copyToBuffer();
    </CODE>
  </ACTION>
</ACTIONS>
```

---

[3]Some plugins, such as Sidekick, Console, and ProjectViewer, create pure Java EditAction-derived Actions, based which services are available, or which files are found in a certain path. However, this is an advanced topic you can explore further in the source and API docs of those plugins.

Actions in jEdit core

You can see how each action in jEdit core is implemented by inspecting the `actions.xml` file that is there.

This file defines three actions. They each use a built-in variable `wm`, which refers to the current view's `DockableWindowManager`. Whenever you need to obtain a reference to the current dockable, or create a new one, this is the class to use. We use the method `addDockable()` followed by `getDockable()` to create if necessary, and then bring up the QuickNotepad plugin dockable. This will be docked or floating, depending on how it was last used.

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, "Writing Macros"; 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.

The dockables.xml Window Catalog

A Dockable is a window that can float like a dialog, or dock into jEdit's docking area. Each dockable needs a label (for display in menus, and on small buttons) and a title (for display in the floating window's title bar).

The jEdit 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 the section called "The Property Files".

For each dockable, jedit defines an action with the same name. This means you do not need to define an explicit action to create your dockable - in fact, jEdit defines three actions: "toggle", "get" and "new floating instance" for each.

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. You can use

position to customize the layout of your plugin depending on whether it appears on the sides, or the top/bottom, or as a floating dockable.

A formal description of each element of the `dockables.xml` file can be found in the documentation of the `DockableWindowManager` class. This class also contains the public interface you should use for getting, showing, hiding, and other interactions with the plugin's top-level windows.

jEdit's dockables

jEdit has its own `dockables.xml` file that you can inspect to see how jEdit creates its own dockables.

The services.xml file

A "service" is a mechanism by which one plugin can work with other plugins and avoid a bidirectional build-dependency. For example, the XML plugin "depends" on Sidekick, but in fact, it is SideKick which creates and operates on an object (a `SideKickParser`, in fact) defined in the XML plugin. In a way, the dependency is bidirectional.

Similarly, the SshConsole plugin defines but does not instantiate a `Shell` object. It is the Console plugin which creates a specific shell for each available service. SideKick and Console use the `ServiceManager` to search for services offered by other plugins.

Here is an example of a service from the XML plugin, which extends Sidekick:

```
<!DOCTYPE SERVICES SYSTEM "services.dtd">
<SERVICES>
    <SERVICE CLASS="sidekick.SideKickParser" NAME="html">
        new sidekick.html.HtmlParser();
    </SERVICE>
  [...]
</SERVICES>
```

The value of the `CLASS=` should be a base-class or interface of the object that is returned by executing the beanshell factory method enclosed in the `<SERVICE>` tag.

In the case above, the returned object tells Sidekick how it can parse files of a specific type (HTML). The API docs for `SideKickParser` should indicate precisely which methods must be implemented in a plugin which offers this service.

For more information about services, refer to the ServiceManager class API documentation. There, you can find out what the tags and attributes mean, as well as how to register and use services. You can also inspect the `services.xml` file of jEdit core to see what services are offered by jEdit itself.

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.

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.

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.

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 the section called "The Property Files".

The build.xml Ant build file

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 a version of `build.xml` that could be used by QuickNotepad:

```
<project name="QuickNotepad" default="build">
  <description>
    This is an ant build.xml file for building the QuickNotepad plugin for jEdi
  </description>

  <property file="build.properties"/>
  <property file="../build.properties"/>
  <property name="user-doc.xml" location = "users-guide.xml"/>
  <import file="${build.support}/plugin-build.xml" />

