

The KatePart Handbook

**Thad McGinnis
Anne-Marie Mahfouf
Anders Lund
T.C. Hollingsworth
Christoph Cullmann
Lauri Watts**



The KatePart Handbook

Contents

1	Introduction	8
2	Some Fundamentals	9
2.1	Drag and Drop	9
2.2	Shortcuts	9
3	Working with the KatePart editor	12
3.1	Overview	12
3.2	Navigating in the Text	13
3.3	Working with the Selection	13
3.3.1	Using Block Selection	14
3.3.2	Using Overwrite Selection	14
3.3.3	Using Persistent Selection	14
3.4	Copying and Pasting Text	14
3.5	Finding and Replacing Text	15
3.5.1	The Search and Replace Bars	15
3.5.2	Finding Text	16
3.5.3	Replacing Text	16
3.6	Using Bookmarks	17
3.7	Automatically Wrapping text	17
3.8	Using automatic indenting	18
3.9	Line Modification Indicators	18
3.10	The Scrollbar Minimap	19
3.11	Multiple cursors	19
3.11.1	Creating multiple cursors	20
3.11.2	Working with multiple cursors	20
4	The Menu Entries	21
4.1	The File Menu	21
4.2	The Edit Menu	22
4.3	The Selection Menu	24
4.4	The View Menu	25
4.5	The Go Menu	26
4.6	The Tools Menu	27
4.7	The Settings and Help Menu	31

5	Advanced Editing Tools	32
5.1	Comment/Uncomment	32
5.2	The Editor Component Command Line	32
5.2.1	Standard Command Line Commands	33
5.2.1.1	Commands for Configuring the Editor	33
5.2.1.2	Commands for editing	34
5.2.1.3	Commands for navigation	39
5.2.1.4	Commands for Basic Editor Functions (These depend on the application the editor component is used in)	40
5.3	Using Code Folding	40
6	Extending KatePart	42
6.1	Introduction	42
6.2	Working with Syntax Highlighting	42
6.2.1	Overview	42
6.2.2	The KatePart Syntax Highlight System	43
6.2.2.1	How it Works	43
6.2.2.2	Rules	44
6.2.2.3	Context Styles and Keywords	44
6.2.2.4	Default Styles	44
6.2.3	The Highlight Definition XML Format	45
6.2.3.1	Overview	45
6.2.3.2	The Sections in Detail	47
6.2.3.3	Available Default Styles	49
6.2.4	Highlight Detection Rules	50
6.2.4.1	The Rules in Detail	53
6.2.4.2	Tips & Tricks	56
6.3	Working with Color Themes	57
6.3.1	Overview	57
6.3.2	The KSyntaxHighlighting Color Themes	58
6.3.3	The Color Themes JSON Format	59
6.3.3.1	Overview	59
6.3.3.2	The JSON Structure	60
6.3.3.3	Main Sections of the JSON Color Theme Files	60
6.3.3.4	Metadata	62
6.3.4	Colors in Detail	62
6.3.4.1	Editor Colors	63
6.3.4.2	Default Text Styles	69
6.3.4.3	Custom Highlighting Text Styles	72
6.3.5	The Color Themes GUI	73
6.3.5.1	Create a new theme	74
6.3.5.2	Import or export JSON theme files	74

The KatePart Handbook

6.3.5.3	Editing color themes	74
6.3.5.3.1	Colors	74
6.3.5.3.2	Default Text Styles	74
6.3.5.3.3	Highlighting Text Styles	74
6.3.6	Tips & Tricks	75
6.3.6.1	Contrast of Text Colors	75
6.3.6.2	Suggestions of Consistency with Syntax Highlighting	75
6.4	Scripting with JavaScript	75
6.4.1	Indentation Scripts	75
6.4.1.1	The Indentation Script Header	76
6.4.1.2	The Indenter Source Code	76
6.4.2	Command Line Scripts	78
6.4.2.1	The Command Line Script Header	78
6.4.2.2	The Script Source Code	79
6.4.2.2.1	Binding Shortcuts	79
6.4.3	Scripting API	80
6.4.3.1	Cursors and Ranges	80
6.4.3.1.1	The Cursor Prototype	80
6.4.3.1.2	The Range Prototype	81
6.4.3.2	Global Functions	82
6.4.3.2.1	Reading & Including Files	83
6.4.3.2.2	Debugging	83
6.4.3.2.3	Translation	83
6.4.3.3	The View API	84
6.4.3.4	The Document API	85
6.4.3.5	The Editor API	90
7	Configure KatePart	92
7.1	The Editor Component Configuration	92
7.1.1	Appearance	92
7.1.1.1	General	92
7.1.1.2	Borders	94
7.1.2	Color Themes	95
7.1.3	Editing	95
7.1.3.1	General	95
7.1.3.2	Text Navigation	96
7.1.3.3	Indentation	97
7.1.3.4	Auto Completion	98
7.1.3.5	Spellcheck	98
7.1.3.6	Vi Input Mode	98
7.1.4	Open/Save	99

The KatePart Handbook

7.1.4.1	General	99
7.1.4.2	Advanced	100
7.1.4.3	Modes & Filetypes	101
7.2	Configuring With Document Variables	102
7.2.1	How KatePart uses Variables	102
7.2.2	Available Variables	103
7.2.3	Extended Options in <code>.kateconfig</code> files	105
8	Credits and License	106
9	The VI Input Mode	107
9.1	VI Input Mode	107
9.1.1	Incompatibilities with Vim	107
9.1.2	Switching Modes	108
9.1.3	Integration with Kate features	108
9.1.4	Supported normal/visual mode commands	108
9.1.5	Supported motions	110
9.1.6	Supported text objects	111
9.1.7	Supported insert mode commands	112
9.1.8	The Comma Text Object	113
9.1.9	Missing Features	113
A	Regular Expressions	114
A.1	Introduction	114
A.2	Patterns	115
A.2.1	Escaping characters	115
A.2.2	Character Classes and abbreviations	115
A.2.2.1	Characters with special meanings inside character classes	117
A.2.3	Alternatives: matching 'one of'	117
A.2.4	Sub Patterns	117
A.2.4.1	Specifying alternatives	117
A.2.4.2	Capturing matching text (back references)	117
A.2.4.3	Lookahead Assertions	118
A.2.4.4	Lookbehind Assertions	118
A.2.5	Characters with a special meaning inside patterns	118
A.3	Quantifiers	119
A.3.1	Greedy	119
A.3.2	In context examples	120
A.4	Assertions	120
B	Index	122

Abstract

KatePart is a fully featured editor component by KDE.

Chapter 1

Introduction

KatePart is a fully featured text editor component used by many Qt™ and KDE applications. KatePart is more than a text editor; it is meant to be a programmer's editor, and could be considered as at least a partial alternative to more powerful editors. One of KatePart's main features is the colorized syntax, customized for many different programming languages such as: C/C++, Java™, Python, Perl, Bash, Modula 2, HTML, and Ada.

KWrite is a simple text editor application based on KatePart. It has a single document interface (SDI) allowing you to edit one file at the time per window. Since KWrite is a very simple implementation of KatePart, it does not require its own documentation. If you know how to use KWrite, you can use KatePart anywhere!

Chapter 2

Some Fundamentals

KWrite and many other KatePart users are very simple to use. Anyone that has used a text editor should have no problems.

2.1 Drag and Drop

KatePart uses the KDE Drag and Drop protocol. Files may be dragged and dropped onto KatePart from the Desktop, the filemanager Dolphin, or some remote FTP site opened in one of Dolphin's windows.

2.2 Shortcuts

Many of the shortcuts are configurable by way of the [Settings](#) menu. By default KatePart honors the following shortcuts:

Ins	Toggle between Insert and Overwrite mode. When in insert mode the editor will add any typed characters to the text while pushing along any data to the right of the text cursor. Overwrite mode causes the entry of each character to eliminate the character immediately to the right of the text cursor.
Left	Move the cursor one character to the left.
Right	Move the cursor one character to the right.
Up	Move the cursor up one line.
Down	Move the cursor down one line.
Ctrl+E	Go to previous edit location in document.
Ctrl+Shift+E	Go to next edit location in document.
Alt+Shift+Up	Move cursor to previous matching indent.
Alt+Shift+Down	Move cursor to previous matching indent.
Ctrl+6	Move to Matching Bracket.
PgUp	Move the cursor up one page.
PgDn	Move the cursor down one page.
Home	Move the cursor to the beginning of the line.

The KatePart Handbook

End	Move the cursor to the end of the line.
Ctrl+Home	Move to Beginning of Document.
Ctrl+End	Move to End of Document.
Ctrl+Up	Scroll Line Up.
Ctrl+Down	Scroll Line Down.
Ctrl+Right	Move Word Right.
Ctrl+Left	Move Word Left.
Ctrl+Shift+Up	Move Lines Up.
Ctrl+Shift+Down	Move Lines Down.
Ctrl+.	Duplicate Selected Lines Down.
Ctrl+B	Set a Bookmark.
Alt+PgUp	Previous Bookmark.
Alt+PgDn	Next Bookmark.
Del	Delete the character to the right of the cursor (or any selected text).
Backspace	Delete the character to the left of the cursor.
Ctrl+Del	Delete Right Word.
Ctrl+Backspace	Delete Left Word.
Ctrl+K	Delete Line.
Shift+Enter	Insert newline including leading characters of the current line which are not letters or numbers. It is useful e.g. to write comments in the code: At the end of the line '// some text' press this shortcut and the next line starts already with '// '. So you do not have to enter the comment characters at the beginning of each new line with comments.
Ctrl+Shift+Enter	Create a new line below current line.
Ctrl+Alt+Enter	Create a new line above current line.
Shift+Left	Mark text one character to the left.
Shift+Right	Mark text one character to the right.
Ctrl+F	Find.
F3	Find Next.
Shift+F3	Find Previous.
Ctrl+H	Find Selected.
Ctrl+Shift+H	Find Selected Backwards.
Ctrl+Shift+Right	Select Word Right.
Ctrl+Shift+Left	Select Word Left.
Shift+Home	Select to Beginning of Line.
Shift+End	Select to End of Line.
Shift+Up	Select to Previous Line.
Shift+Down	Select to Next Line.
Ctrl+Shift+6	Select to Matching Bracket.
Ctrl+Shift+PgUp	Select to Top of View.
Ctrl+Shift+PgDn	Select to Bottom of View.
Shift+PgUp	Select Page Up.
Shift+PgDn	Select Page Down.
Ctrl+Shift+Home	Select to Beginning of Document.
Ctrl+Shift+End	Select to End of Document.
Ctrl+Home	Select All.
Ctrl+Shift+A	Deselect.
Ctrl+Shift+B	Block Selection Mode.
Ctrl+C / Ctrl+Ins	Copy the marked text to the clipboard.

The KatePart Handbook

Ctrl+D	Comment.
Ctrl+Shift+D	Uncomment.
Ctrl+G	Go to line...
Ctrl+I	Indent selection.
Ctrl+Shift+I	Unindent selection.
Ctrl+J	Join Lines.
Ctrl+P	Print .
Ctrl+R	Replace .
Ctrl+S	Invokes the Save command.
Ctrl+Shift+S	Save As.
Ctrl+U	Uppercase.
Ctrl+Shift+U	Lowercase.
Ctrl+Alt+U	Capitalize.
Ctrl+V / Shift+Ins	Paste the clipboard text into line edit.
Ctrl+X / Shift+Ins	Delete the marked text and copy it to the clipboard.
Ctrl+Z	Undo .
Ctrl+Shift+Z	Redo .
Ctrl+-	Shrink Font.
Ctrl++Ctrl+=	Enlarge Font.
Ctrl+Shift+-	Fold Toplevel Nodes.
Ctrl+Shift++	Unfold Toplevel Nodes.
Ctrl+Space	Invoke Code Completion.
F5	Reload .
F6	Show/Hide Icon Border.
F7	Switch to Command Line.
F9	Show/Hide Folding Markers.
F10	Dynamic Word Wrap.
F11	Show/Hide Line Numbers.
Ctrl+T	Transpose Characters.
Ctrl+Shift+O	Automatic Spell Checking.
Ctrl+Shift+V	Switch to Next Input Mode.
Ctrl+8	Reuse Word Above.
Ctrl+9	Reuse Word Below.
Ctrl+Alt+#	Expand Abbreviation.
Ctrl+Alt+Up	Add a cursor above current cursor.
Ctrl+Alt+Down	Add a cursor below current cursor.
Shift+Alt+I	Create a cursor at the end of each line in selection.
Alt+J	Find next occurrence of the word under cursor and select it.
Ctrl+Alt+Shift+J	Find all occurrences of the word under cursor and select them.

Chapter 3

Working with the KatePart editor

Anders Lund
Dominik Haumann

3.1 Overview

The KatePart editor is the editing area of the KatePart window. This editor is shared between Kate and KWrite, and it can also be used in Konqueror for displaying text files from your local computer, or from the network.

The editor is composed of the following components:

The editing area

This is where the text of your document is located.

The Scrollbars

The scrollbars indicate the position of the visible part of the document text, and can be used to move around the document. Dragging the scrollbars will not cause the insertion cursor to be moved.

The scrollbars are displayed and hidden as required.

The Icon Border

The icon border is a small pane on the left side of the editor, displaying a small icon next to marked lines.

You can set or remove a [bookmark](#) in a visible line by clicking the left mouse button in the icon border next to that line.

The display of the icon border can be toggled using the **View** → **Show Icon Border** menu item.

The Line Numbers Pane

The Line numbers pane shows the line numbers of all visible lines in the document.

The display of the Line Numbers Pane can be toggled using the **View** → **Show Line Numbers** menu item.

The Folding Pane

The folding pane allows you to collapse or expand foldable blocks of lines. The calculation of the foldable regions is done according to rules in the syntax highlight definition for the document.

ALSO IN THIS CHAPTER:

- [Navigating in the Text](#)
- [Working with the Selection](#)
- [Copying and Pasting Text](#)
- [Finding and Replacing Text](#)
- [Using Bookmarks](#)
- [Automatically Wrapping Text](#)
- [Using automatic indenting](#)

3.2 Navigating in the Text

Moving around in the text in KatePart is similar to most graphical text editors. You move the cursor using the arrow keys and the **PgUp**, **PgDn**, **Home** and **End** keys in combination with the **Ctrl** and **Shift** modifiers. The **Shift** key is always used to generate a selection, while the **Ctrl** key has different effects on different keys:

- For the **Up** and **Down** keys it means scroll rather than move the cursor.
- For the **Left** and **Right** keys it means skip words rather than characters.
- For the **PgUp** and **PgDn** keys it means move to the visible edge of the view rather than browse.
- For the **Home** and **End** keys it means move to the beginning or end of the document rather than the beginning or end of the line.

KatePart also provides you with a way to quickly jump to a matching brace or parenthesis: place the cursor on the inside of a parenthesis or brace character, and press **Ctrl+6** to jump to the matching parenthesis or brace.

In addition you can use [bookmarks](#) to quickly jump to positions that you define on your own.

3.3 Working with the Selection

There are two basic ways of selecting text in KatePart: using the mouse, and using the keyboard.

To select using the mouse, hold down the left mouse button while dragging the mouse cursor from where the selection should start, to the desired end point. The text gets selected as you drag.

Double-clicking a word will select that word.

Triple-clicking in a line will select the entire line.

If **Shift** is held down while clicking, text will be selected:

- If nothing is already selected, from the text cursor position to the mouse cursor position.
- If there is a selection, from and including that selection to the mouse cursor position.

NOTE

When selecting text by dragging the mouse, the selected text is copied to the clipboard, and can be pasted by clicking the middle mouse button in the editor, or in any other application to which you want to paste the text.

To select using the keyboard, hold down the **Shift** key while using the navigation keys (Arrow keys, **PgUp**, **PgDn**, **Home** and **End**, possibly in combination with **Ctrl** to extend the move of the text cursor).

See also the section [Navigating in the Text](#) in this chapter.

To Copy the current selection, use the **Edit** → **Copy** menu item or the keyboard shortcut (defaults to **Ctrl+C**).

To Deselect the current selection, use the **Edit** → **Deselect** menu item, or the keyboard shortcut (default is **Ctrl+Shift+A**), or click with the left mouse button in the editor.

3.3.1 Using Block Selection

When Block Selection is enabled, you can make ‘vertical selections’ in the text, meaning selecting limited columns from multiple lines. This is handy for working with tab separated lines for example.

Block Selection can be toggled using the **Edit** → **Block Selection Mode** menu item. The default keyboard shortcut is **Ctrl+Shift+B**.

3.3.2 Using Overwrite Selection

If Overwrite Selection is enabled, typing or pasting text into the selection will cause the selected text to be replaced. If not enabled, new text will be added at the position of the text cursor.

Overwrite Selection is enabled by default.

To change the setting for this option, use the [Cursor & Selection](#) page of the [Configuration Dialog](#).

3.3.3 Using Persistent Selection

When Persistent Selection is enabled, typing characters or moving the cursor will not cause the Selection to become deselected. This means that you can move the cursor away from the selection and type text.

Persistent Selection is disabled by default.

Persistent Selection can be enabled in the [Cursor & Selection](#) page of the [Configuration Dialog](#).

WARNING

If Persistent Selection and Overwrite Selection are both enabled, typing or pasting text when the text cursor is inside the selection will cause it to be replaced and deselected.

3.4 Copying and Pasting Text

To copy text, select it and use the **Edit** → **Copy** menu item. Additionally, selecting text with the mouse will cause selected text to be copied to the X selection.

To paste the text currently in the clipboard, use the **Edit** → **Paste** menu item.

Additionally, text selected with the mouse may be pasted by clicking the middle mouse button at the desired position.

TIP

If you are using the KDE desktop, you can retrieve earlier copied text from any application using the Klipper icon in the system tray.

3.5 Finding and Replacing Text

3.5.1 The Search and Replace Bars

KatePart has an incremental search bar and a power search and replace bar, which offers the means of entering a replacement string along with a few extra options.

The bars offer the following common options:

Find

This is where to enter the search string. The interpretation of the string depends on some of the options described below.

Match case

If enabled, the search will be limited to entries that match the case (upper or lower) of each of the characters in the search pattern.

The power search and replace bar offers some additional options:

Plain Text

Literally match any occurrence of the search string.

Whole Words

If selected, the search will only match if there is a word boundary at both ends of the string matching, meaning not an alphanumeric character - either some other visible character or a line end.

Escape Sequences

If selected, the **Add** menuitem at the bottom of the context menu of the text boxes will be enabled and allows you to add escape sequences to the search pattern from a predefined list.

Regular Expression

If selected, the search string is interpreted as a regular expression. The **Add** menuitem at the bottom of the context menu of the text boxes will be enabled and allows you to add regular expression items to the search pattern from a predefined list.

See [Regular Expressions](#) for more on these.

Search in the selection only

If checked, the search and replace will be performed within the selected text only.

Find all



Clicking this button highlights all matches in the document and shows the number of found matches in a small popup.

3.5.2 Finding Text


To find text, launch the incremental search bar with **Ctrl+F** or from the **Edit** → **Find...** menu item.


This opens the incremental search bar at the bottom of the editor window. On the left side of the bar is a button with an icon to close the bar, followed by a small text box for entering the search pattern.

When you start entering the characters of your search pattern, the search starts immediately. If there is a match in the text this is highlighted and the background color of the entry field changes to light green. If the search pattern does not match any string in the text, this is indicated by a light red background color of the entry field.

Use the  or  button to jump to the next or previous match in the document.

Matches in the document are highlighted even when you close the search bar. To clear this highlighting, press the **Esc** key.

You can choose whether the search should be case sensitive. Selecting  will limit finds to entries that match the case (upper or lower) of each of the characters in the search pattern.

Click on the  button at the right side of the incremental search bar to switch to the power search and replace bar.

To repeat the last find operation, if any, without calling the incremental search bar, use **Edit** → **Find Next (F3)** or **Edit** → **Find Previous (Shift+F3)**.



3.5.3 Replacing Text

To replace text, launch the power search and replace bar using the **Edit** → **Replace** command, or the **Ctrl+R** shortcut.



On the upper left side of the bar is a button with an icon to close the bar, followed by a small combo box for entering the search pattern. The box remembers recently used patterns.


You can control the search mode by selecting the options **Plain Text**, **Whole Words**, **Escape Sequences** or **Regular Expression** from the drop down box.

If **Escape sequences** or **Regular expression** are selected, the **Add...** menuitem at the bottom of the context menu of the text boxes will be enabled and allows you to add escape sequences or regular expression items to the search or replace pattern from predefined lists.

Use the  or  button to jump to the next or previous match in the document.

Enter the text to replace with in the text box labeled **Replace** and click the **Replace** button to replace only the highlighted text or the **Replace All** button to replace the search text in the whole document.

You can modify the search and replace behavior by selecting different options at the bottom of the bar. Selecting  will limit finds to entries that match the case (upper or lower) of each of the characters in the search pattern.  will search and replace within the current selection only. The **Find All** button highlights all matches in the document and shows the number of found matches in a small popup.

Click on the  button at the right side of the power search and replace bar to switch to the incremental search bar.

TIP

If you are using a regular expression to find the text to replace, you can employ backreferences to reuse text captured in parenthesized subpatterns of the expression. See [Regular Expressions](#) for more on those.

TIP

You can do **find**, **replace** and **ifind** (incremental search) from the [command line](#).

3.6 Using Bookmarks

The bookmarks feature allows you to mark certain lines, to be able to easily find them again.

You can set or remove a bookmark in a line in two ways:

- Move the insertion cursor to the line and activate the **Bookmarks** → **Set Bookmark (Ctrl+B)** command.
- Click in the Icon Border next to the line.

Bookmarks are available in the **Bookmarks** menu. The individual bookmarks are available as menu items, labeled with the line number of the line with the bookmark, and the first few characters of the text in the line. To move the insertion cursor to the beginning of a bookmarked line, open the menu and select the bookmark.

To quickly move between bookmarks or to the next/previous bookmark, use the **Bookmarks** → **Next (Alt+PgDn)** or **Bookmarks** → **Previous (Alt+PgUp)** commands.

3.7 Automatically Wrapping text

This feature allows you to have the text formatted in a very simple way: the text will be wrapped, so that no lines exceed a maximum number of characters per line, unless there is a longer string of non-whitespace characters.

To enable/disable it, check/uncheck the **Static Word Wrap** checkbox in the [edit page](#) of the [configuration dialog](#).

To set the maximum line width (maximum characters per line), use the **Wrap Words At** option in the [Editing page](#) of the [configuration dialog](#).

If enabled, it has the following effects:

- While typing, the editor will automatically insert a hard line break after the last whitespace character at a position before the maximum line width is reached.
- While loading a document, the editor will wrap the text in a similar way, so that no lines are longer than the maximum line width, if they contain any whitespace allowing that.

NOTE

There is currently no way to set word wrap for document types, or even to enable or disable the feature on a per document level. This will be fixed in a future version of KatePart.

3.8 Using automatic indenting

KatePart's editor component supports a variety of autoindenting modes, designed for different text formats. You can pick from the available modes using the **Tools** → **Indentation** menu. The autoindent modules also provide a function **Tools** → **Format Indentation** which will recalculate the indentation of the selected or current line. Thus, you may reindent your entire document by selecting all the text and activating that action.

All the indent modes use the indentation related settings in the active document.

TIP

You can set all sorts of configuration variables, including those related to indentation using [Document Variables](#) and [File types](#).

AVAILABLE AUTOINDENT MODES

None

Selecting this mode turns automatic indenting off entirely.

Normal

This indenter simply keeps the indentation similar to the previous line with any content other than whitespace. You can combine this with using the indent and unindent actions for indenting to your own taste.

C Style

An indenter for C and similar languages, such as C++, C#, Java™, JavaScript and so on. This indenter will not work with scripting languages such as Perl or PHP.

Haskell

An indenter for the functional programming language Haskell.

Lilypond

An indenter for the Lilypond notation language for music.

Lisp

An indenter specifically for the Lisp scripting language and Lisp dialects.

Python

An indenter specifically for the python scripting language.

XML Style

An indenter specifically for XML like languages.

3.9 Line Modification Indicators

KatePart's line modification indicators let you easily see what you have recently changed in a file. By default, saved changes are indicated by a green bar to the left of a document, while unsaved changes are indicated by an orange bar.

```

// new block for empty buffer I write new stuff HERE
andNewStuffHere = 1;
if (newLineChangedAgain)
    lala ();
more = 1;
afterSaveEdited ();

TextBlock *newBlock = new TextBlock (this, 0);
newBlock->appendLine (TextLine (new TextLineData()));

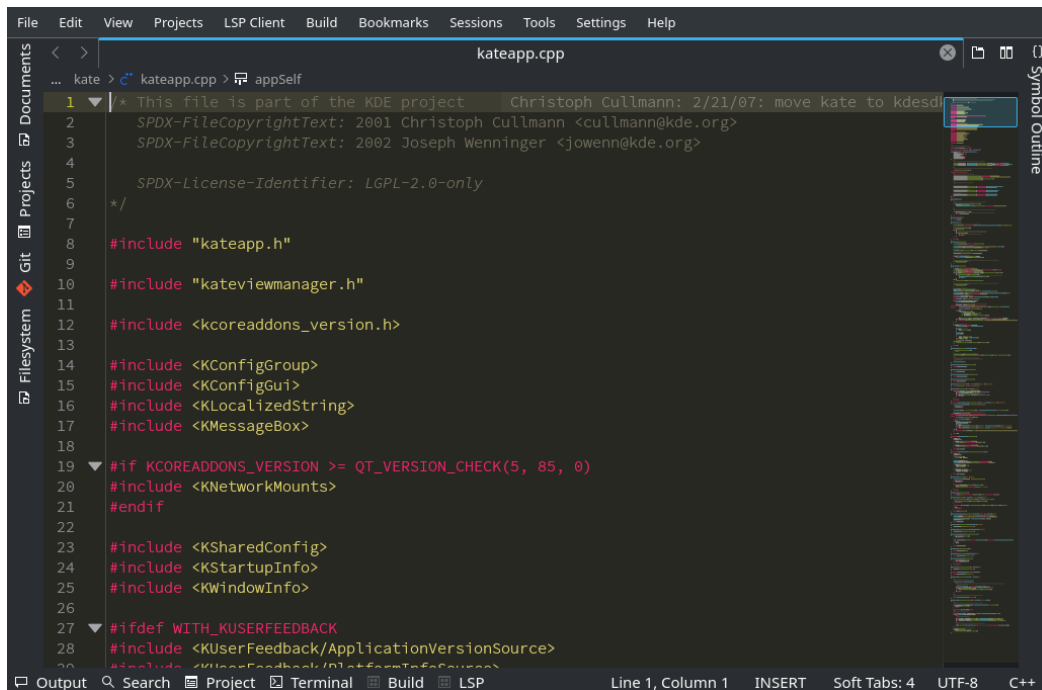
```

Line Modification Indicators in action.

You can change the colors used in the [Fonts & Colors](#) configuration panel, or you can disable this feature completely in the [Borders](#) tab of the [Appearance](#) configuration panel.

3.10 The Scrollbar Minimap

KatePart's Scrollbar Minimap displays a preview of documents in place of the scrollbar. The currently visible portion of the document is highlighted.



The Scrollbar Minimap shows a preview of the Kate source code.

You can temporarily enable or disable the minimap by selecting **View** → **View Scrollbar Minimap** or permanently in the [Appearance](#) section of KatePart's configuration.

3.11 Multiple cursors

Multiple cursor support was introduced with version 5.93 of katepart.

3.11.1 Creating multiple cursors

- To create them via mouse, use **Alt** + left mouse button. The modifier is configurable, see [Configure multicursor modifier](#)
- To create via keyboard, press **Ctrl+Alt+Up** to create cursor above primary cursor and **Ctrl+Alt+Down** to create cursor below. These shortcuts are also configurable
- To create cursors out of a selection, first select some text and then press **Shift+Alt+I**. This will create a cursor at the end of each line in selection.
- Use **Alt+J** to find the next occurrence of the word under cursor and select it + create a cursor. If you want to skip the current word under cursor, press **Alt+K** and it will mark the currently selected word as skipped. When you press **Alt+J** again it will unselect the current word and move to the next word.
- Use **Ctrl+Alt+Shift+J** to find all occurrences of the word under cursor and select them with a cursor at the end of each selection. You can use **Alt+J** to cycle through the selected words and use **Alt+K** to unselect any word as mentioned in the previous paragraph.

