JIDE Code Editor Developer Guide

Contents

Purpose of This Document

JIDE Code Editor is a special text editor for source code of computer languages. This developer guide is designed for developers who want to learn how to use JIDE Code Editor in their applications.

Features

All features you found in JIDE Code Editor are related to source code. If you want to find an editor that can be used to view or edit source code of a computer language, regardless of it is an existing computer language or a new language you created, this is the right component for you. Here are the highlights of features.

- Syntax coloring for 25 different languages
- Virtual white space
- Reads huge files in very short time and consumes much less memory than the file's size
- Displays new line, space and tab using special graphics
- Handles different line break types depending on the platforms
- Handles tab either as white spaces or as tab.
- Bracket matching
- Code folding
- Optional customizable margin components including pre-built line number margin and code folding margin
- Optional customizable status bar including pre-build status bar items for caret position, caret overwrite/insert state etc.
- Customizable font style and syntax color
- Customizable shortcut keys for all commands used by the editor
- Multiple clipboard copy and paste
- Drag-n-drop selection support. Rectangular selection
- Find and Replace
  - Allows literal, regular expression as well as wildcard. (Ctrl-F and Ctrl-R)
  - Incremental search (F3)
  - Quick search (Alt-F3)

  - Customizable search all and replace all
- Extensive commands to operate on the code editor (refer to Shortcut Editor Dialog to list of all commands)
- Optional inspection area to show line markers for errors, warnings, todo's etc.
- Auto indent
- Column Guides
- Column selection

Code Editor

CodeEditor is the main class for JIDE Code Editor.

Creating a CodeEditor

```
CodeEditor editor = new CodeEditor();
editor.setText("…"); // the text
editor.setFileName("..."); // the file name you would like CodeEditor to read in case you have a huge file
```

Once you create a *CodeEditor*, you can add it to any place you want. *CodeEditor* has its own scroll bars so you don't need to add it to a *JScrollPane* as you did to *JTextArea*.

CaretModel

There are several important modules in *CodeEditor*. *CaretModel* is one of them. *CaretModel* is a model that represents the position of a caret in code editor. In Swing's *JTextComponent* and its subclasses, caret position is just one integer value which is the offset in the *Document*. However in *CaretModel*, there are three different types of positions.

- Offset: The same as in *JTextComponent*. It is the number of characters before the current caret position.
- Model caret position: There are two values in model caret position – line and column. It is the caret x and y coordinate as you see on the screen. We need model position because we want to support virtual spaces after the end of a line. With visual spaces turned on, you can place caret at a place beyond the end of a line. In any positions beyond the end of line, the offset will be the same. Only model caret positions are different.
- View caret position: Same as model caret position, there are also two values in view caret position – line and column. We introduced both view caret position and model caret position because of the need from code folding feature. When there is no code folding, view caret position is the same as model caret position. If there is code folding, model caret position is the visual position before the code is folded and view caret position is the visual position after the code is folded.

There are methods in *CodeEditor* to convert from one of the positions to another. You may need to use those positions. Here are a few rules.

- If you want to modify the Document of the *CodeEditor* at its current caret position, you probably should use offset as that's the actual position in the Document.
- If you want to display the caret on screen, you should always use view caret position. When we handle keyboard action for arrow keys, we use view caret position to do it. For example, right key will cause view caret position's *column* value to increase by 1; up key will make view caret position's *line* value decrease by 1.
- When you want to get to certain line number of in code editor, you use model caret position. The *CaretModelPositionStatusBarItem* displays the logical position. If you double click on it to prompt "go to line" dialog, the line number you input is interrupted as model caret position so that it doesn't change with code folding status.

SelectionModel

SelectionModel is used to keep track of the text selection. *DefaultSelectionModel* is the default implementation of *SelectionModel*.

There are two values stored in a *SelectionModel*. They are the start offset and the end offset.

SelectionModel also supports column selection mode. In some editors, it is called vertical selection or rectangular selection. You can call *setColumnSelectionMode* and set it to true or false. To improve customer experience, JIDE CodeEditor would enter column selection mode automatically if the customer press CTRL key while mouse dragging over the text.

There is also *SelectionListener* support. Anyone can add listener to *SelectionModel* to get notification when selection changes.

In current version, we only support continuous selection. We don't have a plan to support non-contiguous selection so far.

Syntax Coloring

Syntax coloring is one of the basic features provided by *CodeEditor*. In order to implement this feature the fastest way, we leveraged an open source project at http://syntax.jedit.org/ called jEdit Syntax Package. This project is a largely simplified version of jEdit project. jEdit project is GPL-ed which prevents many commercial companies from using it. But jEdit Syntax Package, the project we are using, is MIT license which means it can be used in commercial software.

TokenMarker

The main class is *TokenMarker* which is from jEdit Syntax Package. A token marker splits a line of text into tokens. Each token carries a length field and an identification tag that can be mapped to a color for painting that token. jEdit Syntax Package includes the token markers for the following languages or files:

C, C++, Eiffel, HTML, IDL, Java, JavaScript, Makefile, Patch/diff, Perl, PHP, PLSQL, Java properties, Python, Shell script, SQL, TeX, TSQL, Verilog, VHDL, XML.