  <!-- Extra files that should be included in the jar -->
  <selector id="packageFiles">
    <or>
        <filename name="*.txt" />
    </or>
  </selector>
</project>
```

This build file imports another modular build file, `plugin-build.xml` from the `build-support` project. It is available as a package you can check out from subversion, or found online in the jEdit's SVN repository. It contains the common build steps used to build the core jEdit plugins, and some example `build.properties.sample` files which you can adapt for use with your development environment.

Customizing this build file for a different plugin will likely only require three changes to build.xml file:

- the name of the project

- the dependencies of the plugin

- The extra files that need to be copied into the jar.

Because this build file and those used by most plugins import a `build.properties` file from the current and the parent directories, it is possible to build most of jEdit's plugins in a uniform way by setting the common properties in a single `build.properties` file, placed in the plugin source's parent directory.

Tip

For a full discussion of the `Ant` file format and command syntax, you should consult the Ant documentation, also available through jEdit's help system if you installed the Ant Plugin. When editing Ant build files, the XML plugin gives you completion tips for both elements *and* attributes. The Console plugin provides you with an ANT button which you can bind to keyboard actions. In addition, there are the AntFarm and Antelope plugins which also proivde you with alternate means to execute Ant targets through the Console.

Reloading the Plugin

Once you have compiled your plugin, you will need to test its behavior when it is reloaded. Follow these steps to reload your plugin without restarting jEdit:

• From the Plugins menu open the Plugin Manager.

• On the Manage tab uncheck Hide libraries. This will allow you to see plugins that are not loaded.

• Recheck the plugin to reload it.

Tip

The Activator plugin provides a very convenient (dockable) way to test the activating and reloading behavior of your plugin. Be sure to test your plugin's reloading behavior with both the Activator and the Reloader tabs.

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.

Tips for debugging plugins

BeanShell

jEdit includes a Beanshell interface into its currently running JVM at all times. You can access it a variety of ways, but one way is from `Plugins - Console - Shells - BeanShell`. From here, you can interactively inspect the values of any object in memory, call any of its member functions, or create new instances of any class that is currently loaded by jEdit or any of its plugins. All this, without setting any breakpoints!

If you're too lazy to type each Beanshell statement interactively, you can also create debugging code snippets as macros and invoke them from `utilities - beanshell - evaluate selection`, or `Macros - Misc - Evaluate Buffer in Beanshell`, or place the file in your own macros directory and bind it to its own keyboard shortcut.

Other useful tips

This section is new but will be expanded shortly. Please post suggestions to the `jedit-devel` mailing list.

Chapter 19. Plugin Tips and Techniques

Utility Classes

If you need to check what the current running platform is, see `org.gjt.sp.jedit.OperatingSystem`.

If you need to compare file names and don't want to worry about whether you are on a case-sensitive file system or not, use `MiscUtilities.pathsEqual()`. See that class for other useful path manipulation routines.

There are other helpful classes in org.gjt.sp.util, such as `HtmlUtilities`, `StandardUtilities`, `IOUtilities`, and `Log`. In particular, the `Log` class is recommended for reporting errors and also logging debug information.

If your plugin needs to "find a word" (using word boundaries) somewhere in the buffer, you might want to reuse the `org.gjt.sp.jedit.TextUtilities` class.

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.

Bundling Additional Non-Java Libraries

If your plugin bundles non-Java files, like native libraries, you need to list them in the `plugin.class name.files` property. If you don't do so, they don't get deleted if the plugin is uninstalled. See the documentation for the `EditPlugin` class for details.

Storing plugin data

If your plugin needs to create files and store data in the filesystem, you should use the `getPluginHome()` API of the `EditPlugin` class. To signal that you use the plugin home API you have to set the `plugin.class name.usePluginHome` property to `true`. Even if your plugin doesn't create any files, you should set the property to `true`, so that e. g. the plugin manager knows that there is actually no data in favor of not knowing if there is any data and thus displaying that it doesn't know the data size. See the documentation for the `EditPlugin` class for details.

Plugin colors

There are a number of colors used by the View that should also be used by plugins where possible. This helps promote a consistent color scheme throughout jEdit.

The main color properties are:

- `view.bgColor` - the background color of the main text area

- `view.fgColor` - the base foreground color for text in the main text area

- `view.lineHighlightColor` - color of the current line highlight

- `view.selectionColor` - the color of selected text in the main text area

- `view.caretColor` - the color of the caret in the main text area

- `view.eolMarkerColor` - the color of the end-of-line marker

To use these colors in your plugin, use

```
jEdit.getColorProperty("view.whatever", default_color)
```

For example, the QuickNotepad example should have lines like this:

```
        textarea.setBackground(jEdit.getColorProperty("view.bgColor", Color.WHI
        textarea.setForeground(jEdit.getColorProperty("view.fgColor", Color.BLA
```

This sets the foreground and background colors of QuickNotepad to be the same as those in the View.

There are other color properties that may be useful, depending on what your plugin displays.

Gutter colors:

- `view.gutter.bgColor`

- `view.gutter.currentLineColor`

- `view.gutter.fgColor`

- `view.gutter.focusBorderColor`

- `view.gutter.foldColor`

- `view.gutter.highlightColor`

- `view.gutter.markerColor`

- `view.gutter.noFocusBorderColor`

- `view.gutter.registerColor`

- `view.gutter.structureHighlightColor`

Status bar colors:

- `view.status.background`

- `view.status.foreground`

- `view.status.memory.background`

- `view.status.memory.foreground`

Structure highlight colors:

- `view.structureHighlightColor`

- `view.structureHighlightColor`

Style colors. Use `GUIUtilities.parseStyle` for these.

- `view.style.comment1`

- `view.style.comment2`

- `view.style.comment3`

- `view.style.comment4`

- `view.style.digit`

- `view.style.foldLine.0`

- `view.style.foldLine.1`

- `view.style.foldLine.2`

- `view.style.foldLine.3`

- `view.style.function`

- `view.style.invalid`

- `view.style.keyword1`

- `view.style.keyword2`

- `view.style.keyword3`

- `view.style.keyword4`

- `view.style.label`

- `view.style.literal1`

- `view.style.literal2`

- `view.style.literal3`

- `view.style.literal4`

- `view.style.markup`

- `view.style.operator`

- `view.wrapGuideColor`

For example, here is a setting for a fold line color:

```
view.style.foldLine.0=color\:\#000000 bgColor\:\#f5deb8 style\:b
```

Passing the value to `GUIUtilities.parseStyle` will return a SyntaxStyle object, which you can query for background color, foreground color, and font.