3.11.2 Working with multiple cursors

Once you have created a few cursors, you can perform most of the editing operations on them as you would on a single cursor. For example, typing a letter will type it for each cursor. Similarly you can perform text transforms e.g., capitalization for all the positions or selections.

Sometimes you will want to remove cursors. To do so, you can **Alt** + left mouse button on the cursor that you want to remove. If you just want to remove cursors on lines that are empty, there is a ready-made action for it that will do it for you. To invoke the action, open the Command Bar using **Ctrl+Alt+I** and look for **Remove cursors from empty lines** and hit **Enter**. You can also configure a shortcut for this action.

Chapter 4

The Menu Entries

4.1 The File Menu

File → New (Ctrl+N)

This starts a new document in a new and independent editor window.

File → Open... (Ctrl+O)

Displays a standard KDE **Open File** dialog. Use the file view to select the file you want to open, and click on **Open** to open it.

File → Open Recent

This is a shortcut to open recently saved documents. Clicking on this item opens a list to the side of the menu with several of the most recently saved files. Clicking on a specific file will open it in KatePart - if the file still resides at the same location.

File → Save (Ctrl+S)

This saves the current document. If there has already been a save of the document then this will overwrite the previously saved file without asking for the user's consent. If it is the first save of a new document the save as dialog (described below) will be invoked.

File → Save As... (Ctrl+Shift+S)

This allows a document to be saved with a new file name. This is done by means of the file dialog box described above in the [Open](#) section of this help file.

File → Save As with Encoding

Save a document with a new file name in a different encoding.

File → Save Copy As

Save a copy of the document with a new file name and continue editing the original document.

File → Reload (F5)

Reloads the active file from disk. This command is useful if another program or process has changed the file while you have it open in KatePart.

File → Print... (Ctrl+P)

Opens a simple print dialog allowing the user to specify what, where, and how to print.

File → Export as HTML

Save the currently open document as an HTML file, which will be formatted using the current syntax highlighting and color scheme settings.

File → Close (Ctrl+W)

Close the active file with this command. If you have made unsaved changes, you will be prompted to save the file before KatePart closes it.

File → Quit (Ctrl+Q)

This will close the editor window, if you have more than one instance of KatePart running, through the **New** or **New Window** menu items, those instances will not be closed.

4.2 The Edit Menu

Edit → Undo (Ctrl+Z)

Undo the last editing command (typing, copying, cutting etc.)

NOTE

This may undo several editing commands of the same type, like typing in characters.

Edit → Redo (Ctrl+Shift+Z)

This will reverse the most recent change (if any) made using Undo.

Edit → Cut (Ctrl+X)

This command deletes the current selection and places it on the clipboard. The clipboard works invisibly and provides a way to transfer data between applications.

Edit → Copy (Ctrl+C)

This copies the currently selected text to the clipboard so that it may be pasted elsewhere. The clipboard works invisibly and provides a way to transfer data between applications.

Edit → Paste (Ctrl+V)

This will insert the first item in the clipboard at the cursor position. The clipboard works invisibly and provides a way to transfer data between applications.

NOTE

If Overwrite Selection is enabled, the pasted text will overwrite the selection, if any.

Edit → Paste Selection (Ctrl+Shift+Ins)

This will paste the [mouse selection](#) contents that were chosen previously. Mark some text with the mouse pointer to paste it in the currently open file using this menu item.

Edit → Swap with clipboard contents

This will swap the selected text with the [clipboard](#) contents.

Edit → Clipboard History

This submenu will display the beginning of portions of text recently copied to the clipboard. Select an item from this menu to paste it in the currently open file.

Edit → Copy as HTML

Copy the selection as HTML, formatted using the current syntax highlighting and color scheme settings.

Edit → Input Modes

Switch between a normal and a vi-like, modal editing mode. The vi input mode supports the most used commands and motions from vim's normal and visual mode and has an optional vi mode statusbar. This status bar shows commands while they are being entered, output from commands and the current mode. The behavior of this mode can be configured in the [Vi Input Mode](#) tab of the **Editing** page in KatePart's settings dialog.



Edit → Overwrite Mode (Ins)

Toggles the Insert/Overwrite modes. When the mode is **INS**, you insert characters where the cursor is. When the mode is **OVR**, writing characters will replace the current characters if your cursor is positioned before any character. The status bar shows the current state of the Overwrite Mode, either **INS** or **OVR**.


Edit → Find... (Ctrl+F)


This opens the incremental search bar at the bottom of the editor window. On the left side of the bar is a button with an icon to close the bar, followed by a small text box for entering the search pattern.

When you start entering characters of your search pattern, the search starts immediately. If there is a match in the text this is highlighted and the background color of the entry field changes to light green. If the search pattern does not match any string in the text, this is indicated by a light red background color of the entry field.

Use the  or  button to jump to the next or previous match in the document.

Matches in the document are highlighted even when you close the search bar. To clear this highlighting, press the **Esc** key.

You can choose whether the search should be case sensitive. Selecting  will limit finds to entries that match the case (upper or lower) of each of the characters in the search pattern.

Click on the  button at the right side of the incremental search bar to switch to the power search and replace bar.

Edit → Find Variants → Find Next (F3)

This repeats the last find operation, if any, without calling the incremental search bar, and searching forwards through the document starting from the cursor position.

Edit → Find Variants → Find Previous (Shift+F3)

This repeats the last find operation, if any, without calling the incremental search bar, and searching backwards instead of forwards through the document.

Edit → Find Variants → Find Selected (Ctrl+H)

Finds next occurrence of selected text.

Edit → Find Variants → Find Selected Backwards (Ctrl+Shift+H)



Finds previous occurrence of selected text.

Edit → Replace... (Ctrl+R)


This command opens the power search and replace bar. On the upper left side of the bar is a button with an icon to close the bar, followed by a small text box for entering the search pattern.


You can control the search mode by selecting **Plain text**, **Whole words**, **Escape sequences** or **Regular expression** from the drop down box.


If **Escape sequences** or **Regular expression** are selected, the **Add...** menuitem at the bottom of the context menu of the text boxes will be enabled and allows you to add escape sequences or regular expression items to the search or replace pattern from predefined lists.

Use the  or  button to jump to the next or previous match in the document.

Enter the text to replace with in the text box labeled **Replace** and click the **Replace** button to replace only the highlighted text or the **Replace All** button to replace the search text in the whole document.

You can modify the search and replace behavior by selecting different options at the bottom of the bar. Selecting  will limit finds to entries that match the case (upper or lower)

of each of the characters in the search pattern.  will search and replace within the current selection only. The **Find All** button highlights all matches in the document and shows the number of found matches in a small popup.

Click on the  button at the right side of the power search and replace bar to switch to the incremental search bar.

Edit → Go To → Go to Matching Bracket (Ctrl+6)

Move the cursor to the associated opening or closing bracket.

4.3 The Selection Menu

Selection → Select All (Ctrl+A)

This will select the entire document. This could be very useful for copying the entire file to another application.

Selection → Deselect (Ctrl+Shift+A)

Deselects the selected text in the editor if any.

Selection → Block Selection Mode (Ctrl+Shift+B)

Toggles Selection Mode. When the Selection Mode is **BLOCK**, the status bar contains the string **[BLOCK]** and you can make vertical selections, e.g. select column 5 to 10 in lines 9 to 15.

Selection → Comment (Ctrl+D)

This adds one space to the beginning of the line where the text cursor is located or to the beginning of any selected lines.

Selection → Uncomment (Ctrl+Shift+D)

This removes one space (if any exist) from the beginning of the line where the text cursor is located or from the beginning of any selected lines.

Selection → Join Lines (Ctrl+J)

Joins the selected lines, or the current line and the line below with one white space character as a separator. Leading/trailing white space on joined lines is removed in the affected ends.

Selection → Capitalization (Ctrl+Alt+U)

Capitalize the selected text or the current word.

Selection → Uppercase (Ctrl+U)

Put the selected text or the letter after the cursor in uppercase.

Selection → Lowercase (Ctrl+Shift+U)

Put the selected text or the letter after the cursor in lowercase.

Selection → Clean Indentation

This cleans the indentation for the current selection or for the line the cursor is currently in. Cleaning the indentation ensures that all your selected text follows the indentation mode you choose.

Selection → Format Indent

Causes a realign of the current line or selected lines using the indentation mode and indentation settings in the document.

Selection → Align On...

This command aligns lines in the selected block or whole document on the column given by a regular expression that you will be prompted for.

If you give an empty pattern it will align on the first non-blank character by default.

If the pattern has a capture it will indent on the captured match.

Examples:

With `'-'` it will insert spaces before the first `'-'` of each lines to align them all on the same column.

With `'alignon :\\s+(.)'` it will insert spaces before the first non-blank character that occurs after a colon to align them all on the same column.

Selection → Apply Word Wrap

Apply static word wrapping on all the document. That means that a new line of text will automatically start when the current line exceeds the length specified by the **Wrap words at** option in the Editing tab in **Settings → Configure Editor...** menu.

4.4 The View Menu

View → New Window

Create another window containing the current document. All changes to the document in one window are reflected in the other window and vice versa.

View → Switch to Command Line (F7)

Displays the KatePart command line at the bottom of the window. In the command line, type **help** to get help and **help list** to get a list of commands. For more information on the command line, see [The Editor Component Command Line](#).

View → Enlarge Font (Ctrl++)

This increases the display font size.

View → Shrink Font (Ctrl+-)

This decreases the display font size.

View → Word Wrap → Dynamic Word Wrap (F10)

Toggles dynamic word wrap in the current view. Dynamic word wrap makes all the text in a view visible without the need for horizontal scrolling by rendering one actual line on more visual lines as needed.

View → Word Wrap → Dynamic Word Wrap Indicators

Choose when and how the dynamic word wrap indicators should be displayed. This is only available if the **Dynamic Word Wrap** option is checked.

View → Word Wrap → Show Static Word Wrap Marker

If this option is checked, a vertical line will be drawn at the word wrap column as defined in the **Settings → Configure Editor...** in the Editing tab. Please note that the word wrap marker is only drawn if you use a fixed pitch font.

View → Borders → Show Icon Border (F6)

This is a toggle item. Setting it on checked will make the Icon Border visible in the left side of the active editor, and vice versa. The Icon Border indicates the positions of the marked lines in the editor.

View → Borders → Show Line Numbers (F11)

This is a toggle Item. Setting it on checked will make a pane displaying the line numbers of the document visible in the left border of the active editor, and vice versa.

View → Borders → Show Scrollbar Marks

If this option is checked, the view will show marks on the vertical scrollbar. The marks are equivalent to the marks on the [Icon Border](#).

View → Borders → Show Scrollbar Mini-Map

This will replace the scrollbar with a visualization of the current document. For more information on the scrollbar minimap, see [Section 3.10](#).

View → Code Folding

These options pertain to [code folding](#):

Show Folding Markers (F9)

Toggles the display of the folding marker pane in the left side of the view.

Fold Current Node

Collapse the region that contains the cursor.

Unfold Current Node

Expand the region that contains the cursor.

Fold Toplevel Nodes (Ctrl+Shift+-)

Collapse all toplevel regions in the document. Click on the right pointing triangle to expand all toplevel regions.

Unfold Toplevel Nodes (Ctrl+Shift++)

Expand all toplevel regions in the document.

Show Non-Printable Spaces

Show/hide bounding box around non-printable spaces.

4.5 The Go Menu

Go → Go to Line... (Ctrl+G)

This opens the goto line bar at the bottom of the window which is used to have the cursor jump to a particular line (specified by number) in the document. The line number may be entered directly into the text box or graphically by clicking on the up or down arrow spin controls at the side of the text box. The little up arrow will increase the line number and the down arrow decrease it. Close the bar with a click on the button with an icon on the left side of the bar.

Go → Go to Previous Editing Line (Ctrl+E)

This action jumps the previous editing line in the [multicursor configuration](#).

Go → Go to Next Editing Line (Ctrl+Shift+E)

This action jumps the next editing line in the [multicursor configuration](#).

Go → Go to Previous Modified Line

Lines that were changed since opening the file are called modified lines. This action jumps the previous modified line.

Go → Go to Next Modified Line

Lines that were changed since opening the file are called modified lines. This action jumps the next modified line.

Go → Select to Matching Bracket (Ctrl+Shift+6)

Selects the text between associated opening and closing brackets.

Go → Bookmarks (Ctrl+Shift+6)

Below the entries described here, one entry for each bookmark in the active document will be available. The text will be the first few words of the marked line. Choose an item to move the cursor to the start of that line. The editor will scroll as necessary to make that line visible.

Go → Bookmarks → Set Bookmark (Ctrl+B)

Sets or removes a bookmark in the current line of the active document. (If it's there, it is removed, otherwise one is set.)

Go → Bookmarks → Clear All Bookmarks

This command will remove all the markers from the document as well as the list of markers which is appended at the bottom of this menu item.

Go → Bookmarks → Previous (Alt+PgUp)

This will move the cursor to beginning of the first above line with a bookmark. The menuitem text will include the line number and the first piece of text on the line. This item is only available when there is a bookmark in a line above the cursor.

Go → Bookmarks → Next (Alt+PgDn)

This will move the cursor to beginning of the next line with a bookmark. The menuitem text will include the line number and the first piece of text on the line. This item is only available when there is a bookmark in a line below the cursor.

4.6 The Tools Menu

Tools → Read Only Mode

Set the current document to Read Only mode. This prevents any text addition and any changes in the document formatting.

Tools → Mode

Choose the filetype scheme you prefer for the active document. This overwrites the global filetype mode set in **Settings → Configure Editor...** in the Filetypes tab for your current document only.

Tools → Highlighting

Choose the Highlighting scheme you prefer for the active document. This overwrites the global highlighting mode set in **Settings → Configure Editor...** for your current document only.

Tools → Indentation

Choose the style of indentation you want for your active document. This overwrites the global indentation mode set in **Settings → Configure Editor...** for your current document only.

Tools → Encoding

You can overwrite the default encoding set in **Settings → Configure Editor...** in the **Open/Save** page to set a different encoding for your current document. The encoding you set here will be only valid for your current document.

Tools → End of Line

Choose your preferred end of line mode for your active document. This overwrites the global end of line mode set in **Settings → Configure Editor...** for your current document only.

Tools → Add Byte Mark Order (BOM)

Checking this action you can explicitly add a byte order mark for unicode encoded documents. The byte order mark (BOM) is a Unicode character used to signal the endianness (byte order) of a text file or stream, for more information see [Byte Order Mark](#).

Tools → Scripts

This submenu contains a list of all scripted actions. The list can easily be modified by [writing your own scripts](#). This way, KatePart can be extended with user-defined tools.

Tools → Scripts → Navigation

Tools → Scripts → Navigation → Move cursor to previous matching indent (Alt+Shift+Up)

Moves the cursor to the first line above the current line that is indented at the same level as the current line.

Tools → Scripts → Navigation → Move cursor to next matching indent (Alt+Shift+Down)

Moves the cursor to the first line below the current line that is indented at the same level as the current line.

Tools → Scripts → Editing

Tools → Scripts → Editing → Sort Selected Text

Sorts the selected text or whole document in ascending order.

Tools → Scripts → Editing → Move Lines Down (Ctrl+Shift+Down)

Move selected lines down.

Tools → Scripts → Editing → Move Lines Up (Ctrl+Shift+Up)

Move selected lines up.

Tools → Scripts → Editing → Duplicate Selected Lines Down (Ctrl+Alt+Down)

Duplicates the selected lines down.

Tools → Scripts → Editing → Duplicate Selected Lines Up (Ctrl+Alt+Up)

Duplicates the selected lines up.

Tools → Scripts → Editing → URI-encode selected text

Encodes the selected text so that it can be used as part of a query string in a URL, replacing the selection with the encoded text.

Tools → Scripts → Editing → URI-decode selected text

If part of the query string of a URL is selected, this will decode it and replace the selection with the original raw text.

Tools → Scripts → Emmet

Tools → Scripts → Emmet → Expand abbreviation

Converts the selected text to a pair of opening and closing HTML or XML tags. For example, if **div** is selected, this item will replace that with `<div></div>`.

Tools → Scripts → Emmet → Wrap with tag

Wraps the selected text with the tag provided on the [command line](#).

Tools → Scripts → Emmet → Move cursor to matching tag

If the cursor is inside an opening HTML/XML tag, this item will move it to the closing tag. If the cursor is inside the closing tag, it will instead move it to the opening tag.

Tools → Scripts → Emmet → Select HTML/XML tag contents inwards

When the cursor is inside a pair of HTML/XML tags, this option will change the selection to include the contents of those HTML/XML tags, without selecting the tags themselves.

Tools → Scripts → Emmet → Select HTML/XML tag contents outwards

When the cursor is inside a pair of HTML/XML tags, this item will change the selection to include the contents of those HTML/XML tags, including the tags themselves.

Tools → Scripts → Emmet → Toggle Comment

If the selected portion is not a comment, this item will enclose that portion in HTML/XML comments (e.g. `<!-- selected text -->`). If the selected portion is a comment, the comment tags will be removed instead.

Tools → Scripts → Emmet → Delete tag under cursor

If the cursor is presently inside a HTML/XML tag, this item will delete the entire tag.

Tools → Scripts → Emmet → Decrement number by 1

This item will subtract one from the currently selected text, if it is a number. For example, if **5** is selected, it will become 4.

Tools → Scripts → Emmet → Decrement number by 10

This item will subtract 10 from the currently selected text, if it is a number. For example, if **15** is selected, it will become 5.

Tools → Scripts → Emmet → Decrement number by 0.1

This item will subtract 0.1 from the currently selected text, if it is a number. For example, if **4.5** is selected, it will become 4.4.

Tools → Scripts → Emmet → Increment number by 1

This item will add one to the currently selected text, if it is a number. For example, if **5** is selected, it will become 6.

Tools → Scripts → Emmet → Increment number by 10

This item will add 10 to the currently selected text, if it is a number. For example, if **5** is selected, it will become 15.

Tools → Scripts → Emmet → Increment number by 0.1

This item will add 0.1 to the currently selected text, if it is a number. For example, if **4.5** is selected, it will become 4.6.

Tools → Invoke Code Completion (Ctrl+Space)

Manually invoke command completion, usually by using a shortcut bound to this action.

Tools → Word Completion

Reuse Word Below (Ctrl+9) and **Reuse Word Above (Ctrl+8)** complete the currently typed text by searching for similar words backward or forward from the current cursor position. **Shell Completion** pops up a completion box with matching entries.

Tools → Spelling → Automatic Spell Checking (Ctrl+Shift+O)

When **Automatic Spell Checking** is enabled, wrongly spelled text is underlined in the document on-the-fly.

Tools → Spelling → Spelling...

This initiates the spellchecking program - a program designed to help the user catch and correct any spelling errors. Clicking on this entry will start the checker and bring up the speller dialog box through which the user can control the process. There are four settings lined up vertically in the center of the dialog with their corresponding labels just to the left. Starting at the top they are:

Unknown word:

Here, the spellchecker indicates the word currently under consideration. This happens when the checker encounters a word not in its dictionary - a file containing a list of correctly spelled words against which it compares each word in the editor.

Replace with:

If the checker has any similar words in its dictionary the first one will be listed here. The user can accept the suggestion, type in his or her own correction, or choose a different suggestion from the next box.

Language:

If you have installed multiple dictionaries, here you can select which dictionary/language should be used.

On the right side of the dialog box are 6 buttons that allow the user to control the spellcheck process. They are:

Add to Dictionary

Pressing this button adds the **Unknown word** to the checker's dictionary. This means that in the future the checker will always consider this word to be correctly spelled.

Suggest

The checker may list here a number of possible replacements for the word under consideration. Clicking on any one of the suggestions will cause that word to be entered in the **Replace with** box, above.

Replace

This button has the checker replace the word under consideration in the document with the word in the **Replace with** box.

Replace All

This button causes the checker to replace not only the current **Unknown word:** but to automatically make the same substitution for any other occurrences of this **Unknown word** in the document.

Ignore

Activating this button will have the checker move on without making any changes.

Ignore All

This button tells the checker to do nothing with the current **Unknown word:** and to pass over any other instances of the same word.

NOTE
This only applies to the current spellcheck run. If the checker is run again later it will stop on this same word.

Three more buttons are located horizontally along the bottom of the spellcheck dialog. They are:

Help

This invokes the KDE help system with the help page for this dialog.

Finished

This button ends the spellcheck process, and returns to the document.

Cancel

This button cancels the spellcheck process, all modifications are reverted, and you will return to your document.

Tools → Spelling → Spelling (from cursor)...

This initiates the spellchecking program but it starts where your cursor is instead of at the beginning of the document.

Tools → Spelling → Spellcheck Selection...

Spellchecks the current selection.

Tools → Spelling → Change Dictionary

Displays a drop down box with all available dictionaries for spellchecking at the bottom of the editor window. This allows easy switching of the spellcheck dictionary e.g. for automatic spellcheck of text in different languages.

4.7 The Settings and Help Menu

Settings → Editor Color Theme

This menu lists the available color schemes. You can change the schema for the current view here, to change the default schema you need to use the [Fonts & Colors](#) page of the config dialog.

KatePart has the common KDE **Settings** and **Help** menu items, for more information read the sections about the [Settings Menu](#) and [Help Menu](#) of the KDE Fundamentals.

Chapter 5

Advanced Editing Tools

Anders Lund
Dominik Haumann

5.1 Comment/Uncomment

The **Comment** and **Uncomment** commands, available from the **Tools** menu allow you to add or remove comment markers to the selection, or the current line if no text is selected, if comments are supported by the format of the text you are editing.

The rules for how commenting is done are defined in the syntax definitions, so if syntax highlighting is not used, commenting/uncommenting is not possible.

Some formats define single line comment markers, some multiline markers and some both. If multiline markers are not available, commenting out a selection that does not fully include its last line is not possible.

If a single line marker is available, commenting single lines is preferred where applicable, as this helps to avoid problems with nested comments.

When removing comment markers, no uncommented text should be selected. When removing multiline comment markers from a selection, any whitespace outside the comment markers is ignored.

To place comment markers, use the **Tools** → **Comment** menu item or the related keyboard shortcut sequence, the default is **Ctrl+D**.

To remove comment markers, use the **Tools** → **Uncomment** menu item or the related keyboard shortcut, the default is **Ctrl+Shift+D**.

5.2 The Editor Component Command Line

KatePart's editor component has an internal command line, allowing you to perform various actions from a minimal GUI. The command line is a text entry at the bottom of the editor area; to show it select **View** → **Switch to Command Line** or use the shortcut (default is **F7**). The editor provides a set of commands as documented below, and additional commands can be provided by plugins.

To execute a command, type the command then press the return key. The command line will indicate whether it succeeded and possibly display a message. If you entered the command line

by pressing **F7** it will automatically hide after a few seconds. To clear the message and enter a new command, press **F7** again.

The command line has a built-in help system; issue the command **help** to get started. To see a list of all available commands issue **help list**; to view help for a specific command, do **help *command***.

The command line has a built in history, so you can reuse commands already typed. To navigate the history, use the **Up** and **Down** keys. When showing historical commands, the argument part of the command will be selected, allowing you to easily overwrite the arguments.

5.2.1 Standard Command Line Commands

ARGUMENT TYPES

BOOLEAN

This is used with commands that turns things on or off. Legal values are **on**, **off**, **true**, **false**, **1** or **0**.

INTEGER

An integer number.

STRING

A string, surrounded by single quotes (') or double quotes (") when it contains spaces.

5.2.1.1 Commands for Configuring the Editor

These commands are provided by the editor component, and allow you to configure the active document and view only. This is handy if you want to use a setting different from the default settings, for example for indentation.

set-tab-width INTEGER *width*

Sets the tab width to the number **width**.

set-indent-width INTEGER *width*

Sets the indentation width to the number **width**. Used only if you are indenting with spaces.

set-word-wrap-column INTEGER *width*

Sets the line width for hard wrapping to **width**. This is used if you are having your text wrapped automatically.

set-icon-border BOOLEAN *enable*

Sets the visibility of the icon border.

set-folding-markers BOOLEAN *enable*

Sets the visibility of the folding markers pane.

set-line-numbers BOOLEAN *enable*

Sets the visibility of the line numbers pane.

set-replace-tabs BOOLEAN *enable*

If enabled, tabs are replaced with spaces as you type.

set-remove-trailing-space BOOLEAN *enable*

If enabled, trailing whitespace is removed whenever the cursor leaves a line.

set-show-tabs **BOOLEAN enable**

If enabled, TAB characters and trailing whitespace will be visualized by a small dot.

set-show-indent **BOOLEAN enable**

If enabled, indentation will be visualized by a vertical dotted line.

set-indent-spaces **BOOLEAN enable**

If enabled, the editor will indent with `indent-width` spaces for each indentation level, rather than with one TAB character.

set-mixed-indent **BOOLEAN enable**

If enabled, KatePart will use a mix of TAB and spaces for indentation. Each indentation level will be `indent-width` wide, and more indentation levels will be optimized to use as many TAB characters as possible.

When executed, this command will additionally set space indentation enabled, and if the indent width is unspecified it will be set to half of the `tab-width` for the document at the time of execution.

set-word-wrap **BOOLEAN enable**

Enables dynamic word wrap according to **enable**.

set-replace-tabs-save **BOOLEAN enable**

When enabled, tabs will be replaced with whitespace whenever the document is saved.

set-remove-trailing-space-save **BOOLEAN enable**

When enabled, trailing space will be removed from each line whenever the document is saved.

set-indent-mode **STRING name**

Sets the autoindentation mode to **name**. If **name** is not known, the mode is set to 'none'. Valid modes are 'none', 'normal', 'cstyle', 'haskell', 'lilypond', 'lisp', 'python', 'ruby' and 'xml'.

set-auto-indent **BOOLEAN script**

Enable or disable autoindentation.

set-highlight **STRING highlight**

Sets the syntax highlighting system for the document. The argument must be a valid highlight name, as seen in the **Tools** → **Highlighting** menu. This command provides an auto-completion list for its argument.

reload-scripts

Reload all [JavaScript scripts](#) used by Kate, including indenters and command line scripts.

set-mode **STRING mode**

Choose the filetype scheme for the current document.

nn[oremap] **STRING original** **STRING mapped**

Map the key sequence **original** to **mapped**.

5.2.1.2 Commands for editing

These commands modify the current document.

indent

Indents the selected lines or the current line.

unindent

Unindents the selected lines or current line.

cleanindent

Cleans up the indentation of the selected lines or current line according to the indentation settings in the document.

comment

Inserts comment markers to make the selection or selected lines or current line a comment according to the text format as defined by the syntax highlight definition for the document.

uncomment

Removes comment markers from the selection or selected lines or current line according to the text format as defined by the syntax highlight definition for the document.

kill-line

Deletes the current line.

replace STRING pattern STRING replacement

Replaces text matching **pattern** with **replacement**. If you want to include whitespace in the **pattern**, you must quote both the **pattern** and **replacement** with single or double quotes. If the arguments are unquoted, the first word is used as **pattern** and the rest for **replacement**. If **replacement** is empty, each occurrence of **pattern** is removed.

You can set flags to configure the search by adding a colon, followed by one or more letters each representing a configuration, giving the form **replace:options pattern replacement**. Available options are:

- b** Search backwards.
- c** Search from cursor position.
- e** Search in the selection only.
- r** Do regular expression search. If set, you may use **\N** where N is a number to represent captures in the replacement string.
- s** Do case sensitive search.
- p** Prompt for permission to replace the next occurrence.
- w** Match whole words only.

date STRING format

Inserts a date/time string as defined by the specified **format**, or the format 'yyyy-MM-dd hh:mm:ss' if none is specified. The following translations are done when interpreting **format**:

d	The day as number without a leading zero (1-31).
dd	The day as number with a leading zero (01-31).
ddd	The abbreviated localized day name (e.g. 'Mon'..'Sun').
dddd	The long localized day name (e.g. 'Monday'..'Sunday').
M	The month as number without a leading zero (1-12).

MM	The month as number with a leading zero (01-12).
MMMM	The long localized month name (e.g. 'January'..'December').
MMM	The abbreviated localized month name (e.g. 'Jan'..'Dec').
YY	The year as two digit number (00-99).
YYYY	The year as four digit number (1752-8000).
h	The hour without a leading zero (0..23 or 1..12 if AM/PM display).
hh	The hour with a leading zero (00..23 or 01..12 if AM/PM display).
m	The minute without a leading zero (0..59).
mm	The minute with a leading zero (00..59).
s	The second without a leading zero (0..59).
ss	The second with a leading zero (00..59).
z	The milliseconds without leading zeroes (0..999).
zzz	The milliseconds with leading zeroes (000..999).
AP	Use AM/PM display. AP will be replaced by either "AM" or "PM".
ap	Use am/pm display. ap will be replaced by either "am" or "pm".

char STRING identifier

This command allows you to insert literal characters by their numerical identifier, in decimal, octal or hexadecimal form. To use it launch the Editing Command dialog and type **char: [number]** in the entry box, then hit **OK**.

Example 5.1 char examples

Input: **char:234**
 Output: ê
 Input: **char:0x1234**
 Output: jué

s///[ig] %s///[ig]

This command does a sed-like search/replace operation on the current line, or on the whole file (%s///).

In short, the text is searched for text matching the *search pattern*, the regular expression between the first and the second slash, and when a match is found, the matching part of the text is replaced with the expression between the second and last slash. Parentheses in the search pattern create *back references*, that is the command remembers which part of the string matched in the parentheses; these strings can be reused in the replace pattern, referred to as \1 for the first set of parentheses, \2 for the second and so on.

To search for a literal (or), you need to *escape* it using a backslash character: \(\)

If you put an **i** at the end of the expression, the matching will be case insensitive. If you put a **g** at the end, all occurrences of the pattern will be replaced, otherwise only the first occurrence is replaced.

Example 5.2 Replacing text in the current line

Your friendly compiler just stopped, telling you that the class `myClass` mentioned in line 3902 in your source file is not defined.

"Buckle!" you think, it is of course `MyClass`. You go to line 3902, and instead of trying to find the word in the text, you launch the Editing Command Dialog, enter `s/myclass/MyClass/i`, hit the **OK** button, save the file and compile – successfully without the error.

Example 5.3 Replacing text in the whole file

Imagine that you have a file, in which you mention a 'Miss Jensen' several times, when someone comes in and tells you that she just got married to 'Mr Jones'. You want, of course, to replace each and every occurrence of 'Miss Jensen' with 'Ms Jones'.

Enter the command line and issue the command `%s/Miss Jensen/Ms Jones/` and hit return, you are done.

Example 5.4 A More Advanced Example

This example makes use of *back references* as well as a *character class* (if you do not know what that is, please refer to the related documentation mentioned below).

Suppose you have the following line:

```
void MyClass::DoStringOps( String      &foo, String &bar, String *p, int  & ←  
    a, int &b )
```

Now you realize that this is not nice code, and decide that you want to use the `const` keyword for all 'address of' arguments, those characterized by the `&` operator in front of the argument name. You would also like to simplify the white space, so that there is only 1 whitespace character between each word.

Launch the Editing Command Dialog, and enter: `s/\s+(\w+)\s+(\&)/ const \1 \2/g` and hit the **OK** button. The `g` at the end of the expression makes the regular expression recompile for each match to save the *backreferences*.

Output: `void MyClass::DoStringOps(const String &foo, const String &bar, String *p, const int &a, const int &b)`

Mission completed! Now, what happened? Well, we looked for some white space (`\s+`) followed by one or more alphabetic characters (`\w+`) followed by some more whitespace (`\s+`) followed by an ampersand, and in the process saved the alphabetic chunk and the ampersand for reuse in the replace operation. Then we replaced the matching part of our line with one whitespace followed by 'const' followed by one whitespace followed by our saved alphabetical chunk (`\1`) followed by one whitespace followed by our saved ampersand (`\2`)

Now in some cases the alphabetical chunk was 'String', in some 'int', so using the character class `\w` and the `+` quantifier proved a valuable asset.

sort

Sorts the selected text or entire document.

natsort

Sort the selected lines or entire document naturally.

Example 5.5 sort vs. natsort

`sort (a10, a1, a2)` results in a1, a10, a2

`natsort (a10, a1, a2)` results in a1, a2, a10

moveLinesDown

Move selected lines down.

moveLinesUp

Move selected lines up.

uniq

Remove duplicated lines from the selected text or the whole document.

rtrim

Remove trailing space from the selected text or the whole document.

ltrim

Remove leading space from the selected text or the whole document.

join [STRING separator]

Join selected lines or whole document. Optionally takes a parameter defining a separator, for example: **join ' , '**

rmbblank

Remove all blank spaces from the selected text or the whole document.

alignon

This command aligns lines in the selected block or whole document on the column given by a regular expression given as an argument.

If you give an empty pattern it will align on the first non-blank character by default.

If the pattern has a capture it will indent on the captured match.

Examples:

alignon - will insert spaces before the first '-' of each lines to align them all on the same column.

alignon :\\s+(.) will insert spaces before the first non-blank character that occurs after a colon to align them all on the same column.

unwrap

Unwrap the selected text or the whole document.

each STRING script

Given a JavaScript function as an argument, call that for the list of selected lines and replace them with the return value of that callback.

Example 5.6 Join selected lines

```
each 'function(lines){return lines.join(", ")}'
```

Or, more briefly:

```
each 'lines.join(", ").'
```

filter STRING script

Given a JavaScript function as an argument, call that for the list of selected lines and remove those where the callback returns false.

Example 5.7 Remove blank lines

```
filter 'function(1){return 1.length > 0;}'
```

Or, more briefly:

```
filter 'line.length > 0'
```

map STRING script

Given a JavaScript function as an argument, call that for the list of selected lines and replace the line with the value of the callback.

Example 5.8 Remove blank lines

```
map 'function(line){return line.replace(/^s+/, "");}'
```

Or, more briefly:

```
map 'line.replace(/^s+/, "");'
```

duplicateLinesUp

Duplicate the selected lines above the current selection.

duplicateLinesDown

Duplicate the selected lines below the current selection.

5.2.1.3 Commands for navigation

goto INT line

This command navigates to the specified line.

grep STRING pattern

Search the document for the regular expression **pattern**. For more information, see appendix [A](#).

find STRING pattern

This command navigates to the first occurrence of **pattern** according to the configuration. Following occurrences can be found using **Edit** → **Find Next** (the default shortcut is **F3**).

The find command can be configured by appending a colon followed by one or more options, the form is **find:options pattern**. The following options are supported:

- b** Search backwards.
- c** Search from cursor position.
- e** Search in the selection only.
- r** Do regular expression search. If set, you may use **\N** where N is a number to represent captures in the replacement string.
- s** Do case sensitive search.
- w** Match whole words only.

ifind STRING pattern

This command provides 'as-you-type' searching. You can configure the behavior of the search by appending a colon followed by one or more options, like this: **ifind:options pattern**. Allowed options are:

- b** Search backwards.
- r** Do regular expression search.
- s** Do case sensitive search.
- c** Search from cursor position.

5.2.1.4 Commands for Basic Editor Functions (These depend on the application the editor component is used in)

- w** Save the current document.
- wa** Save all currently open documents.
- q** Close the current document.
- qa** Close all open documents.
- wq** Save and close the current document.
- wqa** Save and close all currently open documents.
- x** Save and close the current document only if it has changed.
- x** Save and close all currently open documents only if they have changed.
- bp** Go to the previous document in the documents list.
- bn** Go to the next document in the documents list.
- new** Open a new document in horizontal split view.
- vnew** Open a new document in vertical split view.
- e** Reload the current document if it has changed on disk.
- enew** Edit a new document.
- print** Open the Print dialog to print the current document.

5.3 Using Code Folding

Code folding allows you to hide parts of a document in the editor, making it easier to overview large documents. In KatePart the foldable regions are calculated using rules defined in the syntax highlight definitions, and therefore it is only available in some formats - typically program source code, XML markup and similar. Most highlight definitions supporting code folding also lets you manually define foldable regions, typically using the **BEGIN** and **END** keywords.

To use the code folding feature, activate the folding markers using **View** → **Show Folding Markers** menu item if they are not already visible. The Folding Markers Pane on the left side of the screen displays a graphical view of the foldable regions, with triangle symbols to indicate the

possible operation on a given region: a top down triangle means that the region is expanded, clicking it will collapse the region and a right pointing triangle will be displayed instead.

Three commands are provided to manipulate the state of folding regions, see the [menu documentation](#).

The folded lines are remembered when a file is closed, so when you reopen the file the folded nodes will still be folded. This applies to reload operations as well.

If you do not want to use the code folding feature, you can disable the **Show folding markers (if available)** option in the [Appearance](#) page of the editor configuration.

Chapter 6

Extending KatePart

T.C. Hollingsworth

6.1 Introduction

Like any advanced text editor component, KatePart offers a variety of ways to extend its functionality. You can [write simple scripts to add functionality with JavaScript](#). Finally, once you have extended KatePart, you are welcome to [join us](#) and share your enhancements with the world!

6.2 Working with Syntax Highlighting

6.2.1 Overview

Syntax Highlighting is what makes the editor automatically display text in different styles/colors, depending on the function of the string in relation to the purpose of the file. In program source code for example, control statements may be rendered bold, while data types and comments get different colors from the rest of the text. This greatly enhances the readability of the text, and thus helps the author to be more efficient and productive.

```
Theme Repository::defaultTheme(Repository::DefaultTheme t)
{
    if (t == DarkTheme)
        return theme(QLatin1String("Breeze Dark"));
    return theme(QLatin1String("Default"));
}
```

A C++ function, rendered with syntax highlighting.

```
Theme Repository::defaultTheme(Repository::DefaultTheme t)
{
    if (t == DarkTheme)
        return theme(QLatin1String("Breeze Dark"));
    return theme(QLatin1String("Default"));
}
```

The same C++ function, without highlighting.

Of the two examples, which is easiest to read?

KatePart comes with a flexible, configurable and capable system for doing syntax highlighting, and the standard distribution provides definitions for a wide range of programming, scripting and markup languages and other text file formats. In addition you can provide your own definitions in simple XML files.

KatePart will automatically detect the right syntax rules when you open a file, based on the MIME Type of the file, determined by its extension, or, if it has none, the contents. Should you experience a bad choice, you can manually set the syntax to use from the **Tools** → **Highlighting** menu.

The styles and colors used by each syntax highlight definition can be configured using the [Highlighting Text Styles](#) tab of the [Config Dialog](#), while the MIME Types and file extensions it should be used for are handled by the [Modes & Filetypes](#) tab.

NOTE

Syntax highlighting is there to enhance the readability of correct text, but you cannot trust it to validate your text. Marking text for syntax is difficult depending on the format you are using, and in some cases the authors of the syntax rules will be proud if 98% of text gets correctly rendered, though most often you need a rare style to see the incorrect 2%.

6.2.2 The KatePart Syntax Highlight System

This section will discuss the KatePart syntax highlighting mechanism in more detail. It is for you if you want to know about it, or if you want to change or create syntax definitions.

6.2.2.1 How it Works

Whenever you open a file, one of the first things the KatePart editor does is detect which syntax definition to use for the file. While reading the text of the file, and while you type away in it, the syntax highlighting system will analyze the text using the rules defined by the syntax definition and mark in it where different contexts and styles begin and end.

When you type in the document, the new text is analyzed and marked on the fly, so that if you delete a character that is marked as the beginning or end of a context, the style of surrounding text changes accordingly.

The syntax definitions used by the KatePart Syntax Highlighting System are XML files, containing

- Rules for detecting the role of text, organized into context blocks
- Keyword lists
- Style Item definitions

When analyzing the text, the detection rules are evaluated in the order in which they are defined, and if the beginning of the current string matches a rule, the related context is used. The start point in the text is moved to the final point at which that rule matched and a new loop of the rules begins, starting in the context set by the matched rule.

6.2.2.2 Rules

The detection rules are the heart of the highlighting detection system. A rule is a string, character or [regular expression](#) against which to match the text being analyzed. It contains information about which style to use for the matching part of the text. It may switch the working context of the system either to an explicitly mentioned context or to the previous context used by the text.

Rules are organized in context groups. A context group is used for main text concepts within the format, for example quoted text strings or comment blocks in program source code. This ensures that the highlighting system does not need to loop through all rules when it is not necessary, and that some character sequences in the text can be treated differently depending on the current context.

Contexts may be generated dynamically to allow the usage of instance specific data in rules.

6.2.2.3 Context Styles and Keywords

In some programming languages, integer numbers are treated differently from floating point ones by the compiler (the program that converts the source code to a binary executable), and there may be characters having a special meaning within a quoted string. In such cases, it makes sense to render them differently from the surroundings so that they are easy to identify while reading the text. So even if they do not represent special contexts, they may be seen as such by the syntax highlighting system, so that they can be marked for different rendering.

A syntax definition may contain as many styles as required to cover the concepts of the format it is used for.

In many formats, there are lists of words that represent a specific concept. For example, in programming languages, control statements are one concept, data type names another, and built in functions of the language a third. The KatePart Syntax Highlighting System can use such lists to detect and mark words in the text to emphasize concepts of the text formats.

6.2.2.4 Default Styles

If you open a C++ source file, a Java™ source file and an HTML document in KatePart, you will see that even though the formats are different, and thus different words are chosen for special treatment, the colors used are the same. This is because KatePart has a predefined list of Default Styles which are employed by the individual syntax definitions.

This makes it easy to recognize similar concepts in different text formats. For example, comments are present in almost any programming, scripting or markup language, and when they are rendered using the same style in all languages, you do not have to stop and think to identify them within the text.

TIP

All styles in a syntax definition use one of the default styles. A few syntax definitions use more styles than there are defaults, so if you use a format often, it may be worth launching the configuration dialog to see if some concepts use the same style. For example, there is only one default style for strings, but as the Perl programming language operates with two types of strings, you can enhance the highlighting by configuring those to be slightly different. All [available default styles](#) will be explained later.

6.2.3 The Highlight Definition XML Format

6.2.3.1 Overview

KatePart uses the Syntax-Highlighting framework from KDE Frameworks. The default highlighting XML files shipped with KatePart are compiled into the Syntax-Highlighting library by default.

This section is an overview of the Highlight Definition XML format. Based on a small example it will describe the main components and their meaning and usage. The next section will go into detail with the highlight detection rules.

The formal definition, also known as the XSD you find in [Syntax Highlighting repository](#) in the file `language.xsd`

Custom `.xml` highlight definition files are located in `org.kde.syntax-highlighting/syntax/` in your user folder found with `qtpaths --paths GenericDataLocation` which usually are `$HOME /.local/share/` and `/usr/share/`.

In Flatpak and Snap packages, the above directory will not work as the data location is different for each application. In a Flatpak application, the location of custom XML files is usually `$HOME /.var/app/ flatpak-package-name /data/org.kde.syntax-highlighting/syntax/` and in a Snap application that location is `$HOME /snap/ snap-package-name /current/.local/share/org.kde.syntax-highlighting/syntax/`.

On Windows[®] these files are located `%USERPROFILE%\AppData\Local\org.kde.syntax-highlighting\syntax.` `%USERPROFILE%` usually expands to `C:\Users\user.`

In summary, for most configurations the directory of custom XML files is as follows:

For local user	<code>\$HOME /.local/share/org.kde.syntax-highlighting/syntax/</code>
For all users	<code>/usr/share/org.kde.syntax-highlighting/syntax/</code>
For Flatpak packages	<code>\$HOME /.var/app/ flatpak-package-name /data/org.kde.syntax-highlighting/syntax/</code>
For Snap packages	<code>\$HOME /snap/ snap-package-name /current/.local/share/org.kde.syntax-highlighting/syntax/</code>
On Windows [®]	<code>%USERPROFILE%\AppData\Local\org.kde.syntax-highlighting\syntax</code>

If multiple files exist for the same language, the file with the highest **version** attribute in the **language** element will be loaded.

MAIN SECTIONS OF KATEPART HIGHLIGHT DEFINITION FILES

A highlighting file contains a header that sets the XML version:

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

The root of the definition file is the element `language`. Available attributes are:

Required attributes:

name sets the name of the language. It appears in the menus and dialogs afterwards.

section specifies the category.

extensions defines file extensions, such as `/*.cpp;*.h`

version specifies the current revision of the definition file in terms of an integer number. Whenever you change a highlighting definition file, make sure to increase this number.

kateversion specifies the latest supported KatePart version.

Optional attributes:

mimetype associates files MIME type.

casesensitive defines, whether the keywords are case sensitive or not.

priority is necessary if another highlight definition file uses the same extensions. The higher priority will win.

author contains the name of the author and his email-address.

license contains the license, usually the MIT license for new syntax-highlighting files.

style contains the provided language and is used by the indenters for the attribute `require-syntax-style`.

indenter defines which indenter will be used by default. Available indenters are: *ada*, *normal*, *cstyle*, *cmake*, *haskell*, *latex*, *lilypond*, *lisp*, *lua*, *pascal*, *python*, *replicode*, *ruby* and *xml*.

hidden defines whether the name should appear in KatePart's menus.

So the next line may look like this:

```
<language name="C++" version="1" kateversion="2.4" section="Sources" ←
  extensions="*.cpp;*.h" />
```

Next comes the **highlighting** element, which contains the optional element **list** and the required elements **contexts** and **itemDatas**.

list elements contain a list of keywords. In this case the keywords are *class* and *const*. You can add as many lists as you need.

Since KDE Frameworks 5.53, a list can include keywords from another list or language/file, using the **include** element. **##** is used to separate the list name and the language definition name, in the same way as in the **IncludeRules** rule. This is useful to avoid duplicating keyword lists, if you need to include the keywords of another language/file. For example, the *othername* list contains the *str* keyword and all the keywords of the *types* list, which belongs to the ISO C++ language.

The **contexts** element contains all contexts. The first context is by default the start of the highlighting. There are two rules in the context *Normal Text*, which match the list of keywords with the name *somename* and a rule that detects a quote and switches the context to *string*. To learn more about rules read the next chapter.

The third part is the **itemDatas** element. It contains all color and font styles needed by the contexts and rules. In this example, the **itemData** *Normal Text*, *String* and *Keyword* are used.

```
<highlighting>
  <list name="somename">
    <item>class</item>
    <item>const</item>
  </list>
  <list name="othername">
    <item>str</item>
    <include>types##ISO C++</include>
  </list>
  <contexts>
    <context attribute="Normal Text" lineEndContext="#pop" name=" ←
      Normal Text" >
      <keyword attribute="Keyword" context="#stay" String="somename" ←
        />
```

The KatePart Handbook

```
<keyword attribute="Keyword" context="#stay" String="othername" ↔
  />
<DetectChar attribute="String" context="string" char="&quot;" ↔
  />
</context>
<context attribute="String" lineEndContext="#stay" name="string" ↔
  >
  <DetectChar attribute="String" context="#pop" char="&quot;" />
</context>
</contexts>
<itemDatas>
  <itemData name="Normal Text" defStyleNum="dsNormal" />
  <itemData name="Keyword" defStyleNum="dsKeyword" />
  <itemData name="String" defStyleNum="dsString" />
</itemDatas>
</highlighting>
```

The last part of a highlight definition is the optional **general** section. It may contain information about keywords, code folding, comments, indentation, empty lines and spell checking.

The **comment** section defines with what string a single line comment is introduced. You also can define a multiline comment using *multiLine* with the additional attribute *end*. This is used if the user presses the corresponding shortcut for *comment/uncomment*.

The **keywords** section defines whether keyword lists are case sensitive or not. Other attributes will be explained later.

The other sections, **folding**, **emptyLines** and **spellchecking**, are usually not necessary and are explained later.

```
<general>
  <comments>
    <comment name="singleLine" start="#" />
    <comment name="multiLine" start="###" end="###" region=" ↔
      CommentFolding" />
  </comments>
  <keywords casesensitive="1" />
  <folding indentationsensitive="0" />
  <emptyLines>
    <emptyLine regexpr="\s+" />
    <emptyLine regexpr="\s*#.*/>
  </emptyLines>
  <spellchecking>
    <encoding char="á" string="\`a" />
    <encoding char="à" string="\`a" />
  </spellchecking>
</general>
</language>
```

6.2.3.2 The Sections in Detail

This part will describe all available attributes for contexts, itemDatas, keywords, comments, code folding and indentation.

The element **context** belongs in the group **contexts**. A context itself defines context specific rules such as what should happen if the highlight system reaches the end of a line. Available attributes are:

name states the context name. Rules will use this name to specify the context to switch to if the rule matches.

lineEndContext defines the context the highlight system switches to if it reaches the end of a line. This may either be a name of another context, **#stay** to not switch the context (e.g. do nothing) or **#pop** which will cause it to leave this context. It is possible to use for example **#pop#pop#pop** to pop three times, or even **#pop#pop!OtherContext** to pop two times and switch to the context named **OtherContext**. It is also possible to switch to a context that belongs to another language definition, in the same way as in the **IncludeRules** rules, e.g., **SomeContext##JavaScript**. Note that it is not possible to use this context switch in combination with **#pop**, for example, **#pop!SomeContext##JavaScript** is not valid. Context switches are also described in Section 6.2.4.

lineEmptyContext defines the context if an empty line is encountered. The nomenclature of context switches is the same as previously described in *lineEndContext*. Default: **#stay**.

fallthroughContext specifies the next context to switch to if no rule matches. The nomenclature of context switches is the same as previously described in *lineEndContext*. Default: **#stay**.

fallthrough defines if the highlight system switches to the context specified in **fallthroughContext** if no rule matches. Note that since KDE Frameworks 5.62 this attribute is deprecated in favor of **fallthroughContext**, since if the **fallthroughContext** attribute is present it is implicitly understood that the value of **fallthrough** is *true*. Default: *false*.

noIndentationBasedFolding disables indentation-based folding in the context. If indentation-based folding is not activated, this attribute is useless. This is defined in the element *folding* of the group *general*. Default: *false*.

The element **itemData** is in the group **itemDatas**. It defines the font style and colors. So it is possible to define your own styles and colors. However, we recommend you stick to the default styles if possible so that the user will always see the same colors used in different languages. Though, sometimes there is no other way and it is necessary to change color and font attributes. The attributes **name** and **defStyleNum** are required, the others are optional.

Available attributes are:

name sets the name of the **itemData**. Contexts and rules will use this name in their attribute *attribute* to reference an **itemData**.

defStyleNum defines which default style to use. Available default styles are explained in detail later.

color defines a color. Valid formats are *'#rrggbb'* or *'#rgb'*.

selColor defines the selection color.

italic if *true*, the text will be italic.

bold if *true*, the text will be bold.

underline if *true*, the text will be underlined.

strikeout if *true*, the text will be struck out.

spellChecking if *true*, the text will be spellchecked.

The element **keywords** in the group **general** defines keyword properties. Available attributes are:

casesensitive may be *true* or *false*. If *true*, all keywords are matched case sensitively.

weakDelimiter is a list of characters that do not act as word delimiters. For example, the dot *'.'* is a word delimiter. Assume a keyword in a **list** contains a dot, it will only match if you specify the dot as a weak delimiter.

additionalDelimiter defines additional delimiters.

wordWrapDelimiter defines characters after which a line wrap may occur.

Default delimiters and word wrap delimiters are the characters *. () : ! + , - < = > % & * / ; ? [] ^ { | } ~ *, space (*' '*) and tabulator (*'\t'*).

The element **comment** in the group **comments** defines comment properties which are used for Tools → Comment, Tools → Uncomment and Tools → Toggle Comment. Available attributes are:

name is either *singleLine* or *multiLine*. If you choose *multiLine* the attributes *end* and *region* are required. If you choose *singleLine* you can add the optional attribute *position*.

start defines the string used to start a comment. In C++ this would be `"/**"` in multiline comments. This attribute is required for types *multiLine* and *singleLine*.

end defines the string used to close a comment. In C++ this would be `**/"`. This attribute is only available and is required for comments of type *multiLine*.

region should be the name of the foldable multiline comment. Assume you have `beginRegion="Comment" ... endRegion="Comment"` in your rules, you should use `region="Comment"`. This way uncomment works even if you do not select all the text of the multiline comment. The cursor only must be in the multiline comment. This attribute is only available for type *multiLine*.

position defines where the single line comment is inserted. By default, the single line comment is placed at the beginning of the line at column 0, but if you use `position="afterwhitespace"` the comment is inserted after leading whitespaces right, before the first non-whitespace character. This is useful for putting comments correctly in languages where indentation is important, such as Python or YAML. This attribute is optional and the only possible value is *afterwhitespace*. This is only available for type *singleLine*.

The element **fold** in the group **general** defines code folding properties. Available attributes are:

indentationsensitive if *true*, the code folding markers will be added indentation based, as in the scripting language Python. Usually you do not need to set it, as it defaults to *false*.

The element **emptyLine** in the group **emptyLines** defines which lines should be treated as empty lines. This allows modifying the behavior of the *lineEmptyContext* attribute in the elements **context**. Available attributes are:

regexpr defines a regular expression that will be treated as an empty line. By default, empty lines do not contain any characters, therefore, this adds additional empty lines, for example, if you want lines with spaces to also be considered empty lines. However, in most syntax definitions you do not need to set this attribute.

The element **encoding** in the group **spellchecking** defines a character encoding for spell checking. Available attributes:

char is a encoded character.

string is a sequence of characters that will be encoded as the character *char* in the spell checking. For example, in the language LaTeX, the string `\~{A}` represents the character **Ä**.

6.2.3.3 Available Default Styles

Default Styles were [already explained](#), as a short summary: Default styles are predefined font and color styles.

General default styles:

dsNormal, when no special highlighting is required.

dsKeyword, built-in language keywords.

dsFunction, function calls and definitions.

dsVariable, if applicable: variable names (e.g. `$someVar` in PHP/Perl).

dsControlFlow, control flow keywords like `if`, `else`, `switch`, `break`, `return`, `yield`, ...

dsOperator, operators like + - * / :: < >

dsBuiltIn, built-in functions, classes, and objects.

dsExtension, common extensions, such as Qt™ classes and functions/macros in C++ and Python.

dsPreprocessor, preprocessor statements or macro definitions.

dsAttribute, annotations such as @override and __declspec(...).

String-related default styles:

dsChar, single characters, such as 'x'.

dsSpecialChar, chars with special meaning in strings such as escapes, substitutions, or regex operators.

dsString, strings like "hello world".

dsVerbatimString, verbatim or raw strings like 'raw \backslash' in Perl, CoffeeScript, and shells, as well as r'\raw' in Python.

dsSpecialString, SQL, regexes, HERE docs, L^AT_EX math mode, ...

dsImport, import, include, require of modules.

Number-related default styles:

dsDataType, built-in data types like int, void, u64.

dsDecVal, decimal values.

dsBaseN, values with a base other than 10.

dsFloat, floating point values.

dsConstant, built-in and user defined constants like PI.

Comment and documentation-related default styles:

dsComment, comments.

dsDocumentation, /* Documentation comments */ or """docstrings""".

dsAnnotation, documentation commands like @param, @brief.

dsCommentVar, the variable names used in above commands, like "foobar" in @param foobar.

dsRegionMarker, region markers like //BEGIN, //END in comments.

Other default styles:

dsInformation, notes and tips like @note in doxygen.

dsWarning, warnings like @warning in doxygen.

dsAlert, special words like TODO, FIXME, XXXX.

dsError, error highlighting and wrong syntax.

dsOthers, when nothing else fits.

6.2.4 Highlight Detection Rules

This section describes the syntax detection rules.

Each rule can match zero or more characters at the beginning of the string they are tested against. If the rule matches, the matching characters are assigned the style or *attribute* defined by the rule, and a rule may ask that the current context is switched.

A rule looks like this:

```
<RuleName attribute="(identifier)" context="(identifier)" [rule specific ↔  
attributes] />
```

The *attribute* identifies the style to use for matched characters by name, and the *context* identifies the context to use from here.

The *context* can be identified by:

- An *identifier*, which is the name of the other context.
- An *order* telling the engine to stay in the current context (**#stay**), or to pop back to a previous context used in the string (**#pop**).

To go back more steps, the **#pop** keyword can be repeated: **#pop#pop#pop**

- An *order* followed by an exclamation mark (!) and an *identifier*, which will make the engine first follow the order and then switch to the other context, e.g. **#pop#pop!OtherContext**.
- An *identifier*, which is a context name, followed by two hashes (##) and another *identifier*, which is the name of a language definition. This naming is similar to that used in **IncludeRules** rules and allows you to switch to a context belonging to another syntax highlighting definition, e.g. **SomeContext##JavaScript**. Note that it is not possible to use this context switch in combination with **#pop**, for example, **#pop!SomeContext##JavaScript** is not valid.

Rule specific attributes varies and are described in the following sections.

COMMON ATTRIBUTES

- *attribute*: An attribute maps to a defined *itemData*.
- *context*: Specify the context to which the highlighting system switches if the rule matches.
- *beginRegion*: Start a code folding block. Default: unset.
- *endRegion*: Close a code folding block. Default: unset.
- *lookAhead*: If *true*, the highlighting system will not process the matches length. Default: *false*.
- *firstNonSpace*: Match only, if the string is the first non-whitespace in the line. Default: *false*.
- *column*: Match only, if the column matches. Default: unset.

DYNAMIC RULES

- *dynamic*: may be (*true|false*).

How does it work:

In the [regular expressions](#) of the **RegExp** rules, all text within simple curved brackets (**PATTERN**) is captured and remembered. These captures can be used in the context to which it is switched, in the rules with the attribute **dynamic true**, by $\%N$ (in *String*) or N (in *char*).

It is important to mention that a text captured in a **RegExp** rule is only stored for the switched context, specified in its **context** attribute.

TIP

- If the captures will not be used, both by dynamic rules and in the same regular expression, **non-capturing groups** should be used: **(?:PATTERN)**
The *lookahead* or *lookbehind* groups such as **(?=PATTERN)**, **(?!PATTERN)** or **(?<=PATTERN)** are not captured. See [Regular Expressions](#) for more information.
- The capture groups can be used within the same regular expression, using $\backslash N$ instead of $\%N$ respectively. For more information, see [Capturing matching text \(back references\)](#) in [Regular Expressions](#).

The KatePart Handbook

Example 1:

In this simple example, the text matched by the regular expression `=*` is captured and inserted into `%1` in the dynamic rule. This allows the comment to end with the same amount of `=` as at the beginning. This matches text like: `[[comment]]`, `[=[comment]=]` or `[====[comment]=====]`.

In addition, the captures are available only in the switched context *Multi-line Comment*.

```
<context name="Normal" attribute="Normal Text" lineEndContext="#stay">
  <Regex context="Multi-line Comment" attribute="Comment" String="\[(=*) \[
    \[" beginRegion="RegionComment"/>
</context>
<context name="Multi-line Comment" attribute="Comment" lineEndContext="#
stay">
  <StringDetect context="#pop" attribute="Comment" String="]%1]" dynamic="
true" endRegion="RegionComment"/>
</context>
```

Example 2:

In the dynamic rule, `%1` corresponds to the capture that matches `#+`, and `%2` to `"+;`. This matches text as: `#label~~~~"inside the context"~~~~#`.

These captures will not be available in other contexts, such as *OtherContext*, *FindEscapes* or *SomeContext*.

```
<context name="SomeContext" attribute="Normal Text" lineEndContext="#stay">
  <Regex context="#pop!NamedString" attribute="String" String="(#+) (?:\w
-|[\^[:ascii:]])(&quot;+;)" />
</context>
<context name="NamedString" attribute="String" lineEndContext="#stay">
  <Regex context="#pop!OtherContext" attribute="String" String="%2(?:%1
?)" dynamic="true" />
  <DetectChar context="FindEscapes" attribute="Escape" char="\\" />
</context>
```

Example 3:

This matches text like: `Class::function<T>(...)`.

```
<context name="Normal" attribute="Normal Text" lineEndContext="#stay">
  <Regex context="FunctionName" lookAhead="true"
    String="\b([a-zA-Z_][\w-]*) (?:) ([a-zA-Z_][\w-]*) (?&lt;[\w\-\
s]*&gt;)? \(\)" />
</context>
<context name="FunctionName" attribute="Normal Text" lineEndContext="#pop">
  <StringDetect context="#stay" attribute="Class" String="%1" dynamic="true
"/>
  <StringDetect context="#stay" attribute="Operator" String="%2" dynamic="
true"/>
  <StringDetect context="#stay" attribute="Function" String="%3" dynamic="
true"/>
  <DetectChar context="#pop" attribute="Normal Text" char="4" dynamic="true
"/>
</context>
```

LOCAL DELIMITATORS

- *weakDelimiter*: list of characters that do not act as word delimiters.
- *additionalDelimiter*: defines additional delimiters.

6.2.4.1 The Rules in Detail

DetectChar

Detect a single specific character. Commonly used for example to find the ends of quoted strings.

```
<DetectChar char="(character)" (common attributes) (dynamic) />
```

The **char** attribute defines the character to match.

Detect2Chars

Detect two specific characters in a defined order.

```
<Detect2Chars char="(character)" char1="(character)" (common attributes <->
) />
```

The **char** attribute defines the first character to match, **char1** the second.

AnyChar

Detect one character of a set of specified characters.

```
<AnyChar String="(string)" (common attributes) />
```

The **String** attribute defines the set of characters.

StringDetect

Detect an exact string.

```
<StringDetect String="(string)" [insensitive="true|false"] (common <->
attributes) (dynamic) />
```

The **String** attribute defines the string to match. The **insensitive** attribute defaults to *false* and is passed to the string comparison function. If the value is *true* insensitive comparing is used.

WordDetect

Detect an exact string but additionally require word boundaries such as a dot *'.'* or a whitespace on the beginning and the end of the word. Think of **\b<string>\b** in terms of a regular expression, but it is faster than the rule **RegExpr**.

```
<WordDetect String="(string)" [insensitive="true|false"] (common <->
attributes) (local delimiters) />
```

The **String** attribute defines the string to match. The **insensitive** attribute defaults to *false* and is passed to the string comparison function. If the value is *true* insensitive comparing is used.

Since: Kate 3.5 (KDE 4.5)

RegExpr

Matches against a regular expression.

```
<RegExpr String="(string)" [insensitive="true|false"] [minimal="true|<->
false"] (common attributes) (dynamic) />
```

The **String** attribute defines the regular expression.

insensitive defaults to *false* and is passed to the regular expression engine.

minimal defaults to *false* and is passed to the regular expression engine.

Because the rules are always matched against the beginning of the current string, a regular expression starting with a caret (^) indicates that the rule should only be matched against the start of a line.

See [Regular Expressions](#) for more information on those.

keyword

Detect a keyword from a specified list.

```
<keyword String="(list name)" (common attributes) (local delimiters) <-  
  />
```

The **String** attribute identifies the keyword list by name. A list with that name must exist. The highlighting system processes keyword rules in a very optimized way. This makes it an absolute necessity that any keywords to be matched need to be surrounded by defined delimiters, either implied (the default delimiters), or explicitly specified within the *additionalDelimiter* property of the *keywords* tag.

If a keyword to be matched shall contain a delimiter character, this respective character must be added to the *weakDelimiter* property of the *keywords* tag. This character will then lose its delimiter property in all *keyword* rules. It is also possible to use the *weakDelimiter* attribute of *keyword* so that this modification only applies to this rule.

Int

Detect an integer number (as the regular expression: `\b[0-9]+`).

```
<Int (common attributes) (local delimiters) />
```

This rule has no specific attributes.

Float

Detect a floating point number (as the regular expression: `(\b[0-9]+\.[0-9]*|\.[0-9]+) ([eE] [-+]?[0-9]+)?`).

```
<Float (common attributes) (local delimiters) />
```

This rule has no specific attributes.

HlCOct

Detect an octal point number representation (as the regular expression: `\b0[0-7]+`).

```
<HlCOct (common attributes) (local delimiters) />
```

This rule has no specific attributes.

HlCHex

Detect a hexadecimal number representation (as a regular expression: `\b0[xX][0-9a-fA-F]+`).

```
<HlCHex (common attributes) (local delimiters) />
```

This rule has no specific attributes.

HlCStringChar

Detect an escaped character.

```
<HLCStringChar (common attributes) />
```

This rule has no specific attributes.

It matches literal representations of characters commonly used in program code, for example `\n` (newline) or `\t` (TAB).

The following characters will match if they follow a backslash (`\`): **abefnrtv`'?**. Additionally, escaped hexadecimal numbers such as for example `\xff` and escaped octal numbers, for example `\033` will match.

HLCChar

Detect an C character.

```
<HLCChar (common attributes) />
```

This rule has no specific attributes.

It matches C characters enclosed in a tick (Example: `'c'`). The ticks may be a simple character or an escaped character. See `HLCStringChar` for matched escaped character sequences.

RangeDetect

Detect a string with defined start and end characters.

```
<RangeDetect char="(character)" char1="(character)" (common attributes <-> ) />
```

char defines the character starting the range, **char1** the character ending the range.

Useful to detect for example small quoted strings and the like, but note that since the highlighting engine works on one line at a time, this will not find strings spanning over a line break.

LineContinue

Matches a specified char at the end of a line.

```
<LineContinue (common attributes) [char="\"] />
```

char optional character to match, default is backslash (`'\'`). New since KDE 4.13.

This rule is useful for switching context at end of line. This is needed for example in C/C++ to continue macros or strings.

IncludeRules

Include rules from another context or language/file.

```
<IncludeRules context="contextlink" [includeAttrib="true|false"] />
```

The **context** attribute defines which context to include.

If it is a simple string it includes all defined rules into the current context, example:

```
<IncludeRules context="anotherContext" />
```

If the string contains a **##** the highlight system will look for a context from another language definition with the given name, for example

```
<IncludeRules context="String##C++" />
```

would include the context *String* from the C++ highlighting definition.

If **includeAttrib** attribute is *true*, change the destination attribute to the one of the source. This is required to make, for example, commenting work, if text matched by the included context is a different highlight from the host context.

DetectSpaces

Detect whitespaces.

```
<DetectSpaces (common attributes) />
```

This rule has no specific attributes.

Use this rule if you know that there can be several whitespaces ahead, for example in the beginning of indented lines. This rule will skip all whitespace at once, instead of testing multiple rules and skipping one at a time due to no match.

DetectIdentifier

Detect identifier strings (as the regular expression: `[a-zA-Z_][a-zA-Z0-9_]*`).

```
<DetectIdentifier (common attributes) />
```

This rule has no specific attributes.

Use this rule to skip a string of word characters at once, rather than testing with multiple rules and skipping one at a time due to no match.

6.2.4.2 Tips & Tricks

- If you only match two characters use **Detect2Chars** instead of **StringDetect**. The same applies to **DetectChar**.
- Regular expressions are easy to use but often there is another much faster way to achieve the same result. Consider you only want to match the character '#' if it is the first character in the line. A regular expression based solution would look like this:

```
<RegExpr attribute="Macro" context="macro" String="^\s*#" />
```

You can achieve the same much faster in using:

```
<DetectChar attribute="Macro" context="macro" char="#" firstNonSpace="↔ true" />
```

If you want to match the regular expression '`^#`' you can still use **DetectChar** with the attribute `column="0"`. The attribute `column` counts characters, so a tabulator is only one character.

- In **RegExpr** rules, use the attribute `column="0"` if the pattern `^PATTERN` will be used to match text at the beginning of a line. This improves performance, as it will avoid looking for matches in the rest of the columns.
- In regular expressions, use non-capturing groups (`?:PATTERN`) instead of capturing groups (`PATTERN`), if the captures will not be used in the same regular expression or in dynamic rules. This avoids storing captures unnecessarily.
- You can switch contexts without processing characters. Assume that you want to switch context when you meet the string `*/`, but need to process that string in the next context. The below rule will match, and the **lookAhead** attribute will cause the highlighter to keep the matched string for the next context.

```
<Detect2Chars attribute="Comment" context="#pop" char="*" char1="/" ↔ lookAhead="true" />
```

- Use **DetectSpaces** if you know that many whitespaces occur.
- Use **DetectIdentifier** instead of the regular expression '`[a-zA-Z_]\w*`'.

- Use default styles whenever you can. This way the user will find a familiar environment.
- Look into other XML files to see how other people implement tricky rules.
- You can validate every XML file by using the command **validatehl.sh language.xsd mySyntax.xml**. The files `validatehl.sh` and `language.xsd` are available in [Syntax Highlighting repository](#).
- If you repeat complex regular expression very often you can use *ENTITIES*. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE language SYSTEM "language.dtd"
[
    <!ENTITY myref      "[A-Za-z_:] [\w.:\_]*">
]>
```

Now you can use *&myref*; instead of the regular expression.

6.3 Working with Color Themes

6.3.1 Overview

Color themes define the colors of the [text editing area](#) and the [syntax highlighting](#). A color theme encompasses the following:

- The text style, used for syntax highlighting through the *default styles attributes*. For example, the text color and the selected text color.
- The background of the text editing area, including the text selection and the current line.
- The icon border of the text area: their background, the separator line, the line numbers, the line word wrap markers, the modified line marks and the code folding.
- Text decorators such as the search markers, the indentation and tab/space line marks, the bracket matching and the spell checking.
- Bookmarks and snippets.

To avoid confusion, the following is out of scope:

- The font type and the font size.
- The colors of the text editing application, such as the scroll bar map, the menus, the tab bar, the window color, etc. In KDE applications, like Kate or KDevelop, these colors are defined by the **KDE Plasma global color scheme**, which are set in the [Colors module in System Settings](#) or from the application itself in the menu **Settings** → **Color Scheme**.

```

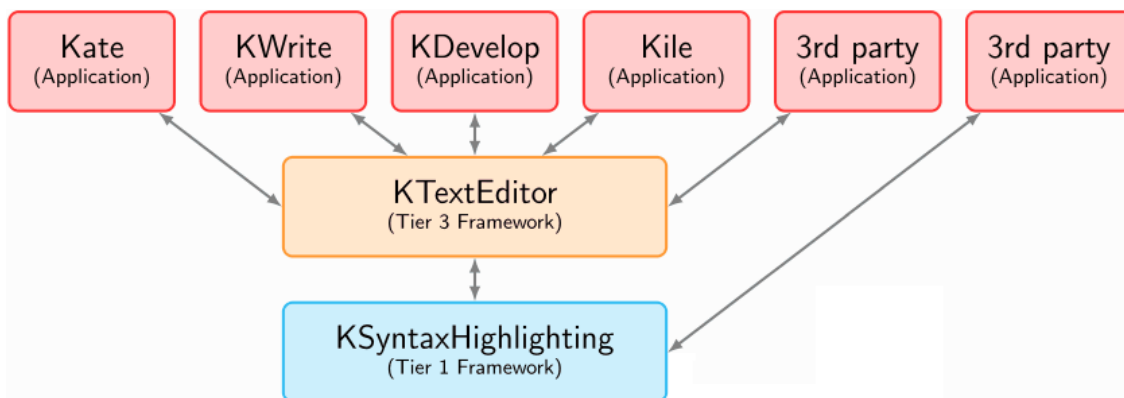
1  /**
2  * SPDX-FileCopyrightText: 2020 Christoph Cullmann <cullmann@kde.org>
3  * SPDX-License-Identifier: MIT
4  */
5
6  // BEGIN
7  #include <string>
8  #include <QString>
9  // END
10
11 /**
12 * TODO: improve documentation
13 * @param magicArgument some magic argument
14 * @return magic return value
15 */
16 int main(uint64_t magicArgument)
17 {
18     if (magicArgument > 1) {
19         const std::string string = "source file: \"__FILE__\"";
20         const QString qString(QStringLiteral("test"));
21         return qrand();
22     }
23
24     /* BUG: bogus integer constant inside next line */
25     const double g = 1.1e12 * 0b01'01'01'01 - 43a + 0x11234 * 0234ULL - 'c' * 42;
26     return g > 1.3f;
27 }

```

'Breeze Light' and 'Breeze Dark' color themes with the 'C++' syntax highlighting.

6.3.2 The KSyntaxHighlighting Color Themes

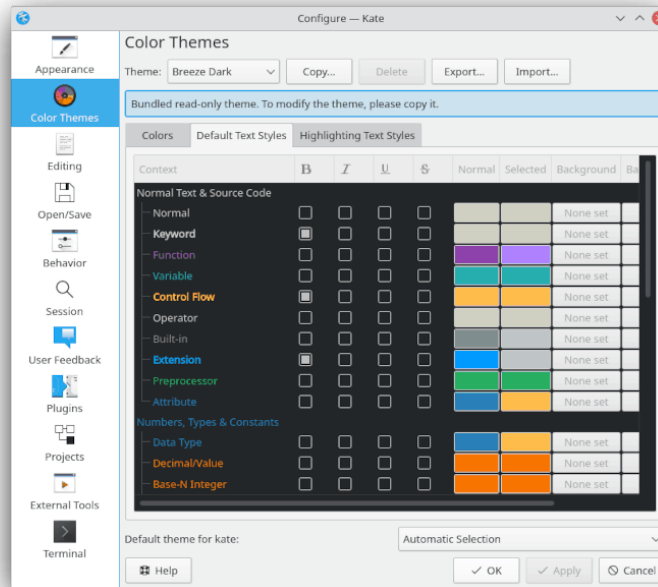
The `KSyntaxHighlighting` framework, which is the `syntax highlighting` engine, is the library that **provides and manages the color themes**. This is part of KDE Frameworks and is used in KDE text editors such as `Kate`, `KWrite`, `Kile` and `KDevelop`. This dependency looks like the following:



Dependence of KDE Frameworks libraries on text editors.

`KSyntaxHighlighting` includes a variety of built-in themes which are displayed on the [Color Themes](#) page of the Kate editor website.

The `KTextEditor` framework, which is the text editing engine, provides a user interface for creating and editing color themes, including a tool for importing and exporting themes. This is the simplest way to create and edit them, you can access it from the `'Configure'` dialog of the text editor. More details in Section 6.3.5.



The GUI to manage color themes in Kate's settings.

It is important to mention that, in the KDE text editors like Kate or KDevelop, the KSyntaxHighlighting color themes are used since KDE Frameworks 5.75, released on October 10, 2020. Previously, Kate's color schemes (KConfig based schema config) were used and are now deprecated. However, it is possible to convert the old Kate schemes to the KSyntaxHighlighting color themes. The [KSyntaxHighlighting repository](#) includes the `utils/kateschema_to_theme_converter.py` script and the `utils/schema-converter/` utility for that purpose.

6.3.3 The Color Themes JSON Format

6.3.3.1 Overview

Color themes are stored in JSON format files, with the `.theme` extension.

In the [KSyntaxHighlighting source code](#), the JSON files of built-in themes are located in the `data/themes/` directory. Note that in text editors, the built-in themes are compiled into the KSyntaxHighlighting library, therefore, the way to access them is through the source code or by [exporting them from the GUI to manage themes of KTextEditor](#).

It is also possible to easily add additional or custom themes, which are loaded from the file system. User-customized theme files are located in the `org.kde.syntax-highlighting/themes/` directory in your user folder, which you can find with the command `qtpaths --paths GenericDataLocation` and are commonly `$HOME/.local/share/` and `/usr/share/`.

In Flatpak and Snap packages, the above directory will not work as the data location is different for each application. In a Flatpak application, the location of custom theme files is usually `$HOME/.var/app/flatpak-package-name/data/org.kde.syntax-highlighting/themes/` and in a Snap application that location is `$HOME/.snap/snap-package-name/current/.local/share/org.kde.syntax-highlighting/themes/`.

On Windows[®] these files are located `%USERPROFILE%\AppData\Local\org.kde.syntax-highlighting\themes`. `%USERPROFILE%` usually expands to `C:\Users\user-name`.

In summary, for most configurations the custom themes directory is as follows:

For local user	\$HOME /.local/share/org.kde.syntax-highlighting/themes/
For all users	/usr/share/org.kde.syntax-highlighting/themes/
For Flatpak packages	\$HOME /.var/app/ flatpak-package-name /data/org.kde.syntax-highlighting/themes/
For Snap packages	\$HOME /snap/ snap-package-name /current/.local/share/org.kde.syntax-highlighting/themes/
On Windows®	%USERPROFILE%\AppData\Local\org.kde.syntax-highlighting\themes

If multiple theme files exist with the same name, the file with the highest **revision** will be loaded.

6.3.3.2 The JSON Structure

The structure of a JSON file is explained on [their website](#). Basically, a JSON format file consists of:

- Collections of key/value pairs, separated by commas and grouped in { } which we will call ‘objects’.
- Ordered lists of values, separated by commas and grouped in [] which we will call ‘array’.

The nomenclature ‘key’, ‘value’, ‘object’ and ‘array’ will be used in this article. If this is your first time working with JSON files, understanding them is as simple as looking at the examples below.

6.3.3.3 Main Sections of the JSON Color Theme Files

The root object of the color theme JSON file contains the following schema keys:

- **metadata**: It is mandatory. The value is an object with the theme’s metadata, such as name, revision and license.
This is detailed in Section [6.3.3.4](#).
- **editor-colors**: It is mandatory. The value is an object with the colors of the text editing area, such as the background, the icon border and the text decoration.
This is detailed in Section [6.3.4.1](#).
- **text-styles**: It is mandatory. The value is an object with the *default text style* attributes of the syntax highlighting. Each attribute defines its *text color*, its *selected text color*, or whether it *bold* or *italic*, for example. The text styles can be referenced from [the attributes of the syntax definition XML files](#).
This is detailed in Section [6.3.4.2](#).
- **custom-styles**: It is optional. Defines text styles for the attributes of specific syntax highlighting definitions. For example, in a highlighting definition such as **Python** or **Markdown** you can specify a different text style that overrides the default defined in **text-styles**.
This is detailed in Section [6.3.4.3](#).

The KatePart Handbook

The JSON language does not support comments. However, you can use the optional key `_comments` in the root object to write comments, for example, if you are adapting an existing theme you can put the URL of the original repository. The most practical way is to use an array of strings.

Below is an example file for the 'Breeze Light' theme. You can notice that, to avoid the example being too large, the `editor-colors` and `text-styles` objects do not contain all the required keys. You can see the full archive of [the Breeze Light theme in the KSyntaxHighlighting repository](#).

```
{
  "_comments": [
    "This is a comment.",
    "If this theme is an adaptation of another, put the link to the ←
      original repository."
  ],
  "metadata": {
    "name" : "Breeze Light",
    "revision" : 5,
    "copyright": [
      "SPDX-FileCopyrightText: 2016 Volker Krause <vkrause@kde.org>",
      "SPDX-FileCopyrightText: 2016 Dominik Haumann <dhaumann@kde.org ←
        >"
    ],
    "license": "SPDX-License-Identifier: MIT"
  },
  "editor-colors": {
    "BackgroundColor" : "#ffffff",
    "CodeFolding" : "#94caef",
    "BracketMatching" : "#ffff00",
    "CurrentLine" : "#f8f7f6",
    "IconBorder" : "#f0f0f0",
    "IndentationLine" : "#d2d2d2",
    "LineNumbers" : "#a0a0a0",
    "CurrentLineNumber" : "#1e1e1e",
    The other editor color keys...
  },
  "text-styles": {
    "Normal" : {
      "text-color" : "#1f1c1b",
      "selected-text-color" : "#ffffff",
      "bold" : false,
      "italic" : false,
      "underline" : false,
      "strike-through" : false
    },
    "Keyword" : {
      "text-color" : "#1f1c1b",
      "selected-text-color" : "#ffffff",
      "bold" : true
    },
    "Function" : {
      "text-color" : "#644a9b",
      "selected-text-color" : "#452886"
    },
    "Variable" : {
      "text-color" : "#0057ae",
      "selected-text-color" : "#00316e"
    },
    The other text style keys...
  }
}
```

```

    },
    "custom-styles": {
      "ISO C++": {
        "Data Type": {
          "bold": true,
          "selected-text-color": "#009183",
          "text-color": "#00b5cf"
        },
        "Keyword": {
          "text-color": "#6431b3"
        }
      },
      "YAML": {
        "Attribute": {
          "selected-text-color": "#00b5cf",
          "text-color": "#00b5cf"
        }
      }
    }
  }
}

```

6.3.3.4 Metadata

The JSON object of the **metadata** key contains relevant information on the theme. This object has the following keys:

- **name:** It is a *string* sets the name of the language. It appears in the menus and dialogs afterwards. It is mandatory.
- **revision:** It is an *integer* number that specifies the current revision of the theme file. Whenever you update a color theme file, make sure to increase this number. It is mandatory.
- **license:** It is a *string* that defines the license of the theme, using the identifier **SPDX-License-Identifier** from the standard [SPDX license communication format](#). It is optional.
You can see the full list of SPDX license identifiers [here](#).
- **copyright:** It is an *array* of *strings* that specifies the authors of the theme, using the identifier **SPDX-FileCopyrightText** from the standard [SPDX license communication format](#). It is optional.

```

"metadata": {
  "name" : "Breeze Light",
  "revision" : 5,
  "copyright": [
    "SPDX-FileCopyrightText: 2016 Volker Krause <vkrause@kde.org>",
    "SPDX-FileCopyrightText: 2016 Dominik Haumann <dhaumann@kde.org>"
  ],
  "license": "SPDX-License-Identifier: MIT"
}

```

6.3.4 Colors in Detail

This section details all the available color attributes and available color settings.

6.3.4.1 Editor Colors

Corresponds to the colors of the [text editing area](#).

In the [JSON theme file](#), the respective key `editor-colors` has as value an *object* where each key references an attribute color of the text editor. Here, **all available keys are mandatory**, their values are **strings** with hexadecimal color codes, like `'#00B5CF'`.

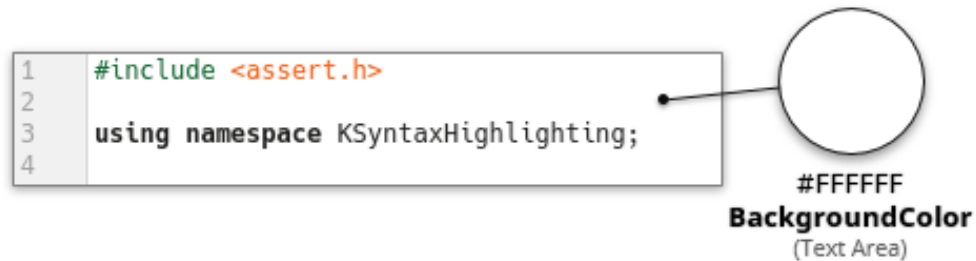
In the [GUI to manage themes of KTextEditor](#), these attributes can be modified in the **Colors** tab.

The available keys are the following; the keys used in the [JSON file](#) are listed in *bold*, the names used in the [GUI](#) are shown in parentheses.

Editor Background Colors

BackgroundColor (Text Area)

This is the default background for the editor area, it will be the dominant color on the editor area.



TextSelection (Selected Text)

This is the background for selected text.



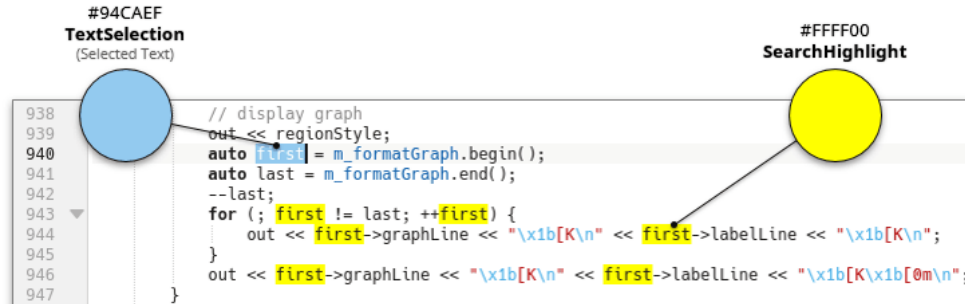
CurrentLine (Current Line)

Set the color for the current line. Setting this a bit different from the Normal text background helps to keep focus on the current line.



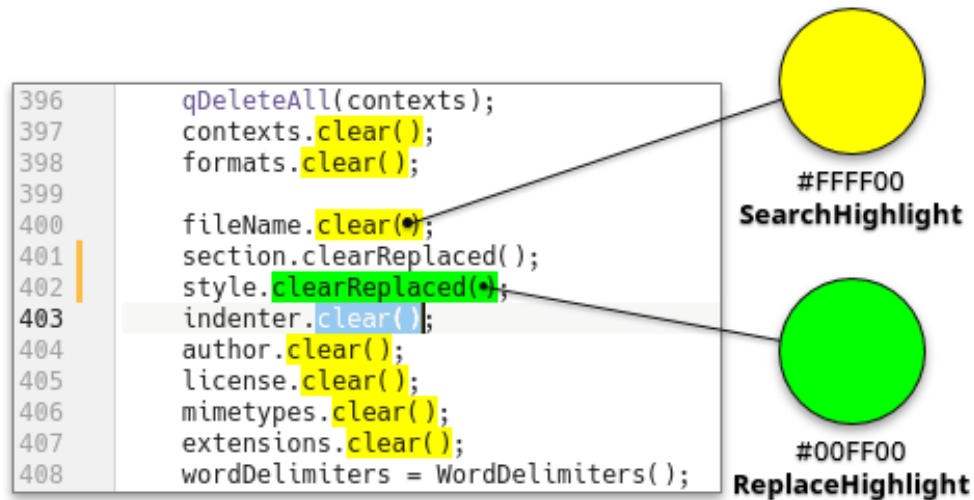
SearchHighlight (Search Highlight)

Set the color for the text that matches your last search.



ReplaceHighlight (Replace Highlight)

Set the color for the text that matches your last replace operation.



Icon Border

IconBorder (Background Area)

This color is used for the marks, line numbers and folding marker borders in the left side of the editor view when they are displayed.

LineNumbers (Line Numbers)

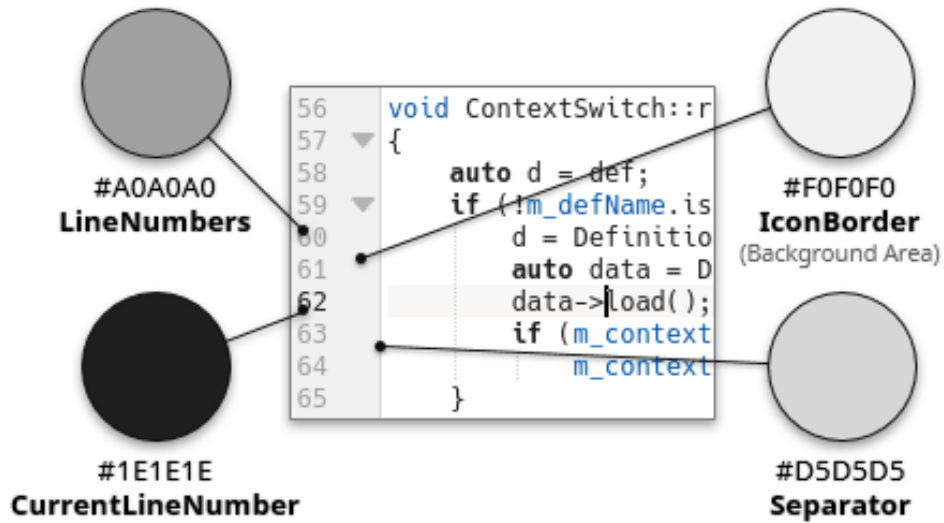
This color is used to draw the line numbers on the left side of the view when displayed.

CurrentLineNumber (Current Line Number)

This color is used to draw the line number of the current line, on the left side of the view when displayed. Setting this a bit different from 'LineNumbers' helps to keep focus on the current line.

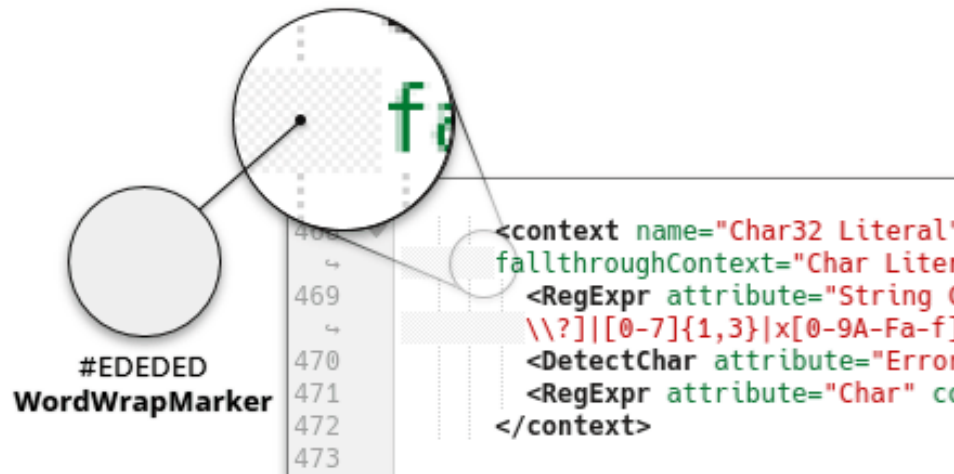
Separator (Separator)

This color is used to draw the vertical line that separates the icon border from the background of the text area.



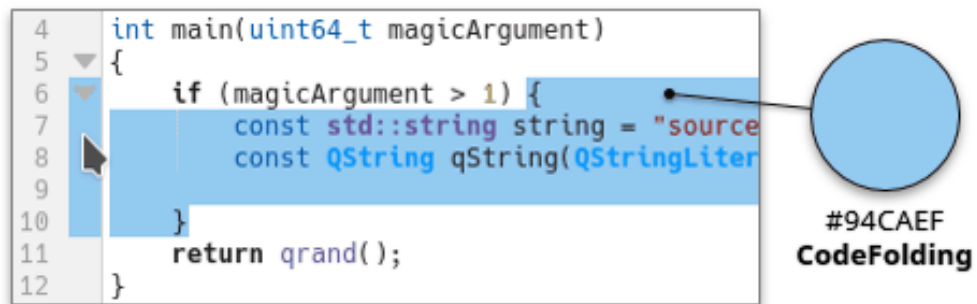
WordWrapMarker (Word Wrap Marker)

This color is used to draw a pattern to the left of dynamically wrapped lines when those are aligned vertically, as well as for the static word wrap marker.



CodeFolding (Code Folding)

This color is used to highlight the section of code that would be folded when you click on the code folding arrow to the left of a document. For more information, see [the code folding documentation](#).

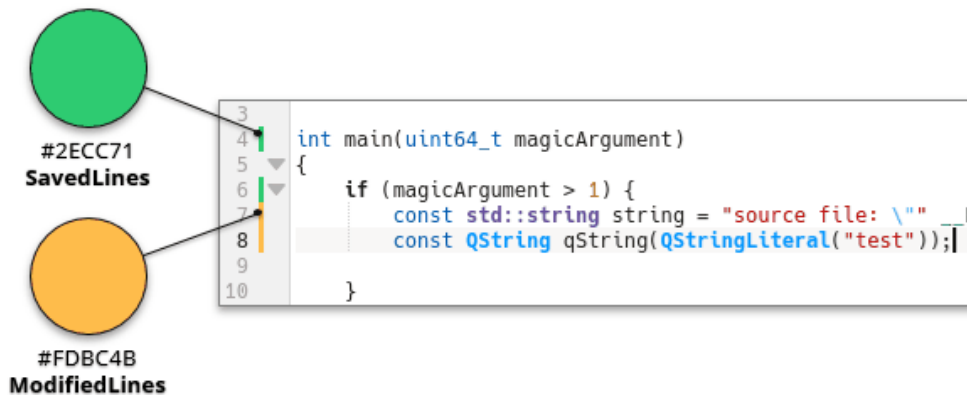


ModifiedLines (Modified Lines)

This color is used to highlight to the left of a document lines that have been modified but not yet saved. For more information, see Section 3.9.

SavedLines (Saved Lines)

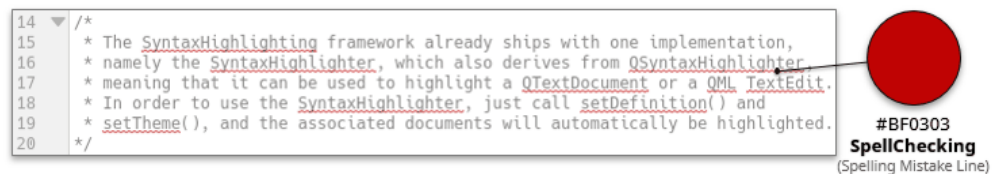
This color is used to highlight to the left of a document lines that have been modified this session and saved. For more information, see Section 3.9.



Text Decorations

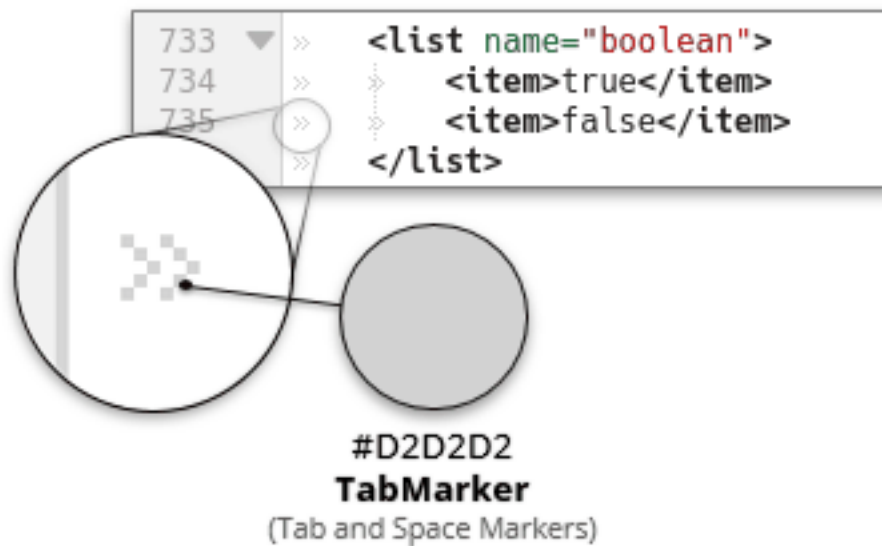
SpellChecking (Spelling Mistake Line)

This color is used to indicate spelling mistakes.



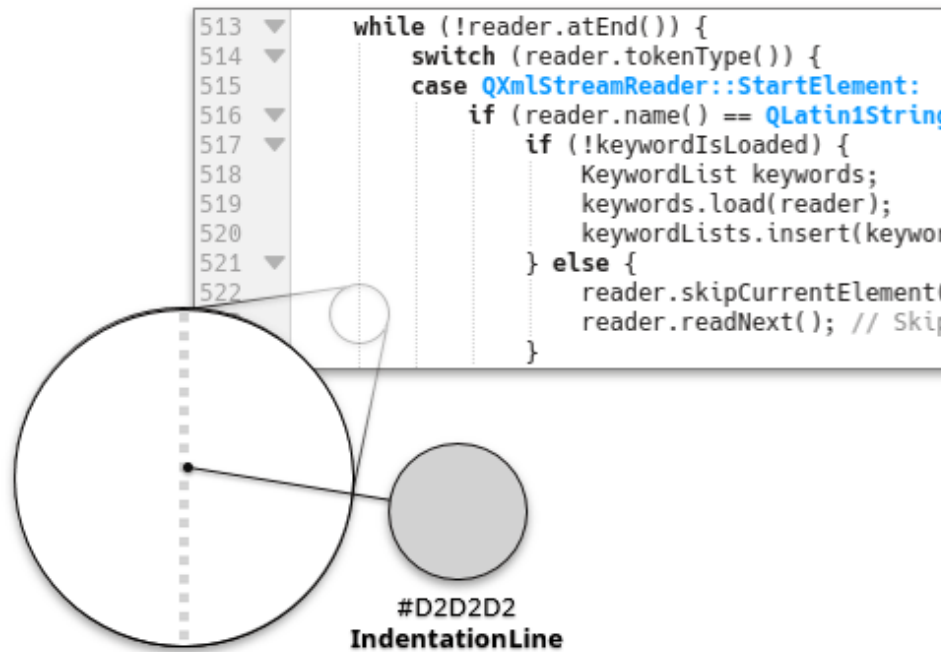
TabMarker (Tab and Space Markers)

This color is used to draw white space indicators, when they are enabled.



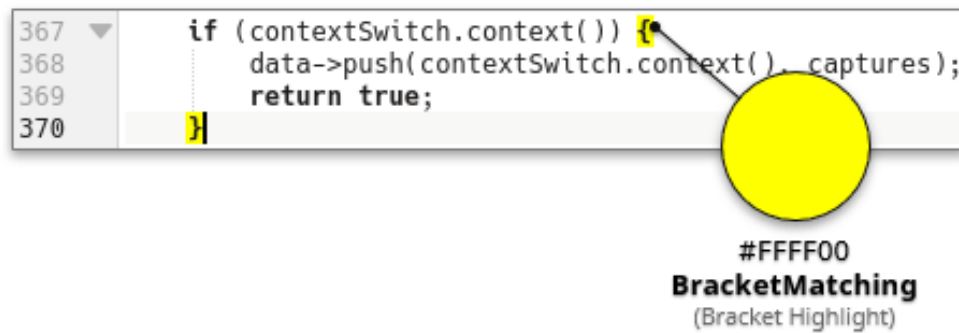
IndentationLine (Indentation Line)

This color is used to draw a line to the left of indented blocks, if [that feature is enabled](#).



BracketMatching (Bracket Highlight)

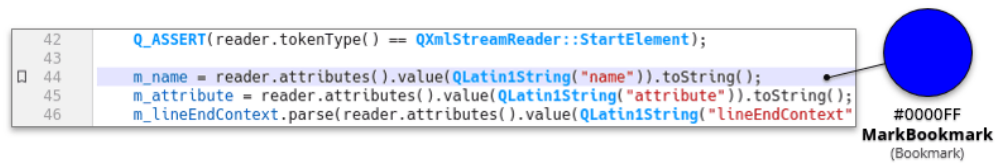
This color is used to draw the background of matching brackets.



Marker Colors

MarkBookmark (Bookmark)

This color is used to indicate bookmarks. Note that this color has an opacity of 22% (and 33% for the current line) with respect to the background. For more information, see Section 3.6.



MarkBreakpointActive (Active Breakpoint)

This color is used by the GDB plugin to indicate an active breakpoint. Notice that this color has an opacity to the background. For more information, see [the GDB Plugin documentation](#).

MarkBreakpointReached (Reached Breakpoint)

This color is used by the GDB plugin to indicate a breakpoint you have reached while debugging. Notice that this color has an opacity to the background. For more information, see [the GDB Plugin documentation](#).

MarkBreakpointDisabled (Disabled Breakpoint)

This color is used by the GDB plugin to indicate an inactive breakpoint. Notice that this color has an opacity to the background. For more information, see [the GDB Plugin documentation](#).

MarkExecution (Execution)

This color is used by the GDB plugin the line presently being executed. Notice that this color has an opacity to the background. For more information, see [the GDB Plugin documentation](#).

MarkWarning (Warning)

This color is used by the build plugin to indicate a line that has caused a compiler warning. Notice that this color has an opacity to the background. For more information, see [the Build Plugin documentation](#).

MarkError (Error)

This color is used by the build plugin to indicate a line that has caused a compiler error. Notice that this color has an opacity to the background. For more information, see [the Build Plugin documentation](#).

Text Templates & Snippets

TemplateBackground (Background)

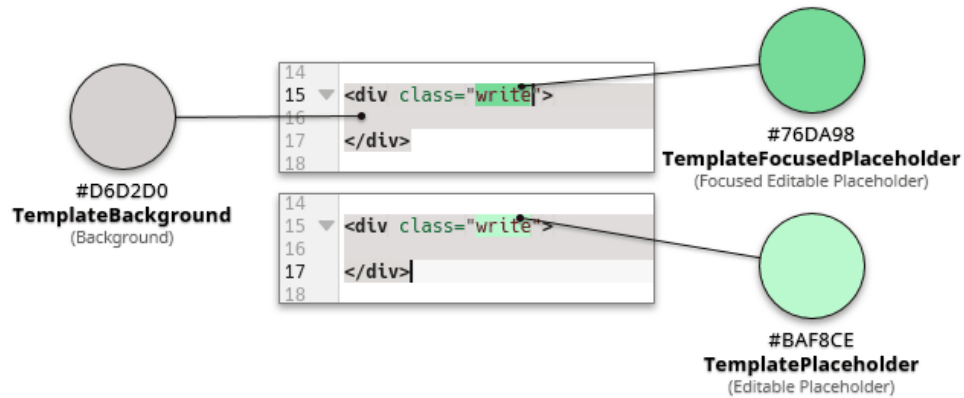
This color is used by the Kate Snippets plugin to mark the background of a snippet. For more information, see [the Kate Snippets documentation](#).

TemplatePlaceholder (Editable Placeholder)

This color is used by the Kate Snippets plugin to mark a placeholder that you can click in to edit manually. For more information, see [the Kate Snippets documentation](#).

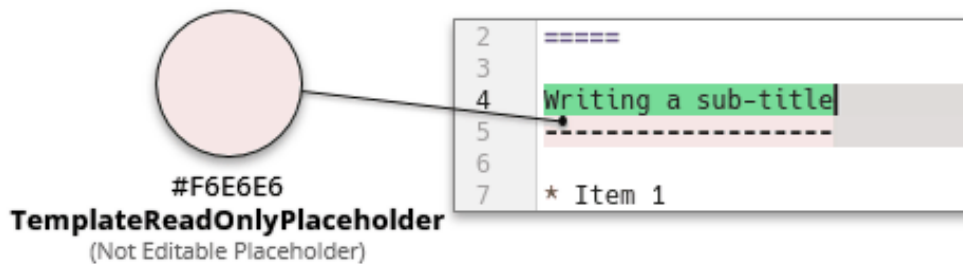
TemplateFocusedPlaceholder (Focused Editable Placeholder)

This color is used by the Kate Snippets plugin to mark the placeholder that you are presently editing. For more information, see [the Kate Snippets documentation](#).



TemplateReadOnlyPlaceholder (Not Editable Placeholder)

This color is used by the Kate Snippets plugin to mark a placeholder that cannot be edited manually, such as one that is automatically populated. For more information, see [the Kate Snippets documentation](#).



6.3.4.2 Default Text Styles

The default text styles are inherited by the highlight text styles, allowing the editor to present text in a very consistent way, for example comment text is using the same style in almost all of the text formats that KSyntaxHighlighting can highlight.

NOTE
 These text styles can be referenced from the **default styles** used in [syntax highlighting](#) definition XML files, for example, the 'Normal' attribute is equivalent to 'dsNormal' in the XML files, and 'DataType' is equivalent to 'dsDataType'. See Section 6.2.3.3 in the syntax highlighting documentation.

TIP

Make sure to choose readable colors with good contrast especially in combination with the **Editor Colors**. See Section 6.3.6.1.

In the **JSON file**, the respective key **text-styles** has as value an *object* where each key corresponds to the name of a *default text style*, which are equivalent to those used in the syntax highlighting definitions. Here, **all available text style keys are mandatory**, these are listed below.

```
"text-styles": {
  "Normal" : {
    "text-color" : "#1f1c1b",
    "selected-text-color" : "#ffffff",
    "bold" : false,
    "italic" : false,
    "underline" : false,
    "strike-through" : false
  },
  "Keyword" : {
    "text-color" : "#1f1c1b",
    "selected-text-color" : "#ffffff",
    "bold" : true
  },
  "Function" : {
    "text-color" : "#644a9b",
    "selected-text-color" : "#452886"
  },
  The other text style keys...
}
```

Each key of *default text style* has a **JSON object** as its value, where values such as *color*, *bold*, *italic*, etc. are specified. These keys are as follows:

text-color: It is a *string* with the text color in hexadecimal color code. This key/value is required.

selected-text-color: The text color when it is selected is generally the same value as 'text-color'. When the text is selected, the background is defined by the value of **TextSelection** in the **Editor Colors**, so you must ensure that the text has good contrast and is readable with this background. The value is a *string* with a hexadecimal color code. This key/value is required.

bold: It is a *boolean* that determines if the text is in bold. This key is optional, the default value is **false**.

italic: It is a *boolean* that determines if the text is curved. This key is optional, the default value is **false**.

underline: It is a *boolean* that determines if the text is underlined. This key is optional, the default value is **false**.

strike-through: It is a *boolean* that determines if the text is strike through. This key is optional, the default value is **false**.

background-color: Determines the background of the text, used for example in alerts in comments. The value is a *string* with a hexadecimal color code. This key is optional, by default there is no background.

selected-background-color: Determines the background of the text when it is selected. The value is a *string* with a hexadecimal color code. This key is optional, by default there is no background.

In the [GUI to manage color themes of KTextEditor](#), these attributes can be modified in the **Default Text Styles** tab. The name in the list of styles is using the style configured for the item, providing you with an immediate preview when configuring a style. Each style lets you select common attributes as well as foreground and background colors. To unset a background color, right-click to use the context menu.

The available text style keys are the following; the keys used in the [JSON file](#) are listed in *bold*, the names used in the [GUI](#) are shown in parentheses if they are different.

Normal Text & Source Code

Normal: Default text style for normal text and source code without special highlighting.

Keyword: Text style for built-in language keywords.

Function: Text style for function definitions and function calls.

Variable: Text style for variables, if applicable. For instance, variables in PHP/Perl typically start with a **\$**, so all identifiers following the pattern **\$foo** are highlighted as variable.

ControlFlow (Control Flow): Text style for control flow keywords, such as *if, then, else, return, switch, break, yield, continue*, etc.

Operator: Text style for operators, such as **+, -, *, / , %**, etc.

BuiltIn (Built-in): Text style for built-in language classes, functions and objects.

Extension: Text style for well-known extensions, such as Qt™ classes, functions/macros in C++ and Python or boost.

Preprocessor: Text style for preprocessor statements or macro definitions.

Attribute: Text style for annotations or attributes of functions or objects, e.g. **@override** in Java, or **__declspec(...)** and **__attribute__((...))** in C++.

Numbers, Types & Constants

DataType (Data Type): Text style for built-in data types such as *int, char, float, void, u64*, etc.

DecVal (Decimal/Value): Text style for decimal values.

BaseN (Base-N Integer): Text style for numbers with base other than 10.

Float (Floating Point): Text style for floating point numbers.

Constant: Text style for language constants and user defined constants, e.g. *True, False, None* in Python or *nullptr* in C/C++; or math constants like *PI*.

Strings & Characters

Char (Character): Text style for single characters such as **'x'**.

SpecialChar (Special Character): Text style for escaped characters in strings, e.g. **'hello\n'**, and other characters with special meaning in strings, such as substitutions or regex operators.

String: Text style for strings like **'hello world'**.

VerbatimString (Verbatim String): Text style for verbatim or raw strings like **'raw \backslashbackslash'** in Perl, CoffeeScript, and shells, as well as **r'\raw'** in Python, or such as HERE docs.

SpecialString (Special String): Text style for special strings, such as regular expressions in ECMAScript, the L^AT_EX math mode, SQL, etc.

Import (Imports, Modules, Includes): Text style for includes, imports, modules or L^AT_EX packages.

Comments & Documentation

Comment: Text style for normal comments.

Documentation: Text style for comments that reflect API documentation, such as **/** doxygen comments */** or **"""docstrings"""**.

Annotation: Text style for annotations in comments or documentation commands, such as `@param` in Doxygen or JavaDoc.

CommentVar (Comment Variable): Text style that refers to variables names used in above commands in a comment, such as `foobar` in `'@param foobar'`, in Doxygen or JavaDoc.

RegionMarker (Region Marker): Text style for region markers, typically defined by `/ /BEGIN` and `/ /END` in comments.

Information: Text style for information, notes and tips, such as the keyword `@note` in Doxygen.

Warning: Text style for warnings, such as the keyword `@warning` in Doxygen.

Alert: Text style for special words in comments, such as `TODO`, `FIXME`, `XXXX` and `WARNING`.

Miscellaneous

Error: Text style indicating error highlighting and wrong syntax.

Others: Text style for attributes that do not match any of the other default styles.

6.3.4.3 Custom Highlighting Text Styles

Here you can establish text styles for a specific syntax highlighting definition, overriding the **default text style** described in [the previous section](#).

In the [JSON theme file](#), this corresponds to the `custom-styles` key, whose value is an *object* where each subschema key corresponds to the **name of a syntax highlighting definition**. Its value is an *object* where each key refers to the **style attributes name** defined in the [itemData elements](#) of the syntax highlighting XML file, and the respective value is a sub-object with the keys `text-color`, `selected-text-color`, `bold`, `italic`, `underline`, `strike-through`, `background-color` and `selected-background-color`, defined in [the previous section](#). Each of these values are optional, since if they are not present, the style set in `text-styles` is considered.

For example, in this piece of code, the 'ISO C++' syntax highlighting definition has a special text style for the 'Type Modifiers' and 'Standard Classes' attributes. In the corresponding XML file 'isocpp.xml', the defined attribute 'Standard Classes' uses the default style `BuiltIn` (or `dsBuiltIn`). In this attribute, only the value of `text-color` is overwritten by the new color '#6431b3'.

```
"custom-styles": {
  "ISO C++": {
    "Standard Classes": {
      "text-color": "#6431b3"
    },
    "Type Modifiers": {
      "bold": true,
      "selected-text-color": "#009183",
      "text-color": "#00b5cf"
    }
  }
}
```

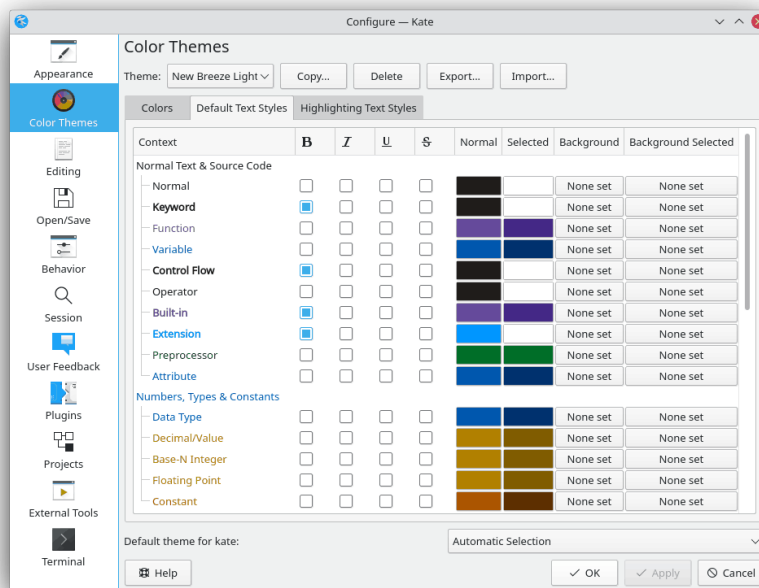
NOTE

- You should consider that these text styles are associated with the attribute names defined in the syntax highlighting XML files. If an XML file is updated and some attributes are renamed or removed, the custom style defined in the theme will no longer apply.
- Syntax highlighting definitions often include other definitions. For example, the 'QML' highlighter includes the 'JavaScript' highlighter, since they share functionality in highlighting.

In the GUI to manage themes of `KTextEditor`, these attributes can be modified in the **Highlighting Text Styles** tab. By default, the editor preselects the highlighting of the current document. You will notice that many highlights contain other highlights represented by groups in the style list. For example most highlights import the 'Alert' highlight, and many source code formats import the 'Doxygen' highlight.

6.3.5 The Color Themes GUI

The simplest way to create and edit color themes is through the GUI within the 'Configure' dialog provided by `KTextEditor`. To access it, select **Settings** → **Configure Application...** from the menubar in your text editor. This brings up the **Configure** dialog box, there select **Color Themes** in the side panel.



Kate's settings dialog box with the color theme management.

In this [dialog](#) you can configure all the colors in any theme you have, as well as create/copy new themes, delete them, export them to a `.theme` file with [JSON format](#) or import them from external `.theme` files. Each theme has settings for text colors and styles.

The built-in themes cannot be modified by default. To do this, you need to copy them and give them a new name.

To use a theme permanently in your text editor, you must select it in the combobox labeled **Default theme for Application** at the bottom of the dialog and press **Apply** or **OK**. By default, the **Automatic Selection** option is active, which chooses a more appropriate color theme according to the *KDE Plasma color scheme* used in the text editing application; it usually chooses between 'Breeze Light' and 'Breeze Dark' if the scheme is light or dark, respectively.

TIP

You can adjust the KDE global color scheme in the [Colors module in System Settings](#). You can also change it in some applications individually such as Kate or KDevelop, from the menu **Settings** → **Color Scheme**.

6.3.5.1 Create a new theme

To create a new theme, it is first necessary to copy an existing one. Select an existing theme which you want to use as a base, such as 'Breeze Light' or 'Breeze Dark', and click **Copy**. Then write a name for the new theme.

If you want to modify a built-in or read-only theme, you must first copy it under a different name.

6.3.5.2 Import or export JSON theme files

You can export a selected theme (including built-in ones) to a [JSON file](#) with `.theme` extension, with the **Export** button. This will open a dialog to save the file. To add a color theme from an external [JSON file](#), just press the **Import** button and select the `.theme` file from the dialog.

TIP

- As [mentioned above](#), user-customized theme files are stored in the `org.kde.syntax-highlighting/themes/` directory. When you copy or create a theme, it will automatically appear there. Also, importing or adding a theme is equivalent to copying an external `.theme` file into this directory. `KSyntaxHighlighting` automatically picks up color theme files from this directory.
- If you want to publish a theme created by you, it is essential to check the [metadata](#) object of the [JSON file](#), adding the respective license and checking the revision number.

6.3.5.3 Editing color themes

6.3.5.3.1 Colors

Here the colors of the text editing area are adjusted. These are detailed in [Section 6.3.4.1](#).

6.3.5.3.2 Default Text Styles

The default text styles are inherited by the highlight text styles, allowing the editor to present text in a very consistent way, for example comment text is using the same style in almost all of the text formats that `KSyntaxHighlighting` can highlight.

The name in the list of styles is using the style configured for the item, providing you with an immediate preview when configuring a style.

Each style lets you select common attributes as well as foreground and background colors. To unset a background color, right-click to use the context menu.

The attributes of this area are detailed in [Section 6.3.4.2](#).

6.3.5.3.3 Highlighting Text Styles

Here you can edit the text styles used by a specific highlight definition. The editor preselects the highlight used by your current document. To work on a different highlight, select one in the **Highlight** combobox above the style list.

The name in the list of styles is using the style configured for the item, providing you with an immediate preview when configuring a style.

Each style lets you select common attributes as well as foreground and background colors. To unset a background color, right-click to use the context menu. In addition you can see if a style is equal to the default style used for the item, and set it to that if not.

You will notice that many highlights contain other highlights represented by groups in the style list. For example most highlights import the Alert highlight, and many source code formats imports the Doxygen highlight. Editing colors in those groups only affects the styles when used in the edited highlight format.

6.3.6 Tips & Tricks

6.3.6.1 Contrast of Text Colors

An important aspect when working with color themes is to choose a text contrast that makes it easier to read, especially in combination with the background.

The **Kontrast** application is a color contrast checker. It tells you that the text color and background color combinations are readable and accessible, so this is an excellent tool to help you create color themes.

You can download **Kontrast** from [the KDE Applications website](#) or from [the Flatpak package on Flathub](#) (only in GNU/Linux).

The GNOME **Contrast** application is similar. You can download [the Flatpak package on Flathub](#) (only in GNU/Linux).

6.3.6.2 Suggestions of Consistency with Syntax Highlighting

KSyntaxHighlighting includes [more than 300 syntax highlighting definitions](#), therefore it is ideal that you make sure your new theme looks good in all syntax highlighting definitions. The built-in color themes have the following similarities that it is recommended (but not required) to follow to achieve a correct display of all syntax highlighting definitions:

- Use bold for the 'Keyword' and 'ControlFlow' [text styles](#).
- Do not use background color in any [text style](#), except 'Alert' and 'RegionMarker'.

Most of the syntax highlighters are intended to look good on the default themes 'Breeze Light' and 'Breeze Dark', therefore, another way to maintain consistency is to use similar colors in the [text styles](#), like *green* for 'Preprocessor' and 'Others', *blue* for 'DataType' and 'Attribute', or *purple* for 'Function'.

Note that these recommendations are not mandatory when creating and publishing a theme.

6.4 Scripting with JavaScript

The KatePart editor component is easily extensible by writing scripts. The scripting language is ECMAScript (widely known as JavaScript). KatePart supports two kinds of scripts: indentation and command line scripts.

6.4.1 Indentation Scripts

Indentation scripts - also referred as indenters - automatically indent the source code while typing text. As an example, after hitting the return key the indentation level often increases.

The following sections describe step by step how to create the skeleton for a simple indenter. As a first step, create a new *.js file called e.g. javascript.js in the local home folder `$XDG_DATA_HOME /katepart5/script/indentation`. Therein, the environment variable `XDG_DATA_HOME` typically expands to either `~/ .local` or `~/ .local/share`.

On Windows[®] these files are located in `%USERPROFILE%\AppData\Local\katepart5\script\indentation`. `%USERPROFILE%` usually expands to `C:\\Users\\user`.

6.4.1.1 The Indentation Script Header

The header of the file `javascript.js` is embedded as JSON at the beginning of the document as follows:

```
var katescript = {
  "name": "JavaScript",
  "author": "Example Name <example.name@some.address.org>",
  "license": "BSD License",
  "revision": 1,
  "kate-version": "5.1",
  "required-syntax-style": "javascript",
  "indent-languages": ["javascript"],
  "priority": 0,
}; // kate-script-header, must be at the start of the file without comments
```

Each entry is explained in detail now:

- `name` [required]: This is the indenter name that appears in the menu **Tools** → **Indentation** and in the configuration dialog.
- `author` [optional]: The author's name and contact information.
- `license` [optional]: Short form of the license, such as BSD License or LGPLv3.
- `revision` [required]: The revision of the script. This number should be increased whenever the script is modified.
- `kate-version` [required]: Minimum required KatePart version.
- `required-syntax-style` [optional]: The required syntax style, which matches the specified `style` in syntax highlighting files. This is important for indenters that rely on specific highlight information in the document. If a required syntax style is specified, the indenter is available only when the appropriate highlighter is active. This prevents 'undefined behavior' caused by using the indenter without the expected highlighting schema. For instance, the Ruby indenter makes use of this in the files `ruby.js` and `ruby.xml`.
- `indent-languages` [optional]: JSON array of syntax styles the indenter can indent correctly, e.g.: `["c++", "java"]`.
- `priority` [optional]: If several indenters are suited for a certain highlighted file, the priority decides which indenter is chosen as default indenter.

6.4.1.2 The Indenter Source Code

Having specified the header this section explains how the indentation scripting itself works. The basic skeleton of the body looks like this:

```
// required katepart js libraries, e.g. range.js if you use Range
require ("range.js");

triggerCharacters = "{}/:;";
function indent(line, indentWidth, ch)
{
  // called for each newline (ch == '\n') and all characters specified in
  // the global variable triggerCharacters. When calling ToolsFormat ↵
  Indentation
  // the variable ch is empty, i.e. ch == ''.
  //
  // see also: Scripting API
```

```
return -2;
}
```

The function `indent()` has three parameters:

- `line`: the line that has to be indented
- `indentWidth`: the indentation width in number of spaces
- `ch`: either a newline character (`ch == '\n'`), the trigger character specified in `triggerCharacters` or empty if the user invoked the action **Tools** → **Format Indentation**.

The return value of the `indent()` function specifies how the line will be indented. If the return value is a simple integer number, it is interpreted as follows:

- return value `-2`: do nothing
- return value `-1`: keep indentation (searches for previous non-blank line)
- return value `0`: numbers `>= 0` specify the indentation depth in spaces

Alternatively, an array of two elements can be returned:

- `return [indent, align];`

In this case, the first element is the indentation depth as above with the same meaning of the special values. However, the second element is an absolute value representing a column for 'alignment'. If this value is higher than the indent value, the difference represents a number of spaces to be added after the indentation of the first parameter. Otherwise, the second number is ignored. Using tabs and spaces for indentation is often referred to as 'mixed mode'.

Consider the following example: Assume using tabs to indent, and tab width is set to 4. Here, `<tab>` represents a tab and `' '` a space:

```
1: <tab><tab>foobar("hello",
2: <tab><tab>....."world");
```

When indenting line 2, the `indent()` function returns `[8, 15]`. As result, two tabs are inserted to indent to column 8, and 7 spaces are added to align the second parameter under the first, so that it stays aligned if the file is viewed with a different tab width.

A default KDE installation ships KatePart with several indenters. The corresponding JavaScript source code can be found in `$XDG_DATA_DIRS /katepart5/script/indentation`.

On Windows® these files are located in `%USERPROFILE%\AppData\Local\katepart5\script\indentation`. `%USERPROFILE%` usually expands to `C:\Users\user`.

Developing an indenter requires reloading the scripts to see whether the changes behave appropriately. Instead of restarting the application, simply switch to the command line and invoke the command **reload-scripts**.

If you develop useful scripts please consider contributing to the KatePart Project by [contacting the mailing list](#).

6.4.2 Command Line Scripts

As it is hard to satisfy everyone's needs, KatePart supports little helper tools for quick text manipulation through the [built-in command line](#). For instance, the command `sort` is implemented as a script. This section explains how to create `*.js` files to extend KatePart with arbitrary helper scripts.

Command line scripts are located in the same folder as indentation scripts. So as a first step, create a new `*.js` file called `myutils.js` in the local home folder `$XDG_DATA_HOME /katepart5/script/commands`. Therein, the environment variable `XDG_DATA_HOME` typically expands to either `~/.local` or `~/.local/share`.

On Windows[®] these files are located in `%USERPROFILE%\AppData\Local\katepart5\script\commands`. `%USERPROFILE%` usually expands to `C:\\Users\\user`.

6.4.2.1 The Command Line Script Header

The header of each command line script is embedded in JSON at the beginning of the script as follows:

```
var katescript = {
  "author": "Example Name <example.name@some.address.org>",
  "license": "LGPLv2+",
  "revision": 1,
  "kate-version": "5.1",
  "functions": ["sort", "moveLinesDown"],
  "actions": [
    {
      "function": "sort",
      "name": "Sort Selected Text",
      "category": "Editing",
      "interactive": "false"
    },
    {
      "function": "moveLinesDown",
      "name": "Move Lines Down",
      "category": "Editing",
      "shortcut": "Ctrl+Shift+Down",
      "interactive": "false"
    }
  ]
}; // kate-script-header, must be at the start of the file without comments
```

Each entry is explained in detail now:

- `author` [optional]: The author's name and contact information.
- `license` [optional]: Short form of the license, such as BSD License or LGPLv2.
- `revision` [required]: The revision of the script. This number should be increased whenever the script is modified.
- `kate-version` [required]: Minimum required KatePart version.
- `functions` [required]: JSON array of commands in the script.
- `actions` [optional]: JSON Array of JSON objects that defines the actions that appear in the application menu. Detailed information is provided in the section [Binding Shortcuts](#).

Since the value of `functions` is a JSON array, a single script is able to contain an arbitrary number of command line commands. Each function is available through KatePart's [built-in command line](#).

6.4.2.2 The Script Source Code

All functions specified in the header have to be implemented in the script. For instance, the script file from the example above needs to implement the two functions `sort` and `moveLinesDown`. All functions have the following syntax:

```
// required katepart js libraries, e.g. range.js if you use Range
require ("range.js");

function <name>(arg1, arg2, ...)
{
    // ... implementation, see also: Scripting API
}
```

Arguments in the command line are passed to the function as `arg1`, `arg2`, etc. In order to provide documentation for each command, simply implement the 'help' function as follows:

```
function help(cmd)
{
    if (cmd == "sort") {
        return i18n("Sort the selected text.");
    } else if (cmd == "...") {
        // ...
    }
}
```

Executing `help sort` in the command line then calls this help function with the argument `cmd` set to the given command, i.e. `cmd == "sort"`. KatePart then presents the returned text as documentation to the user. Make sure to [translate the strings](#).

Developing a command line script requires reloading the scripts to see whether the changes behave appropriately. Instead of restarting the application, simply switch to the command line and invoke the command `reload-scripts`.

6.4.2.2.1 Binding Shortcuts

In order to make the scripts accessible in the application menu and assign shortcuts, the script needs to provide an appropriate script header. In the above example, both functions `sort` and `moveLinesDown` appear in the menu due to the following part in the script header:

```
var katescript = {
    ...
    "actions": [
        { "function": "sort",
          "name": "Sort Selected Text",
          "icon": "",
          "category": "Editing",
          "interactive": "false"
        },
        { "function": "moveLinesDown",
          "name": "Move Lines Down",
          "icon": "",
          "category": "Editing",
          "shortcut": "Ctrl+Shift+Down",
          "interactive": "false"
        }
    ]
};
```

The fields for one action are as follows:

- `function` [required]: The function that should appear in the menu **Tools** → **Scripts**.
- `name` [required]: The text appears in the script menu.
- `icon` [optional]: The icon appears next to the text in the menu. All KDE icon names can be used here.
- `category` [optional]: If a category is specified, the script appears in a submenu.
- `shortcut` [optional]: The shortcut given here is the default shortcut. Example: `Ctrl+Alt+t`. See the [Qt documentation](#) for further details.
- `interactive` [optional]: If the script needs user input in the command line, set this to `true`.

If you develop useful scripts please consider contributing to the KatePart Project by [contacting the mailing list](#).

6.4.3 Scripting API

The scripting API presented here is available to all scripts, i.e. indentation scripts and command line commands. The `Cursor` and `Range` classes are provided by library files in `$XDG_DATA_DIRS/katepart5/libraries`. If you want to use them in your script, which needs to use some of the `Document` or `View` functions, please include the necessary library by using:

```
// required katepart js libraries, e.g. range.js if you use Range
require ("range.js");
```

To extend the standard scripting API with your own functions and prototypes simply create a new file in KDE's local configuration folder `$XDG_DATA_HOME /katepart5/libraries` and include it into your script using:

```
require ("myscriptnamehere.js");
```

On Windows® these files are located in `%USERPROFILE%\AppData\Local\katepart5\libraries`. `%USERPROFILE%` usually expands to `C:\Users\user`.

To extend existing prototypes like `Cursor` or `Range`, the recommended way is to *not* modify the global `*.js` files. Instead, change the `Cursor` prototype in JavaScript after the `cursor.js` is included into your script via `require`.

6.4.3.1 Cursors and Ranges

As KatePart is a text editor, all the scripting API is based on cursors and ranges whenever possible. A `Cursor` is a simple `(line, column)` tuple representing a text position in the document. A `Range` spans text from a starting cursor position to an ending cursor position. The API is explained in detail in the next sections.

6.4.3.1.1 The Cursor Prototype

Cursor ();

Constructor. Returns a `Cursor` at position `(0, 0)`.

Example: `var cursor = new Cursor();`

Cursor(int line, int column);

Constructor. Returns a Cursor at position (line, column).

Example: `var cursor = new Cursor(3, 42);`

Cursor(Cursor other);

Copy constructor. Returns a copy of the cursor *other*.

Example: `var copy = new Cursor(other);`

Cursor Cursor.clone();

Returns a clone of the cursor.

Example: `var clone = cursor.clone();`

Cursor.setPosition(int line, int column);

Sets the cursor position to *line* and *column*.

Since: KDE 4.11

bool Cursor.isValid();

Check whether the cursor is valid. The cursor is invalid, if line and/or column are set to -1.

Example: `var valid = cursor.isValid();`

Cursor Cursor.invalid();

Returns a new invalid cursor located at (-1, -1).

Example: `var invalidCursor = cursor.invalid();`

int Cursor.compareTo(Cursor other);

Compares this cursor to the cursor *other*. Returns

- -1, if this cursor is located before the cursor *other*,
- 0, if both cursors equal and
- +1, if this cursor is located after the cursor *other*.

bool Cursor.equals(Cursor other);

Returns true, if this cursor and the cursor *other* are equal, otherwise false.

String Cursor.toString();

Returns the cursor as a string of the form 'Cursor(line, column)'.

6.4.3.1.2 The Range Prototype

Range();

Constructor. Calling `new Range()` returns a Range at (0, 0) - (0, 0).

Range(Cursor start, Cursor end);

Constructor. Calling `new Range(start, end)` returns the Range (*start*, *end*).

Range(int startLine, int startColumn, int endLine, int endColumn);

Constructor. Calling `new Range(startLine, startColumn, endLine, endColumn)` returns the Range from (*startLine*, *startColumn*) to (*endLine*, *endColumn*).

Range(Range other);

Copy constructor. Returns a copy of Range *other*.

Range Range.clone();

Returns a clone of the range.

Example: `var clone = range.clone();`

bool Range.isEmpty();

Returns `true`, if the start and end cursors are equal.

Example: `var empty = range.isEmpty();`

Since: KDE 4.11

bool Range.isValid();

Returns `true`, if both start and end cursor are valid, otherwise `false`.

Example: `var valid = range.isValid();`

Range Range.invalid();

Returns the Range from (-1, -1) to (-1, -1).

bool Range.contains(Cursor cursor);

Returns `true`, if this range contains the cursor position, otherwise `false`.

bool Range.contains(Range other);

Returns `true`, if this range contains the Range *other*, otherwise `false`.

bool Range.containsColumn(int column);

Returns `true`, if *column* is in the half open interval `[start.column, end.column)`, otherwise `false`.

bool Range.containsLine(int line);

Returns `true`, if *line* is in the half open interval `[start.line, end.line)`, otherwise `false`.

bool Range.overlaps(Range other);

Returns `true`, if this range and the range *other* share a common region, otherwise `false`.

bool Range.overlapsLine(int line);

Returns `true`, if *line* is in the interval `[start.line, end.line]`, otherwise `false`.

bool Range.overlapsColumn(int column);

Returns `true`, if *column* is in the interval `[start.column, end.column]`, otherwise `false`.

bool Range.onSingleLine();

Returns `true`, if the range starts and ends at the same line, i.e. if `Range.start.line == Range.end.line`.

Since: KDE 4.9

bool Range.equals(Range other);

Returns `true`, if this range and the Range *other* are equal, otherwise `false`.

String Range.toString();

Returns the range as a string of the form `'Range(Cursor(line, column), Cursor(line, column))'`.

6.4.3.2 Global Functions

This section lists all global functions.

6.4.3.2.1 Reading & Including Files

String read(*String* file);

Will search the given *file* relative to the `katepart5/script/files` directory and return its content as a string.

void require(*String* file);

Will search the given *file* relative to the `katepart5/script/libraries` directory and evaluate it. `require` is internally guarded against multiple inclusions of the same *file*.

Since: KDE 4.10

6.4.3.2.2 Debugging

void debug(*String* text);

Prints *text* to `stdout` in the console launching the application.

6.4.3.2.3 Translation

In order to support full localization, there are several functions to translate strings in scripts, namely `i18n`, `i18nc`, `i18np` and `i18ncp`. These functions behave exactly like [KDE's translation functions](#).

The translation functions translate the wrapped strings through KDE's translation system to the language used in the application. Strings in scripts being developed in the official KatePart sources are automatically extracted and translatable. In other words, as a KatePart developer you do not have to bother with message extraction and translation. It should be noted though, that the translation only works inside the KDE infrastructure, i.e., new strings in 3rd-party scripts developed outside of KDE are not translated. Therefore, please consider contributing your scripts to Kate such that proper translation is possible.

void i18n(*String* text, *arg1*, ...);

Translates *text* into the language used by the application. The arguments *arg1*, ..., are optional and used to replace the placeholders `%1`, `%2`, etc.

void i18nc(*String* context, *String* text, *arg1*, ...);

Translates *text* into the language used by the application. Additionally, the string *context* is visible to translators so they can provide a better translation. The arguments *arg1*, ..., are optional and used to replace the placeholders `%1`, `%2`, etc.

void i18np(*String* singular, *String* plural, *int* number, *arg1*, ...);

Translates either *singular* or *plural* into the language used by the application, depending on the given *number*. The arguments *arg1*, ..., are optional and used to replace the placeholders `%1`, `%2`, etc.

void i18ncp(*String* context, *String* singular, *String* plural, *int* number, *arg1*, ...);

Translates either *singular* or *plural* into the language used by the application, depending on the given *number*. Additionally, the string *context* is visible to translators so they can provide a better translation. The arguments *arg1*, ..., are optional and used to replace the placeholders `%1`, `%2`, etc.

6.4.3.3 The View API

Whenever a script is being executed, there is a global variable 'view' representing the current active editor view. The following is a list of all available View functions.

void view.copy()

Copy the selection if there is one, otherwise the current line if the option [] **Copy/Cut the current line if no selection** is set.

Since: KDE Frameworks 5.79

void view.cut()

Cut the selection if there is one, otherwise the current line if the option [] **Copy/Cut the current line if no selection** is set.

Since: KDE Frameworks 5.79

void view.paste()

Paste the clipboard content.

Since: KDE Frameworks 5.79

Cursor view.cursorPosition()

Returns the current cursor position in the view.

void view.setCursorPosition(int line, int column); void view.setCursorPosition(Cursor cursor);

Set the current cursor position to either (line, column) or to the given cursor.

Cursor view.virtualCursorPosition();

Returns the virtual cursor position with each tab counting the corresponding amount of spaces depending on the current tab width.

void view.setVirtualCursorPosition(int line, int column); void view.setVirtualCursorPosition(Cursor cursor);

Set the current virtual cursor position to (line, column) or to the given cursor.

String view.selectedText();

Returns the selected text. If no text is selected, the returned string is empty.

bool view.hasSelection();

Returns true, if the view has selected text, otherwise false.

Range view.selection();

Returns the selected text range. The returned range is invalid if there is no selected text.

void view.setSelection(Range range);

Set the selected text to the given range.

void view.removeSelectedText();

Remove the selected text. If the view does not have any selected text, this does nothing.

void view.selectAll();

Selects the entire text in the document.

void view.clearSelection();

Clears the text selection without removing the text.

void view.setBlockSelection(bool on);

Set block selection mode on or off.

bool view.blockSelection();

Returns `true`, if block selection mode is on, otherwise `false`.

void view.align(Range range);

Properly re-indent lines within `range` according to current indentation settings.

void view.alignOn(Range range, String pattern = "");

Aligns lines in `range` on the column given by the regular expression `pattern`. With an empty `pattern` it will align on the first non-blank character by default. If the pattern has a capture it will indent on the captured match.

Examples:

`view.alignOn(document.documentRange(), '-');` will insert spaces before the first - of each lines to align them all on the same column.

`view.alignOn(document.documentRange(), ':\s+(.)');` will insert spaces before the first non-blank character that occurs after a colon to align them all on the same column.

object view.executeCommand(String command, String args, Range range);

Executes the [command line command](#) `command` with the optional arguments `args` and the optional `range`. The returned `object` has a boolean property `object.ok` that indicates whether execution of the `command` was successful. In case of an error, the string `object.status` contains an error message.

Since: KDE Frameworks 5.50

Range view.searchText(Range range, String pattern, bool backwards = false);

Search for the first occurrence of `pattern` in `range` and returns the matched range. Search is performed backwards if the optional boolean parameter `backwards` is set to `true`.

The returned range is invalid (see `Range.isValid()`) if `pattern` is not found in `range`.

Since: KDE Frameworks 5.97

6.4.3.4 The Document API

Whenever a script is being executed, there is a global variable 'document' representing the current active document. The following is a list of all available Document functions.

String document.fileName();

Returns the document's filename or an empty string for unsaved text buffers.

String document.url();

Returns the document's full URL or an empty string for unsaved text buffers.

String document.mimeType();

Returns the document's MIME type or the MIME type `application/octet-stream` if no appropriate MIME type could be found.

String document.encoding();

Returns the currently used encoding to save the file.

String document.highlightingMode();

Returns the global highlighting mode used for the whole document.

String document.highlightingModeAt(Cursor pos);

Returns the highlighting mode used at the given position in the document.

Array document.embeddedHighlightingModes();

Returns an array of highlighting modes embedded in this document.

bool document.isModified();

Returns `true`, if the document has unsaved changes (modified), otherwise `false`.

String document.text();

Returns the entire content of the document in a single text string. Newlines are marked with the newline character `'\n'`.

String document.text(int fromLine, int fromColumn, int toLine, int toColumn); **String document.text(Cursor from, Cursor to);** **String document.text(Range range);**

Returns the text in the given range. It is recommended to use the cursor and range based version for better readability of the source code.

String document.line(int line);

Returns the given text line as string. The string is empty if the requested line is out of range.

String document.wordAt(int line, int column); **String document.wordAt(Cursor cursor);**

Returns the word at the given cursor position.

Range document.wordRangeAt(int line, int column); **Range document.wordRangeAt(Cursor cursor);**

Return the range of the word at the given cursor position. The returned range is invalid (see `Range.isValid()`), if the text position is after the end of a line. If there is no word at the given cursor, an empty range is returned.

Since: KDE 4.9

String document.charAt(int line, int column); **String document.charAt(Cursor cursor);**

Returns the character at the given cursor position.

String document.firstChar(int line);

Returns the first character in the given `line` that is not a whitespace. The first character is at column 0. If the line is empty or only contains whitespace characters, the returned string is empty.

String document.lastChar(int line);

Returns the last character in the given `line` that is not a whitespace. If the line is empty or only contains whitespace characters, the returned string is empty.

bool document.isSpace(int line, int column); **bool document.isSpace(Cursor cursor);**

Returns `true`, if the character at the given cursor position is a whitespace, otherwise `false`.

bool document.matchesAt(int line, int column, String text); **bool document.matchesAt(Cursor cursor, String text);**

Returns `true`, if the given `text` matches at the corresponding cursor position, otherwise `false`.

bool document.startsWith(int line, String text, bool skipWhiteSpaces);

Returns `true`, if the line starts with `text`, otherwise `false`. The argument `skipWhiteSpaces` controls whether leading whitespaces are ignored.

bool document.endsWith(int line, String text, bool skipWhiteSpaces);

Returns `true`, if the line ends with `text`, otherwise `false`. The argument `skipWhiteSpaces` controls whether trailing whitespaces are ignored.

bool document.setText(String text);

Sets the entire document text.

bool document.clear();

Removes the entire text in the document.

bool document.truncate(int line, int column); bool document.truncate(Cursor cursor);

Truncate the given line at the given column or cursor position. Returns `true` on success, or `false` if the given line is not part of the document range.

bool document.insertText(int line, int column, String text); bool document.insertText(Cursor cursor, String text);

Inserts the `text` at the given cursor position. Returns `true` on success, or `false`, if the document is in read-only mode.

bool document.removeText(int fromLine, int fromColumn, int toLine, int toColumn); bool document.removeText(Cursor from, Cursor to); bool document.removeText(Range range);

Removes the text in the given range. Returns `true` on success, or `false`, if the document is in read-only mode.

bool document.insertLine(int line, String text);

Inserts text in the given line. Returns `true` on success, or `false`, if the document is in read-only mode or the line is not in the document range.

bool document.removeLine(int line);

Removes the given text line. Returns `true` on success, or `false`, if the document is in read-only mode or the line is not in the document range.

bool document.wrapLine(int line, int column); bool document.wrapLine(Cursor cursor);

Wraps the line at the given cursor position. Returns `true` on success, otherwise `false`, e.g. if `line < 0`.

Since: KDE 4.9

void document.joinLines(int startLine, int endLine);

Joins the lines from `startLine` to `endLine`. Two succeeding text lines are always separated with a single space.

int document.lines();

Returns the number of lines in the document.

bool document.isLineModified(int line);

Returns `true`, if `line` currently contains unsaved data.

Since: KDE 5.0

bool document.isLineSaved(int line);

Returns `true`, if `line` was changed, but the document was saved. Hence, the line currently does not contain any unsaved data.

Since: KDE 5.0

bool document.isLineTouched(int line);

Returns `true`, if `line` currently contains unsaved data or was changed before.

Since: KDE 5.0

bool document.findTouchedLine(int startLine, bool down);

Search for the next touched line starting at `line`. The search is performed either upwards or downwards depending on the search direction specified in `down`.

Since: KDE 5.0

The KatePart Handbook

int document.length();

Returns the number of characters in the document.

int document.lineLength(int line);

Returns the *line's* length.

void document.editBegin();

Starts an edit group for undo/redo grouping. Make sure to always call `editEnd()` as often as you call `editBegin()`. Calling `editBegin()` internally uses a reference counter, i.e., this call can be nested.

void document.editEnd();

Ends an edit group. The last call of `editEnd()` (i.e. the one for the first call of `editBegin()`) finishes the edit step.

int document.firstColumn(int line);

Returns the first non-whitespace column in the given *line*. If there are only whitespaces in the line, the return value is `-1`.

int document.lastColumn(int line);

Returns the last non-whitespace column in the given *line*. If there are only whitespaces in the line, the return value is `-1`.

**int document.prevNonSpaceColumn(int line, int column); int
document.prevNonSpaceColumn(Cursor cursor);**

Returns the column with a non-whitespace character starting at the given cursor position and searching backwards.

**int document.nextNonSpaceColumn(int line, int column); int
document.nextNonSpaceColumn(Cursor cursor);**

Returns the column with a non-whitespace character starting at the given cursor position and searching forwards.

int document.prevNonEmptyLine(int line);

Returns the next non-empty line containing non-whitespace characters searching backwards.

int document.nextNonEmptyLine(int line);

Returns the next non-empty line containing non-whitespace characters searching forwards.

bool document.isInWord(String character, int attribute);

Returns `true`, if the given *character* with the given *attribute* can be part of a word, otherwise `false`.

bool document.canBreakAt(String character, int attribute);

Returns `true`, if the given *character* with the given *attribute* is suited to wrap a line, otherwise `false`.

bool document.canComment(int startAttribute, int endAttribute);

Returns `true`, if a range starting and ending with the given attributes is suited to be commented out, otherwise `false`.

String document.commentMarker(int attribute);

Returns the comment marker for single line comments for a given *attribute*.

String document.commentStart(int attribute);

Returns the comment marker for the start of multi-line comments for a given *attribute*.

String document.commentEnd(int attribute);

Returns the comment marker for the end of multi-line comments for a given *attribute*.

Range `document.documentRange()`;

Returns a range that encompasses the whole document.

Cursor `documentEnd()`;

Returns a cursor positioned at the last column of the last line in the document.

bool `isValidTextPosition(int line, int column)`; **bool**
`isValidTextPosition(Cursor cursor)`;

Returns `true`, if the given cursor position is positioned at a valid text position. A text position is valid only if it locate at the start, in the middle, or the end of a valid line. Further, a text position is invalid if it is located in a Unicode surrogate.

Since: KDE 5.0

int `document.attribute(int line, int column)`; **int** `document.attribute(Cursor cursor)`;

Returns the attribute at the given cursor position.

bool `document.isAttribute(int line, int column, int attribute)`; **bool**
`document.isAttribute(Cursor cursor, int attribute)`;

Returns `true`, if the attribute at the given cursor position equals `attribute`, otherwise `false`.

String `document.attributeName(int line, int column)`; **String**
`document.attributeName(Cursor cursor)`;

Returns the attribute name as human readable text. This is equal to the `itemData` name in the syntax highlighting files.

bool `document.isAttributeName(int line, int column, String name)`; **bool**
`document.isAttributeName(Cursor cursor, String name)`;

Returns `true`, if the attribute name at a certain cursor position matches the given `name`, otherwise `false`.

String `document.variable(String key)`;

Returns the value of the requested document variable `key`. If the document variable does not exist, the return value is an empty string.

void `document.setVariable(String key, String value)`;

Set the value of the requested document variable `key`.

See also: [Kate document variables](#)

Since: KDE 4.8

int `document.firstVirtualColumn(int line)`;

Returns the virtual column of the first non-whitespace character in the given line or `-1`, if the line is empty or contains only whitespace characters.

int `document.lastVirtualColumn(int line)`;

Returns the virtual column of the last non-whitespace character in the given line or `-1`, if the line is empty or contains only whitespace characters.

int `document.toVirtualColumn(int line, int column)`;
int `document.toVirtualColumn(Cursor cursor)`; **Cursor**
`document.toVirtualCursor(Cursor cursor)`;

Converts the given 'real' cursor position to a virtual cursor position, either returning an `int` or a `Cursor` object.

int `document.fromVirtualColumn(int line, int virtualColumn)`;
int `document.fromVirtualColumn(Cursor virtualCursor)`; **Cursor**
`document.fromVirtualCursor(Cursor virtualCursor)`;

Converts the given virtual cursor position to a 'real' cursor position, either returning an `int` or a `Cursor` object.

```
Cursor document.anchor(int line, int column, Char character); Cursor  
document.anchor(Cursor cursor, Char character);
```

Searches backward for the given character starting from the given cursor. As an example, if '(' is passed as character, this function will return the position of the opening '('. This reference counting, i.e. other '(...)' are ignored.

```
Cursor document.rfind(int line, int column, String text, int attribute  
= -1); Cursor document.rfind(Cursor cursor, String text, int attribute =  
-1);
```

Find searching backwards the given text with the appropriate *attribute*. The argument *attribute* is ignored if it is set to -1. The returned cursor is invalid, if the text could not be found.

```
int document.defStyleNum(int line, int column); int  
document.defStyleNum(Cursor cursor);
```

Returns the default style used at the given cursor position.

```
bool document.isCode(int line, int column); bool document.isCode(Cursor  
cursor);
```

Returns true, if the attribute at the given cursor position is not equal to all of the following styles: dsComment, dsString, dsRegionMarker, dsChar, dsOthers.

```
bool document.isComment(int line, int column); bool  
document.isComment(Cursor cursor);
```

Returns true, if the attribute of the character at the cursor position is dsComment, otherwise false.

```
bool document.isString(int line, int column); bool document.isString(C-  
ursor cursor);
```

Returns true, if the attribute of the character at the cursor position is dsString, otherwise false.

```
bool document.isRegionMarker(int line, int column); bool  
document.isRegionMarker(Cursor cursor);
```

Returns true, if the attribute of the character at the cursor position is dsRegionMarker, otherwise false.

```
bool document.isChar(int line, int column); bool document.isChar(Cursor  
cursor);
```

Returns true, if the attribute of the character at the cursor position is dsChar, otherwise false.

```
bool document.isOthers(int line, int column); bool document.isOthers(C-  
ursor cursor);
```

Returns true, if the attribute of the character at the cursor position is dsOthers, otherwise false.

```
void document.indent(Range range, int change);
```

Indents all lines in *range* by *change* tabs or *change* times *tabSize* spaces depending on the users preferences. The *change* parameter can be negative.

6.4.3.5 The Editor API

In addition to the document and view API, there is a general editor API that provides functions for general editor scripting functionality.

```
String editor.clipboardText();
```

Returns the text that currently is in the global clipboard.

Since: KDE Frameworks 5.50

String editor.clipboardHistory();

The editor holds a clipboard history that contains up to 10 clipboard entries. This function returns all entries that currently are in the clipboard history.

Since: KDE Frameworks 5.50

void editor.setClipboardText(String text);

Set the contents of the clipboard to *text*. The *text* will be added to the clipboard history.

Since: KDE Frameworks 5.50

Chapter 7

Configure KatePart

Selecting **Settings** → **Configure Application...** from the menu brings up the **Configure** dialog box. This dialog can be used to alter a number of different settings. The settings available for change vary according to which category the user chooses from a vertical list on the left side of the dialog. By means of three buttons along the bottom of the box the user can control the process.

You may invoke the **Help** system, accept the current settings and close the dialog by means of the **OK** button, or **Cancel** the process. The categories **Appearance**, **Fonts & Colors**, **Editing**, **Open/Save** and **Extensions** are detailed below.

7.1 The Editor Component Configuration

This group contains all pages related to the editor component of KatePart. Most of the settings here are defaults, they can be overridden by [defining a filetype](#), by [Document Variables](#) or by changing them per document during an editing session.

7.1.1 Appearance

7.1.1.1 General

Editor font

Here you can choose the font of the editor text. You can choose from any font available on your system, and set a default size. A sample text is displayed at the bottom of the dialog, so you can see the effect of your choices.

For more information about selecting a font, see the [Choosing Fonts section of the KDE Fundamentals documentation](#).

Show whitespace indicators

Never

The editor will never display dots to indicate the presence of whitespace.

At the end of a line

The editor will display dots to indicate the presence of extra whitespace at the end of lines.

Always

The editor will always display dots to indicate the presence of whitespace.

Whitespace indicator size

Use the slider to change the size of the visible indicator marker.

Show tab indicators

When checked the editor will display a » symbol to indicate the presence of a tab in the text.

Show focus frame around editor

When checked the editor shows focus frame around the main source text control.

Bracket matching

Highlight range between selected brackets

If this is enabled, the range between the selected matching brackets will be highlighted.

Show preview of matching open bracket

When enabled, the editor will show a tooltip of the matching open bracket.

Flash matching bracket when cursor moves to other bracket in pair

If enabled, moving on the brackets ({, [,], },(or)) will quickly flash the matching bracket.

Show indentation lines

If this is checked, the editor will display vertical lines to help identifying indent lines.

Counts

Show word count

Displays the number of words and characters in the document and in the current selection in the status bar. This option is also available in the status bar context menu.

Show line count

Displays the number of total lines in the document in the status bar. This option is also available in the status bar context menu.

Fold first line

If enabled, the first line is folded, if possible. This is useful, if the file starts with a comment, such as a copyright

Dynamic Word Wrap

If this option is checked, the text lines will be wrapped at the view border on the screen.

Wrap dynamically at static word wrap marker

When checked, editor wraps lines dynamically at the [static word wrap position](#).

Disregard word boundaries for dynamic wrapping

When checked, the editor does not take into account word boundaries when wrapping text lines.

Dynamic word wrap indicators

Choose when the Dynamic word wrap indicators should be displayed, either **Off**, **Follow Line Numbers** or **Always on**.

Indent wrapped lines

Additionally, this allows you to set a maximum width of the screen, as a percentage, after which dynamically wrapped lines will no longer be vertically aligned. For example, at 50%, lines whose indentation levels are deeper than 50% of the width of the screen will not have vertical alignment applied to subsequent wrapped lines.

Line Height Multiplier

This value will be multiplied by the font's default line height. A value of 1.0 means that the default height will be used.

7.1.1.2 Borders

Code block folding

Show arrows to collapse code blocks

If this option is checked, the current view will display marks for code folding, if code folding is available.

Show preview of folded blocks on hover

If checked, hovering over a folded region shows a preview of the folded text in a popup.

Folding arrows visibility

Switch the folding arrows between **Show on Hover** and **Show Always**.

Left side

Show marks

If this is checked, you will see an icon border on the left hand side. The icon border shows bookmark signs for instance.

Show line numbers

If this is checked, you will see line numbers on the left hand side.

Highlight changed and unsaved lines

If this is checked, line modification markers will be visible. For more information, see Section 3.9.

Scrollbars

Show marks

If this option is checked the current view will show marks on the vertical scrollbar. These marks will for instance show bookmarks.

Show preview when hovering over scrollbar

If this option is checked, and you hover the scrollbar with the mouse cursor a small text preview with several lines of the current document around the cursor position will be displayed. This allows you to quickly switch to another part of the document.

Minimap

Show minimap

If this option is checked, every new view will show a minimap of the document on the vertical scrollbar.

For more information on the scrollbar minimap, see Section 3.10.

Minimap Width

Adjusts the width of the scrollbar mini-map, defined in pixels.

Scrollbars visibility

Switch the scrollbar on, off or show the scrollbar only when needed. Click with the left mouse button on the blue rectangle to display the line number range of the document displayed on the screen. Keep the left mouse button pressed outside the blue rectangle to automatically scroll through the document.

Sort bookmarks menu

By date created

Each new bookmark will be added to the bottom, independently from where it is placed in the document.

By line number

The bookmarks will be ordered by the line numbers they are placed at.

7.1.2 Color Themes

This section of the dialog lets you configure all colors in any color theme you have, and create new themes, delete existing ones or just **Follow System Color Scheme**. Each scheme has settings for colors and normal and highlighted text styles.

KatePart will preselect the currently active theme for you, if you want to work on a different theme start by selecting that from the **Select theme** combobox. With the **Copy** and **Delete** buttons you can create a new theme (copying an existing one) or delete existing ones.

This is described in detail in Section [6.3.5](#).

7.1.3 Editing

7.1.3.1 General

Word wrap

Word wrap is a feature that causes the editor to automatically start a new line of text and move (wrap) the cursor to the beginning of that new line. KatePart will automatically start a new line of text when the current line reaches the length specified by the [Wrap words at:](#) option.

Wrap words at a fixed column

Turns static word wrap on or off.

Draw vertical line at the word wrap column

If this option is checked, a vertical line will be drawn at the word wrap column as defined in the **Settings** → **Configure Editor...** in the Editing tab. Please note that the word wrap marker is only drawn if you use a fixed pitch font.

Wrap words at:

If the [Wrap words at a fixed column](#) option is selected this entry determines the length (in characters) at which the editor will automatically start a new line.

Default input mode

The selected input mode will be enabled when opening a new view. You can still toggle the vi input mode on/off for a particular view in the **Edit** menu.

Brackets

If the **Automatically close brackets when opening bracket is typed** option is selected when the user types a left bracket ([, (, or {) KatePart automatically enters the right bracket (],), or }) to the right of the cursor.

Enclosing characters

It is possible to select the enclosing characters using the corresponding drop-down list. When text is selected, typing one of these characters wraps the selected text.

Copy and paste

Move selected text when dragged

This option enables drag-and-drop of the selected text inside the editor window.

Copy/cut the current line if invoked without any text selected

If this option is enabled and the text selection is empty, copy and cut action are performed for the line of text at the actual cursor position.

Don't move the text cursor when pasting by mouse

If this option is enabled and you paste some text in the editor window with the middle mouse button clicking, KatePart will not move the text cursor into the clicked position.

7.1.3.2 Text Navigation

Text Cursor Movement

Smart home and smart end

When selected, pressing the home key will cause the cursor to skip white space and go to the start of a line's text.

PageUp/PageDown moves cursor

This option changes the behavior of the cursor when the user presses the **PgUp** or **PgDn** key. If unselected the text cursor will maintain its relative position within the visible text in KatePart as new text becomes visible as a result of the operation. So if the cursor is in the middle of the visible text when the operation occurs it will remain there (except when one reaches the beginning or end.) With this option selected, the first key press will cause the cursor to move to either the top or bottom of the visible text as a new page of text is displayed.

Enable camel case cursor movement

This option changes the behavior of the cursor when the user presses the **Ctrl-Left arrow** or **Ctrl-Right arrow** shortcut. If unselected the text cursor jumps over the full words. With this option selected, the cursor jumps break at camel case humps.

Autocenter cursor:

Sets the number of lines to maintain visible above and below the cursor when possible.

Text Selection Mode

Normal

Selections will be overwritten by typed text and will be lost on cursor movement.

Persistent

Selections will stay even after cursor movement and typing.

Allow scrolling past the end of the document

This option lets you scroll past the end of the document. This can be used to vertically centre the bottom of the document, or put it on top of the current view.

Backspace key removes character's base with its diacritics

When selected, composed characters are removed with their diacritics instead of only removing the base character. This is useful for Indic locales.

Multicursor modifier

This option lets you set the modifier that will be used to create multiple cursors with left mouse button click. You need to press the modifiers and click left mouse button to create a cursor at the desired location. See [Creating multiple cursors](#) to discover other ways to create multiple cursors.

7.1.3.3 Indentation

Default indentation mode:

Select the automatic indentation mode you want to use as default. It is strongly recommended to use **None** or **Normal** here, and use filetype configurations to set other indentation modes for text formats like C/C++ code or XML.

Indent using

Tabulators

When this is enabled the editor will insert tabulator characters when you press the **Tab** key or use [automatic indentation](#).

Spaces

When this is enabled the editor will insert a calculated number of spaces according to the position in the text and the `tab-width` setting when you press the **Tab** key or use [automatic indentation](#).

Tabulators and Spaces

When this is enabled, the editor will insert spaces as describe above when indenting or pressing **Tab** at the beginning of a line, but insert tabulators when the **Tab** key is pressed in the middle or end of a line.

Tab width:

This configures the number of spaces that are displayed in place of a tabulator character.

Indentation width:

The indentation width is the number of spaces which is used to indent a line. If configured to indent using tabulators, a tabulator character is inserted if the indentation is divisible by the tab width.

Indentation Properties

Keep extra spaces

If this option is disabled, changing the indentation level aligns a line to a multiple of the width specified in **Indentation width**.

Adjust indentation of text pasted from the clipboard

If this option is selected, text pasted from the clipboard is indented. Triggering the **Undo** action removes the indentation.

Indentation Actions

Backspace key in leading blank space unindents

If this option is selected, the **Backspace** key decreases the indentation level if the cursor is located in the leading blank space of a line.

Tab key action (if no selection exists)

If you want **Tab** to align the current line in the current code block like in Emacs, make **Tab** a shortcut to the action **Format Indentation**.

Always advance to the next tab position

If this option is selected, the **Tab** key always inserts white space so that the next tab position is reached. If the option **Insert spaces instead of tabulators** on the **General** tab in the **Editing** page is enabled, spaces are inserted; otherwise, a single tabulator is inserted.

Always increase indentation level

If this option is selected, the **Tab** key always indents the current line by the number of character positions specified in **Indentation width**.

Increase indentation level if in leading blank space

If this option is selected, the **Tab** key either indents the current line or advances to the next tab position. If the insertion point is at or before the first non-space character in the line, or if there is a selection, the current line is indented by the number of character positions specified in **Indentation width**. If the insertion point is located after the first non-space character in the line and there is no selection, white space is inserted so that the next tab position is reached: if the option **Insert spaces instead of tabulators** on the **General** tab in the **Editing** page is enabled, spaces are inserted; otherwise, a single tabulator is inserted.

7.1.3.4 Auto Completion

General

Enable auto completion

If enabled, a word completion box automatically pops up during typing showing a list of text entries to complete the current text under the cursor.

Auto select first completion entry

If enabled, the first auto completion item is always preselected so you can insert it with **Enter**. If you do not want such behavior, e.g. if you want pressing **Enter** to only insert a newline, then disable this item.

Minimal word length to complete

While typing text, the word completion searches for words in the document starting with the already typed text. This option configures the minimal amount of characters that are needed to make the word completion active and pop up a completion box.

Remove tail on complete

Remove the tail of a previous word when the completion item is chosen from a list.

Keyword completion

If enabled, the built-in autocompletion uses the keywords defined by the syntax highlighting.

7.1.3.5 Spellcheck

These configuration options are described in the documentation for the System Settings module [Spell Checker](#).

7.1.3.6 Vi Input Mode

General

Let Vi commands override Kate shortcuts

When selected, Vi commands will override KatePart's built-in commands. For example: **Ctrl+R** will redo, and override the standard action (showing the search and replace dialog).

Display relative line numbers

if this is enabled, the current line always refers to line 0. Lines above and below increase the line number relatively.

Key Mapping

Key mapping is used to change the meaning of typed keys. This allows you to move commands to other keys or make special keypresses for doing a series of commands.

Example:

F2 -> **I-- Esc**

This will prepend **I--** to a line when pressing **F2**.

7.1.4 Open/Save

7.1.4.1 General

File Format

Encoding

This defines the standard encoding to use to open/save files, if not changed in the open/save dialog or by using a command line option.

Encoding Detection

Select an item from the drop down box, either to disable autodetection or use **Universal** to enable autodetection for all encodings. But as this may probably only detect utf-8/utf-16, selecting a region will use custom heuristics for better results. If neither the encoding chosen as standard above, nor the encoding specified in the open/save dialog, nor the encoding specified on command line match the content of the file, this detection will be run.

Fallback Encoding

This defines the fallback encoding to try for opening files if neither the encoding chosen as standard above, nor the encoding specified in the open/ save dialog, nor the encoding specified on command line match the content of the file. Before this is used, an attempt will be made to determine the encoding to use by looking for a byte order mark at start of file: if one is found, the right unicode encoding will be chosen; otherwise encoding detection will run, if both fail fallback encoding will be tried.

End of line

Choose your preferred end of line mode for your active document. You have the choice between UNIX[®], DOS/Windows[®] or Macintosh.

Automatic end of line detection

Check this if you want the editor to autodetect the end of line type. The first found end of line type will be used for the whole file.

Enable byte order mark (BOM)

The byte order mark is a special sequence at the beginning of unicode encoded documents. It helps editors to open text documents with the correct unicode encoding. For more information see [Byte Order Mark](#).

Line Length Limit

Unfortunately, due to deficiencies in Qt[™], KatePart experiences poor performance when working with extremely long lines. For that reason, KatePart will automatically wrap lines when they are longer than the number of characters specified here. To disable this, set this to 0.

Automatic Cleanups on Save

Remove trailing spaces

The editor will automatically eliminate extra spaces at the ends of lines of text while saving the file. You can select **Never** to disable this functionality, **On Modified Lines** to do so only on lines that you have modified since you last saved the document, or **In Entire Document** to remove them unconditionally from the entire document.

Append newline at end of file on save

The editor will automatically append a newline to the end of the file if one is not already present upon saving the file.

Enable Auto Save (local files only)

Check this if you want the editor to autosave documents while you are working on them.

Auto save document when focus leaves the editor

The editor will automatically save documents when you switch to something outside the editor, e.g., the terminal panel in Kate.

Auto save interval

You can determine the autosave interval in seconds here. If the interval is 0, the document will not be autosaved after intervals.

7.1.4.2 Advanced

Write a backup file on save for

Backing up on save will cause KatePart to copy the disk file (the previously saved version of the file) to `<prefix><filename><suffix>` before saving the new changes. A backup file can help you recover work if something goes wrong while saving or if you later want to recover the previous version of the file. The suffix defaults to `~` and prefix is empty by default.

Local files

Check this if you want backups of local files when saving.

Remote files

Check this if you want backups of remote files when saving.

Prefix for backup files

Enter the prefix to prepend to the backup file names.

Suffix for backup files

Enter the suffix to add to the backup file names.

Swap file mode

KatePart is able to recover (most) unsaved work in the case of a crash or power failure. A swap file (`<filename>.kate-swp`) is created when a document is edited. If the user doesn't save the changes and KatePart crashes, the swap file remains on the disk. When opening a file, KatePart checks if there is a swap file for the document and if it is, it asks the user whether he wants to recover the lost data or not. The user has the possibility to view the differences between the original file and the recovered one, too. The swap file is deleted after every save and on normal exit.

KatePart syncs the swap files on the disk every 15 seconds, but only if they have changed since the last sync. The user can disable the swap files syncing if he wants, by selecting **Disable**, but this can lead to more data loss.

When the swap file is enabled, it is possible to switch between two modes, namely **Enabled, Store in Default Directory** and **Enabled, Store in Custom Directory**.

Store swap files in

By default, the swap files are saved in the same folder as the file. When **Enabled, Store in Custom Directory** is chosen for the swap file mode, swap files are created in the specified folder. This is useful for network file systems to avoid unnecessary network traffic.

Save swap files every

KatePart syncs the swap files on the disk every 15 seconds, but only if they have changed since the last sync. You can change the sync interval as you like.

7.1.4.3 Modes & Filetypes

This page allows you to override the default configuration for documents of specified MIME types. When the editor loads a document, it will try if it matches the file masks or MIME types for one of the defined filetypes, and if so apply the variables defined. If more filetypes match, the one with the highest priority will be used.

Filetype:

The filetype with the highest priority is the one displayed in the first drop down box. If more filetypes were found, they are also listed.

New

This is used to create a new filetype. After you click on this button, the fields below get empty and you can fill the properties you want for the new filetype.

Delete

To remove an existing filetype, select it from the drop down box and press the Delete button.

Properties of *current filetype*

The filetype with the highest priority is the one displayed in the first drop down box. If more filetypes were found, they are also listed.

Name:

The name of the filetype will be the text of the corresponding menu item. This name is displayed in the **Tools** → **Filetypes** menu.

Section:

The section name is used to organize the file types in menus. This is also used in the **Tools** → **Filetypes** menu.

Variables:

This string allows you to configure KatePart's settings for the files selected by this MIME type using KatePart variables. You can set almost any configuration option, such as highlight, indent-mode, etc.

Press **Edit** to see a list of all available variables and their descriptions. Select the checkbox on the left to enable a particular variable and then set the value of the variable on the right. Some variables provide a drop-down box to select possible values from while others require you to enter a valid value manually.

For complete information on these variables, see [Configuring with Document Variables](#).

Highlighting:

If you create a new file type, this drop down box allows you to select a filetype for highlighting.

Indentation Mode:

The drop down box specifies the indentation mode for new documents.

File extensions:

The wildcards mask allows you to select files by filename. A typical mask uses an asterisk and the file extension, for example *.txt; *.text. The string is a semicolon-separated list of masks.

MIME types:

Displays a wizard that helps you easily select MIME types.

Priority:

Sets a priority for this file type. If more than one file type selects the same file, the one with the highest priority will be used.

7.2 Configuring With Document Variables

KatePart variables is KatePart's implementation of document variables, similar to Emacs and vi modelines. In katepart, the lines have the following format: **kate: VARIABLENAME VALUE;** [**VARIABLENAME VALUE;** . . .] The lines can of course be in a comment, if the file is in a format with comments. Variable names are single words (no whitespace), and anything up to the next semicolon is the value. The semicolon is required.

Here is an example variable line, forcing indentation settings for a C++, Java™ or JavaScript file:

```
// kate: replace-tabs on; indent-width 4; indent-mode cstyle;
```

NOTE

Only the first and last 10 lines are searched for variable lines.

Additionally, document variables can be placed in a file called `.kateconfig` in any directory, and the configured settings will be applied as if the modelines were entered on every file in the directory and its subdirectories. Document variables in `.kateconfig` use the same syntax as in modelines, but with [extended options](#).

There are variables to support almost all configurations in KatePart, and additionally plugins can use variables, in which case it should be documented in the plugin's documentation.

KatePart has support for reading configurations from `.editorconfig` files, when the `editorconfig` library is installed. KatePart automatically searches for a `.editorconfig` whenever you open a file. It gives priority to `.kateconfig` files, though.

7.2.1 How KatePart uses Variables

When reading configuration, katepart looks in the following places (in that order):

- The global configuration.
- Optional session data.
- The "Filetype" configuration.
- Document variables in `.kateconfig`.
- Document variables in the document itself.
- Settings made during editing from menu or command line.

As you can see, document variables are only overridden by changes made at runtime. Whenever a document is saved, the document variables are reread, and will overwrite changes made using menu items or the command line.

Any variable not listed below is stored in the document and can be queried by other objects such as plugins, which can use them for their own purpose. For example, the variable `indent mode` uses document variables for its configuration.

The variables listed here documents KatePart version 5.38. More variables may be added in the future. There are 3 possible types of values for variables, with the following valid expressions:

- **BOOL** - on | off | true | false | 1 | 0
- **INTEGER** - any integer number
- **STRING** - anything else

7.2.2 Available Variables

auto-brackets [BOOL]

Enable automatic insertion of brackets.

auto-center-lines [INT]

Set the number of autocenter lines.

background-color [STRING]

Set the document background color. The value must be something that can be evaluated to a valid color, for example **#ff0000**.

backspace-indent [BOOL]

Enable or disable unindenting when **Backspace** is pressed.

block-selection [BOOL]

Turn **block selection** on or off.

bom | **byte-order-mark** | **byte-order-marker** [BOOL]

Enable/disable the byte order mark (BOM) when saving files in Unicode format (utf8, utf16, utf32).

Since: Kate 3.4 (KDE 4.4)

bracket-highlight-color [STRING]

Set the color for the bracket highlight. The value must be something that can be evaluated to a valid color, for example **#ff0000**.

current-line-color [STRING]

Set the color for the current line. The value must be something that can be evaluated to a valid color, for example **#ff0000**.

default-dictionary [STRING]

Sets the default dictionary used for spellchecking.

Since: Kate 3.4 (KDE 4.4)

dynamic-word-wrap [BOOL]

Turns **dynamic word wrap** on or off.

eol | **end-of-line** [STRING]

Set the end of line mode. Valid settings are **unix**, **mac** and **dos**.

folding-markers [BOOL]

Set the display of **folding markers** on or off.

folding-preview [BOOL]

Enable folding preview in the editor border.

font-size [INT]

Set the point size of the document font.

font [STRING]

Set the font of the document. The value should be a valid font name, for example **courier**.

hl | **syntax** [STRING]

Set the syntax highlighting. Valid strings are all the names available in the menus. For instance, for C++ simply write **C++**.

icon-bar-color [STRING]

Set the icon bar color. The value must be something that can be evaluated to a valid color, for example `#ff0000`.

icon-border [BOOL]

Set the display of the icon border on or off.

indent-mode [STRING]

Set the auto-indentation mode. The options `none`, `normal`, `cstyle`, `haskell`, `lilypond`, `lisp`, `python`, `ruby` and `xml` are recognized. See the section Section 3.8 for details.

indent-pasted-text [BOOL]

Enable/disable adjusting indentation of text pasted from the clipboard.

Since: Kate 3.11 (KDE 4.11)

indent-width [INT]

Set the indentation width.

keep-extra-spaces [BOOL]

Set whether to keep extra spaces when calculating indentation width.

line-numbers [BOOL]

Set the display of line numbers on or off.

newline-at-eof [BOOL]

Add an empty line at the end of the file (EOF) when saving the document.

Since: Kate 3.9 (KDE 4.9)

overwrite-mode [BOOL]

Set overwrite mode on or off.

persistent-selection [BOOL]

Set [persistent selection](#) on or off.

replace-tabs-save [BOOL]

Set tab to space conversion on save on or off.

replace-tabs [BOOL]

Set dynamic tab to space conversion on or off.

remove-trailing-spaces [STRING]

Removes trailing spaces when saving the document. Valid options are:

- `none`, `-` or `0`: never remove trailing spaces.
- `modified`, `mod`, `+` or `1`: remove trailing spaces only in modified lines. The modified lines are marked by the line modification system.
- `all`, `*` or `2`: remove trailing spaces in the entire document.

scrollbar-minimap [BOOL]

Show scrollbar minimap.

scrollbar-preview [BOOL]

Show scrollbar preview.

scheme [STRING]

Set the color scheme. The string must be the name of a color scheme that exists in your configuration to have any effect.

selection-color [STRING]

Set the selection color. The value must be something that can be evaluated to a valid color, for example `#ff0000`.

show-tabs [BOOL]

Set the visual tab character on or off.

smart-home [BOOL]

Set [smart home navigation](#) on or off.

tab-indent [BOOL]

Set **Tab** key indentation on or off.

tab-width [INT]

Set the tab character display width.

undo-steps [INT]

Set the number of undo steps to remember.

Note: Deprecated since Kate 3 in KDE4. This variable is ignored. The maximal count of undo steps is unlimited.

word-wrap-column [INT]

Set the [static word wrap](#) width.

word-wrap-marker-color [STRING]

Set the word wrap marker color. The value must be something that can be evaluated to a valid color, for example `#ff0000`.

word-wrap [BOOL]

Set static word wrapping on or off.

7.2.3 Extended Options in `.kateconfig` files

KatePart always search for a `.kateconfig` file for local files (not remote files). In addition, it is possible to set options based on wildcards (file extensions) as follows:

```
kate: tab-width 4; indent-width 4; replace-tabs on;
kate-wildcard(*.xml): indent-width 2;
kate-wildcard(Makefile): replace-tabs off;
```

In this example, all files use a tab-width of 4 spaces, an indent-width of 4 spaces, and tabs are replaced expanded to spaces. However, for all `*.xml` files, the indent width is set to 2 spaces. And Makefiles use tabs, i.e. tabs are not replaced with spaces.

Wildcards are semicolon separated, i.e. you can also specify multiple file extensions as follows:

```
kate-wildcard(*.json;*.xml): indent-width 2;
```

Further, you can also use the MIME type to match certain files, e.g. to indent all C++ source files with 4 spaces, you can write:

```
kate-mimetype(text/x-c++src): indent-width 4;
```

NOTE

Next to the support in `.kateconfig` files, wildcard and MIME type dependent document variables are also supported in the files itself as comments.

Chapter 8

Credits and License

KatePart and KWrite Copyright 2001-2022 by the Kate team.

Based on the original KWrite, which was Copyright 2000 by Jochen Wilhelmy digisnap@cs.tu-berlin.de

Contributions:

- Christoph Cullmann cullmann@kde.org
- Michael Bartl michael.bartl1@chello.at
- Phlip phlip_cpp@my-deja.com
- Anders Lund anders@alweb.dk
- Matt Newell newellm@proaxis.com
- Joseph Wenninger kde@jowenn.at
- Jochen Wilhelmy digisnap@cs.tu-berlin.de
- Michael Koch koch@kde.org
- Christian Gebauer gebauer@kde.org
- Simon Hausmann hausmann@kde.org
- Glen Parker glenebob@nwlink.com
- Scott Manson sdmanson@altel.net
- John Firebaugh jfirebaugh@kde.org
- Nivaldo González nibgonz@gmail.com

The KatePart documentation is based on the original KWrite documentation, modified to be relevant to all KatePart consumers.

The original KWrite documentation was written by Thad McGinnis ctmcginnis@compuserve.com, with lots of modifications from Christian Tibirna tibirna@kde.org. Converted to docbook/proofreading by Lauri Watts lauri@kde.org and updated by Anne-Marie Mahfouf annma@kde.org and Anders Lund anders@alweb.dk

The current KatePart documentation is maintained by T.C. Hollingsworth thollingsworth@gmail.com. Please send comments or suggestions to the KatePart development mailing list at kwrite-devel@kde.org or file a bug in the [KDE Bugtracking System](#).

This documentation is licensed under the terms of the [GNU Free Documentation License](#).

This program is licensed under the terms of the [GNU General Public License](#).

Chapter 9

The VI Input Mode

Erlend Hamberg

9.1 VI Input Mode

The goal of the VI mode is not to be a complete replacement for Vim and support all Vim's features. Its aim is to make the 'Vim way' of text editing - and the Vim habits learned - available for programs using the KatePart text editor as their internal editor.

The VI mode aims to integrate nicely with the program and deviate from Vim's behavior where it makes sense. For example, `:w` will open a save dialog in KatePart's VI mode.

To enable the VI Input Mode for all new views, go to **Settings** → **Configure KatePart...+Editing** → **VI Input Mode**. On this tab you can set options for the VI Input Mode and define and edit the key mapping in this mode. VI Input Mode can also be toggled with the **VI Input Mode** setting in the **Edit** menu. (The default shortcut key is **Meta+Ctrl+V** - where **Meta** usually is the **Windows** key).

NOTE

Many Vi Mode keyboard commands are case-sensitive, unlike most KDE keyboard shortcuts. That means that **y** and **Y** are different commands. To enter the **y** (yank) command, make sure **Caps Lock** is disabled and press **Y**. To enter the **Y** (yank to end of line) command, **Shift+Y**.

This does not apply to commands that use the **Ctrl** key, which may be entered regardless of **Caps Lock** mode and without pressing **Shift**. However, some commands require the use of a **Ctrl**-key combination followed by another key which is case sensitive. For instance, to input '**Ctrl+W, h**' (switch to split view right) make sure **Caps Lock** is disabled, press **Ctrl+W**, release, and then press **H**.

9.1.1 Incompatibilities with Vim

There are only a few features of KatePart's VI mode which are incompatible with Vim (not counting things missing). They are listed below together with the respective reasons.

- KatePart: **U** and **Ctrl+R** is redo.

Vim: **Ctrl+R** is normal redo, **U** is used to undo all latest changes on one line.

The reason for having **U** act as redo in KatePart's VI mode is that the shortcut **Ctrl+R** by default is taken by KatePart's replace function (search and replace). By default, the VI mode won't override KatePart's shortcuts (this can be configured in **Settings** → **Configure**

KatePart...+Editing → **Vi Input Mode**), therefore a redo-action needs to be available as a ‘regular’ key press, too. Besides, the behavior of the **U** command in Vim does not map well to KatePart’s internal undo system, so it would be non-trivial to support anyway.

- KatePart: **print** shows the **Print** dialog.
Vim: **print** prints the lines of the given range like its grandfather **ed**.
Commands like **:print** are available not only in the VI mode but for users using ‘regular’ KatePart, too - therefore the **:print** command opens the print dialog - following the principle of least surprise instead of mimicking Vim’s behavior.
- KatePart: **Y** yanks to end of line.
Vim: **Y** yanks whole line, just like **yy**.
VI’s behavior for the **Y** command is in practice a bug. For both change and delete commands, **cc/ dd** will do its action on the current line and **C/D** will work from the cursor column to the end of the line. However, both **yy** and **Y** yanks the current line. In KatePart’s VI Mode **Y** will yank to the end of the line. This is described as ‘more logical’ in the Vim [documentation](#).
- KatePart: **O** and **o** opens [*count*] new lines and puts you in insert mode.
Vim: **O** and **o** opens a new line and inserts text [*count*] times when leaving insert mode.
This is mostly done as a consequence of witnessing many people being confused by this behavior on a vim IRC channel ([#vim on Libera Chat](#)).

9.1.2 Switching Modes

- *Normal Mode* permits you to enter commands to navigate or edit a document, and is the default mode. You can return to it from any other mode by pressing **Esc**.
- *Visual Mode* permits you to highlight text in a document. Most Normal Mode commands are also valid in this mode. You can enter it by pressing **v** to select characters or **V** to select lines.
- *Insert Mode* permits you to edit the document directly. You can enter it by pressing **i** or one of several other commands listed below.
- *Command Mode* invokes KatePart’s command line, permitting you to run many commands available in Vi implementations as well as some specific to KatePart. For more information on these commands, see [Section 5.2](#). To use it, press **:**, enter the command, and press **Enter**.

9.1.3 Integration with Kate features

- Visual Mode is entered automatically when text is selected with the mouse. It is also entered when using functions of Kate that select text, such as Select All (either from the menu or via **Ctrl+A**.)
- Vi marks and [Kate bookmarks](#) are integrated. When a mark is created in Vi Mode, a corresponding Kate bookmark is created and appears in the **Bookmarks** menu. Conversely, when a Kate bookmark is created, a corresponding Vi mark at the 0 column is also created.

9.1.4 Supported normal/visual mode commands

The KatePart Handbook

a	Enter Insert Mode; append after cursor
A	Enter Insert Mode; append after line
i	Enter Insert Mode; insert before cursor
Ins	Enter Insert Mode; insert before cursor
I	Enter Insert Mode; insert before first non-blank char on line
gi	Enter Insert Mode; insert before place, where leaving the last insert mode
v	Enter Visual Mode; select characters
V	Enter Visual Mode; select lines
Ctrl+v	Enter Visual Mode; select blocks
gb	Enter Visual Mode; reselect the last selection
o	Open a new line below current line
O	Open a new line above current line
J	Join lines
c	Change: follow by a motion to delete and enter Insert mode
C	Change to end of line: Delete to end of line and enter Insert Mode
cc	Change line: Delete line and enter Insert Mode
s	Substitute character
S	Substitute line
dd	Delete line
d	Follow by a motion to delete
D	Delete to end of line
x	Delete character to right of cursor
Del	Delete character to right of cursor
X	Delete character to left of cursor
gu	Follow with a motion to make lowercase
guu	Make the current line lowercase
gU	Follow with a motion to make uppercase
gUU	Make the current line uppercase
y	Follow by a motion to 'yank' (copy)
YY	Yank (copy) line
Y	Yank (copy) line
p	Paste after cursor
P	Paste before cursor
]p	Paste after cursor indented
[p	Paste before cursor indented
r	Follow with a character to replace the character after the cursor
R	Enter Replace Mode
:	Enter Command Mode
/	Search
u	Undo
Ctrl+R	Redo
U	Redo
m	Set mark (can be used by motions later)
n	Find next
N	Find previous
>>	Indent line

<<	Unindent line
>	Indent lines
<	Unindent lines
Ctrl+F	Page down
Ctrl+B	Page up
ga	Print the ASCII value of the character
.	Repeat last change
==	commandAlignLine
=	commandAlignLines
~	Change case of current character
Ctrl+S	Split view horizontally
Ctrl+V	Split view vertically
Ctrl+W, w	Cycle to next split window
Ctrl+W, h CtrlW Left	Go to left split window
Ctrl+W, l CtrlW Right	Go to right split window
Ctrl+W, k CtrlW Up	Go to above split window
Ctrl+W, j CtrlW Down	Go to below split window

9.1.5 Supported motions

These may be used to move around a document in Normal or Visual mode, or in conjunction with one of the above commands. They may be preceded by a count, which indicates how many of the appropriate movements to make.

h	Left
Left	Left
Backspace	Left
j	Down
Down	Down
k	Up
Up	Up
l	Right
Right	Right
Space	Right
\$	End of line
End	End of line
0	First character of line (Column 0)
Home	First character of line
^	First non-blank character of line
f	Follow by character to move to right of cursor
F	Follow by character to move to left of cursor
t	Follow by character to move to right of cursor, placing the cursor on character before it

T	Follow by character to move to left of cursor, placing the cursor on character before it
gg	First line
G	Last line
w	Next Word
W	Next word separated by whitespace
b	Previous word
B	Previous word separated by whitespace
e	End of word
E	End of word separated by whitespace
ge	End of previous word
gE	End of previous word separated by whitespace
 	Follow by a column number to move to that column
%	Follow by an item to move to that item
`	Mark
^	First non-whitespace character of the line the mark is on
[[Previous opening bracket
]]	Next opening bracket
[]	Previous closing bracket
]]	Next closing bracket
Ctrl+I	Jump to next location
Ctrl+O	Jump to previous location
H	Go to first line of screen
M	Go to middle line of screen
L	Go to last line of screen
%percentage	Go to specified percentage of the document
gk	Go one line up visually (when using dynamic word wrap)
gj	Go one line down visually (when using dynamic word wrap)
Ctrl+Left	Move one word left
Ctrl+Right	Move one word right

9.1.6 Supported text objects

These may be used to select certain portions of a document.

iw	Inner word: word including whitespace
aw	A word: word excluding whitespace
i"	Previous double-quote (") to next double-quote, including quotation marks
a"	Previous double-quote (") to next double-quote, excluding quotation marks
i'	Previous single-quote (') to next single-quote, including quotation marks
a'	Previous single-quote (') to next single-quote, excluding quotation marks

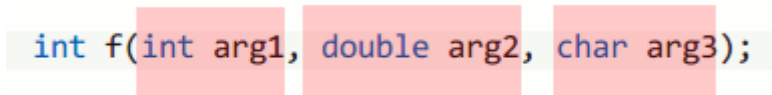
i (Previous opening parenthesis [(] to next closing parenthesis [)], including the parenthesis
a (Previous opening parenthesis [(] to next closing parenthesis [)], excluding the parenthesis
i [Previous opening square bracket ([] to next closing square bracket (]), including the brackets
a [Previous opening square bracket ([] to next closing square bracket (]), excluding the brackets
i {	Previous opening curly bracket ({ } to next closing curly bracket (}), including the brackets
a {	Previous opening curly bracket ({ } to next closing curly bracket (}), excluding the brackets
i <	Previous opening angle bracket (< > to next closing angle bracket (>), including the brackets
a <	Previous opening angle bracket (< > to next closing square bracket (>), excluding the brackets
i `	Previous backtick (`) to next backtick, including the backticks
a `	Previous backtick (`) to next backtick, excluding the backticks

9.1.7 Supported insert mode commands

Ctrl+D	Unindent
Ctrl+T	Indent
Ctrl+E	Insert from below
Ctrl+Y	Delete word
Ctrl+W	Delete word
Ctrl+U	Delete line
Ctrl+J	New line
Ctrl+H	Delete char backward
Ctrl+Home	Move to first character in the document
Ctrl+R n	Insert the contents of register n
Ctrl+O, <i>command</i>	Enter normal mode for one command only
Ctrl+A	Increment currently selected number
Ctrl+X	Decrement currently selected number

9.1.8 The Comma Text Object

This object is missing in Vim. The comma text object makes it easy to modify parameter lists in C-like languages and other comma separated lists. It is basically the area between two commas or between a comma and a bracket. In the line shown in the illustration, the three ranges this text object can span are highlighted.



```
int f(int arg1, double arg2, char arg3);
```

*Comma text object ranges. If the cursor is over e.g. `arg2`, pressing **ci**, ('change inner comma') would delete `double arg2` and place the cursor between the two commas in insert mode. A very convenient way to change a function's parameters.*

9.1.9 Missing Features

As stated earlier, the goal of KatePart's VI Mode is not to support 100% of Vim's features.

Appendix A

Regular Expressions

This Appendix contains a brief but hopefully sufficient and covering introduction to the world of *regular expressions*. It documents regular expressions in the form available within KatePart, which is not compatible with the regular expressions of perl, nor with those of for example **grep**.

A.1 Introduction

Regular Expressions provides us with a way to describe some possible contents of a text string in a way understood by a small piece of software, so that it can investigate if a text matches, and also in the case of advanced applications with the means of saving pieces or the matching text.

An example: Say you want to search a text for paragraphs that starts with either of the names 'Henrik' or 'Pernille' followed by some form of the verb 'say'.

With a normal search, you would start out searching for the first name, 'Henrik' maybe followed by 'sa' like this: **Henrik sa**, and while looking for matches, you would have to discard those not being the beginning of a paragraph, as well as those in which the word starting with the letters 'sa' was not either 'says', 'said' or so. And then of course repeat all of that with the next name...

With Regular Expressions, that task could be accomplished with a single search, and with a larger degree of preciseness.

To achieve this, Regular Expressions defines rules for expressing in details a generalization of a string to match. Our example, which we might literally express like this: 'A line starting with either 'Henrik' or 'Pernille' (possibly following up to 4 blanks or tab characters) followed by a whitespace followed by 'sa' and then either 'ys' or 'id' could be expressed with the following regular expression:

```
^[ \t]{0,4}(Henrik|Pernille) sa(ys|id)
```

The above example demonstrates all four major concepts of modern Regular Expressions, namely:

- Patterns
- Assertions
- Quantifiers
- Back references

The caret (^) starting the expression is an assertion, being true only if the following matching string is at the start of a line.

The strings `[\t]` and `(Henrik|Pernille) sa(ys|id)` are patterns. The first one is a *character class* that matches either a blank or a (horizontal) tab character; the other pattern contains first a subpattern matching either `Henrik` or `Pernille`, then a piece matching the exact string `sa` and finally a subpattern matching either `ys` or `id`.

The string `{0,4}` is a quantifier saying ‘anywhere from 0 up to 4 of the previous’.

Because regular expression software supporting the concept of *back references* saves the entire matching part of the string as well as sub-patterns enclosed in parentheses, given some means of access to those references, we could get our hands on either the whole match (when searching a text document in an editor with a regular expression, that is often marked as selected) or either the name found, or the last part of the verb.

All together, the expression will match where we wanted it to, and only there.

The following sections will describe in details how to construct and use patterns, character classes, assertions, quantifiers and back references, and the final section will give a few useful examples.

A.2 Patterns

Patterns consists of literal strings and character classes. Patterns may contain sub-patterns, which are patterns enclosed in parentheses.

A.2.1 Escaping characters

In patterns as well as in character classes, some characters have a special meaning. To literally match any of those characters, they must be marked or *escaped* to let the regular expression software know that it should interpret such characters in their literal meaning.

This is done by prepending the character with a backslash (\).

The regular expression software will silently ignore escaping a character that does not have any special meaning in the context, so escaping for example a ‘j’ (`\j`) is safe. If you are in doubt whether a character could have a special meaning, you can therefore escape it safely.

Escaping of course includes the backslash character itself, to literally match a such, you would write `\\`.

A.2.2 Character Classes and abbreviations

A *character class* is an expression that matches one of a defined set of characters. In Regular Expressions, character classes are defined by putting the legal characters for the class in square brackets, `[]`, or by using one of the abbreviated classes described below.

Simple character classes just contains one or more literal characters, for example `[abc]` (matching either of the letters ‘a’, ‘b’ or ‘c’) or `[0123456789]` (matching any digit).

Because letters and digits have a logical order, you can abbreviate those by specifying ranges of them: `[a-c]` is equal to `[abc]` and `[0-9]` is equal to `[0123456789]`. Combining these constructs, for example `[a-fynot1-38]` is completely legal (the last one would match, of course, either of ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘y’, ‘n’, ‘o’, ‘t’, ‘1’, ‘2’, ‘3’ or ‘8’).

As capital letters are different characters from their non-capital equivalents, to create a caseless character class matching ‘a’ or ‘b’, in any case, you need to write it `[aAbB]`.

The KatePart Handbook

It is of course possible to create a ‘negative’ class matching as ‘anything but’ To do so put a caret (^) at the beginning of the class:

[**^abc**] will match any character *but* ‘a’, ‘b’ or ‘c’.

In addition to literal characters, some abbreviations are defined, making life still a bit easier:

\a

This matches the ASCII bell character (BEL, 0x07).

\f

This matches the ASCII form feed character (FF, 0x0C).

\n

This matches the ASCII line feed character (LF, 0x0A, Unix newline).

\r

This matches the ASCII carriage return character (CR, 0x0D).

\t

This matches the ASCII horizontal tab character (HT, 0x09).

\v

This matches the ASCII vertical tab character (VT, 0x0B).

\xhhhh

This matches the Unicode character corresponding to the hexadecimal number hhhh (between 0x0000 and 0xFFFF). \0ooo (i.e., \zero ooo) matches the ASCII/Latin-1 character corresponding to the octal number ooo (between 0 and 0377).

. (**dot**)

This matches any character (including newline).

\d

This matches a digit. Equal to [0-9]

\D

This matches a non-digit. Equal to [^0-9] or [^\d]

\s

This matches a whitespace character. Practically equal to [\t\n\r]

\S

This matches a non-whitespace. Practically equal to [^ \t\n\r], and equal to [^\s]

\w

Matches any ‘word character’ - in this case any letter, digit or underscore. Equal to [a-zA-Z0-9_]

\W

Matches any non-word character - anything but letters, numbers or underscore. Equal to [^a-zA-Z0-9_] or [^\w]

The *POSIX notation of classes*, [**:<class name>:**] are also supported. For example, [**:digit:**] is equivalent to **\d**, and [**:space:**] to **\s**. See the full list of POSIX character classes [here](#).

The abbreviated classes can be put inside a custom class, for example to match a word character, a blank or a dot, you could write [**\w \.]**

A.2.2.1 Characters with special meanings inside character classes

The following characters has a special meaning inside the '[' character class construct, and must be escaped to be literally included in a class:

]

Ends the character class. Must be escaped unless it is the very first character in the class (may follow an unescaped caret).

^ (caret)

Denotes a negative class, if it is the first character. Must be escaped to match literally if it is the first character in the class.

- (dash)

Denotes a logical range. Must always be escaped within a character class.

\ (backslash)

The escape character. Must always be escaped.

A.2.3 Alternatives: matching 'one of'

If you want to match one of a set of alternative patterns, you can separate those with | (vertical bar character).

For example to find either 'John' or 'Harry' you would use an expression **John|Harry**.

A.2.4 Sub Patterns

Sub patterns are patterns enclosed in parentheses, and they have several uses in the world of regular expressions.

A.2.4.1 Specifying alternatives

You may use a sub pattern to group a set of alternatives within a larger pattern. The alternatives are separated by the character '|' (vertical bar).

For example to match either of the words 'int', 'float' or 'double', you could use the pattern **int|float|double**. If you only want to find one if it is followed by some whitespace and then some letters, put the alternatives inside a subpattern: **(int|float|double)\s+\w+**.

A.2.4.2 Capturing matching text (back references)

If you want to use a back reference, use a sub pattern (**PATTERN**) to have the desired part of the pattern remembered. To prevent the sub pattern from being remembered, use a non-capturing group (**?:PATTERN**).

For example, if you want to find two occurrences of the same word separated by a comma and possibly some whitespace, you could write **(\w+),\s*\1**. The sub pattern **\w+** would find a chunk of word characters, and the entire expression would match if those were followed by a comma, 0 or more whitespace and then an equal chunk of word characters. (The string **\1** references *the first sub pattern enclosed in parentheses*.)

NOTE

To avoid ambiguities with usage of `\1` with some digits behind it (e.g. `\12` can be 12th subpattern or just the first subpattern with `2`) we use `\{12}` as syntax for multi-digit subpatterns.

Examples:

- `\{12}1` is 'use subpattern 12'
- `\123` is 'use capture 1 then 23 as the normal text'

A.2.4.3 Lookahead Assertions

A lookahead assertion is a sub pattern, starting with either `?=` or `?!`.

For example to match the literal string 'Bill' but only if not followed by ' Gates', you could use this expression: `Bill(?! Gates)`. (This would find 'Bill Clinton' as well as 'Billy the kid', but silently ignore the other matches.)

Sub patterns used for assertions are not captured.

See also [Assertions](#).

A.2.4.4 Lookbehind Assertions

A lookbehind assertion is a sub pattern, starting with either `?<=` or `?<!`.

Lookbehind has the same effect as the lookahead, but works backwards. For example to match the literal string 'fruit' but only if not preceded by 'grape', you could use this expression: `(?<!grape)fruit`.

Sub patterns used for assertions are not captured.

See also [Assertions](#)

A.2.5 Characters with a special meaning inside patterns

The following characters have meaning inside a pattern, and must be escaped if you want to literally match them:

\ (backslash)

The escape character.

^ (caret)

Asserts the beginning of the string.

\$

Asserts the end of string.

() (left and right parentheses)

Denotes sub patterns.

{ } (left and right curly braces)

Denotes numeric quantifiers.

[] (left and right square brackets)

Denotes character classes.

| (vertical bar)

logical OR. Separates alternatives.

+ (plus sign)

Quantifier, 1 or more.

*** (asterisk)**

Quantifier, 0 or more.

? (question mark)

An optional character. Can be interpreted as a quantifier, 0 or 1.

A.3 Quantifiers

Quantifiers allows a regular expression to match a specified number or range of numbers of either a character, character class or sub pattern.

Quantifiers are enclosed in curly brackets (`{` and `}`) and have the general form `{[minimum-occurrences][, [maximum-occurrences]]}`

The usage is best explained by example:

`{1}`

Exactly 1 occurrence

`{0, 1}`

Zero or 1 occurrences

`{, 1}`

The same, with less work;

`{5, 10}`

At least 5 but maximum 10 occurrences.

`{5, }`

At least 5 occurrences, no maximum.

Additionally, there are some abbreviations:

*** (asterisk)**

similar to `{0, }`, find any number of occurrences.

+ (plus sign)

similar to `{1, }`, at least 1 occurrence.

? (question mark)

similar to `{0, 1}`, zero or 1 occurrence.

A.3.1 Greed

When using quantifiers with no maximum, regular expressions defaults to match as much of the searched string as possible, commonly known as *greedy* behavior.

Modern regular expression software provides the means of 'turning off greediness', though in a graphical environment it is up to the interface to provide you with access to this feature. For example a search dialog providing a regular expression search could have a check box labeled 'Minimal matching' as well as it ought to indicate if greediness is the default behavior.

A.3.2 In context examples

Here are a few examples of using quantifiers:

`^\d{4,5}\s`

Matches the digits in '1234 go' and '12345 now', but neither in '567 eleven' nor in '223459 somewhere'.

`\s+`

Matches one or more whitespace characters.

`(bla){1,}`

Matches all of 'blablabla' and the 'bla' in 'blackbird' or 'tabla'.

`/?>`

Matches '/>' in '<closeditem/>' as well as '>' in '<openitem>'.

A.4 Assertions

Assertions allows a regular expression to match only under certain controlled conditions.

An assertion does not need a character to match, it rather investigates the surroundings of a possible match before acknowledging it. For example the *word boundary* assertion does not try to find a non word character opposite a word one at its position, instead it makes sure that there is not a word character. This means that the assertion can match where there is no character, i.e. at the ends of a searched string.

Some assertions actually do have a pattern to match, but the part of the string matching that will not be a part of the result of the match of the full expression.

Regular Expressions as documented here supports the following assertions:

^ (caret: beginning of string)

Matches the beginning of the searched string.

The expression `^Peter` will match at 'Peter' in the string 'Peter, hey!' but not in 'Hey, Peter!'

\$ (end of string)

Matches the end of the searched string.

The expression `you\?$` will match at the last you in the string 'You didn't do that, did you?' but nowhere in 'You didn't do that, right?'

\b (word boundary)

Matches if there is a word character at one side and not a word character at the other.

This is useful to find word ends, for example both ends to find a whole word. The expression `\bin\b` will match at the separate 'in' in the string 'He came in through the window', but not at the 'in' in 'window'.

\B (non word boundary)

Matches wherever '\b' does not.

That means that it will match for example within words: The expression `\Bin\b` will match at in 'window' but not in 'integer' or 'I'm in love'.

(?=PATTERN) (Positive lookahead)

A lookahead assertion looks at the part of the string following a possible match. The positive lookahead will prevent the string from matching if the text following the possible match does not match the *PATTERN* of the assertion, but the text matched by that will not be included in the result.

The expression **handy (=?\w)** will match at 'handy' in 'handyman' but not in 'That came in handy!'

(?!PATTERN) (Negative lookahead)

The negative lookahead prevents a possible match to be acknowledged if the following part of the searched string does match its *PATTERN*.

The expression **const \w+\b(?!\s*&)** will match at 'const char' in the string 'const char* foo' while it can not match 'const QString' in 'const QString& bar' because the '&' matches the negative lookahead assertion pattern.

(?<=PATTERN) (Positive lookbehind)

Lookbehind has the same effect as the lookahead, but works backwards. A lookbehind looks at the part of the string previous a possible match. The positive lookbehind will match a string only if it is preceded by the *PATTERN* of the assertion, but the text matched by that will not be included in the result.

The expression **(?<=cup) cake** will match at 'cake' if it is succeeded by 'cup' (in 'cupcake' but not in 'cheesecake' or in 'cake' alone).

(?<!PATTERN) (Negative lookbehind)

The negative lookbehind prevents a possible match to be acknowledged if the previous part of the searched string does match its *PATTERN*.

The expression **(?<![\w\.]) [0-9]+** will match at '123' in the strings '=123' and '-123' while it can not match '123' in '.123' or 'word123'.

(PATTERN) (Capturing group)

The sub pattern within the parentheses is captured and remembered, so that it can be used in back references. For example, the expression **("+) [^" ;] *\1** matches **text ""** and **text "**.

See the section [Capturing matching text \(back references\)](#) for more information.

(?:PATTERN) (Non-capturing group)

The sub pattern within the parentheses is not captured and is not remembered. It is preferable to always use non-capturing groups if the captures will not be used.

Appendix B

Index

C
comment, [32](#)

R
replace, sed style
 search, sed style, [36](#)

U
uncomment, [32](#)