We will add more *TokenMarkers* for other commonly used languages in the future based on user feedbacks. You can also create your own *TokenMarker*. The source code of all *TokenMarkers* can be downloaded from the link above. You can study the code to find out how to make one.

TokenMarker and its subclasses use 11 kinds of identification tags to mark a token. They are *NULL*, *COMMENT1*, *COMMENT2*, *LITERAL1*, *LITERAL2*, *LABEL*, *KEYWORD1, KEYWORD2*, *KEYWORD3*, *OPERATOR* and *INVALID*. As you can see, those tags are language neutral which means for any language, you have to use one of the tags above for all tokens. This design simplified the token marker but certainly has its limitation. So it will be one area where our future release will improve.

CodeEditor has *setTokenMarker* method so you can set any *TokenMarker* to it. However we suggest you to use a more systematic way through *LanguageSpec* and *LanguageSpecManager*.

LanguageSpec and LangaugeSpecManager

LanguageSpec defines several properties for a language type, for example, the name of the language, possible suffixes, the token marker, non-word delimiters, the string used for line style comment, the strings used for block style comment etc.

LanguageSpecManager puts all language specs together so that you can look it up by the language name. Once you have the *LangaugeSpec*, you can call *configureCodeEditor* method on it to set all the properties of that language in one shot. If you use *CodeEditorDocumentPane*, you will find *setLanguageName* method on *CodeEditorDocumentComponent* which uses the way mentioned above to configure a code editor easily.

See below for an example code to create a *CodeEditorDocumentComponent*.

```
CodeEditorDocumentComponent createDocumentComponent(String resourceName, String languageName) {
    CodeEditorDocumentComponent documentComponent = new
CodeEditorDocumentComponent(resourceName);
    try {
      documentComponent.open(CodeEditorDemo.class.getClassLoader(), resourceName);
    }
    catch (IOException e) {
      e.printStackTrace();
    }
    documentComponent.setLanguageName(languageName);
    return documentComponent;
  }
```

Status Bar

Adding a status bar

After you create a *CodeEditor*, you can create a *CodeEditorStatusBar* for this *CodeEditor*. Typically, you can add *CodeEditor* to CENTER of a border layout panel and add status bar to the SOUTH of the panel. In this case you will have one status bar per editor. See code below.

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(editor);
panel.add(new CodeEditorStatusBar(editor), BorderLayout.AFTER_LAST_LINE);
```

You can also need to create a common status bar shared by all cell editors. If so, you still create a *CodeEditorStatusBar* first, then you need make sure you call *setEditor* method whenever active editor changes.

```
StatusBar statusBar = new CodeEditorStatusBar();

// when the active editor changes, you call
statusBar.setEditor(activeEditor);
```

You may also already have a status bar for your application and you want to use it instead of creating a new status bar just for code editor. If so, you can use *StatusBarItem* directly. Let's say you want to add a status bar item for caret position to your existing status bar.

```
StatusBar statusBar = new StatusBar(); // an existing status bar
CaretModelPositionStatusBarItem caretPosition = new CaretModelPositionStatusBarItem ();
statusBar.add(caretPosition);

// when the active editor changes, you call
caretPosition.setEditor(activeEditor);
```

Creating your own status bar item for CodeEditor

All status bar items for *CodeEditor* should implement interface *CodeEditorStatusBarItem*. This interface has methods like *set/getEditor()*, *initialize()*, *registerListener(editor)*, and *unregisterListener(editor)*. Abstract class *AbstractCodeEditorStatusBarItem* extends *LabelStatusBarItem* and implements *set/getEditor()* methods and leave the rest three methods for you to implement.

Here is a real example in our source code to create a status bar item that displays caret overwrite/insert status.

```
public class CaretOverwriteStatusBarItem extends AbstractCodeEditorStatusBarItem implements
PropertyChangeListener {

    public final static String INSERT = "Insert";
    public final static String OVERWRITE = "Overwrite";
```

```
    public CaretOverwriteStatusBarItem() {
    }

    public CaretOverwriteStatusBarItem(String name) {
        super(name);
    }

    public void initialize() {
        setHorizontalAlignment(JLabel.CENTER);
        setPreferredWidth(80);
    }

    public void registerListener(CodeEditor editor) {
        if (editor != null) {
            editor.addPropertyChangeListener(this);
            setText(editor.isOverwriteEnabled() ? OVERWRITE : INSERT);
        }
    }

    public void unregisterListener(CodeEditor editor) {
        if (editor != null) {
            editor.removePropertyChangeListener(this);
        }
    }

    public void propertyChange(PropertyChangeEvent evt) {
        if (CodeEditor.PROPERTY_OVERWRITE_ENABLED.equals(evt.getPropertyName())) {
            setText(Boolean.TRUE.equals(evt.getNewValue()) ? OVERWRITE : INSERT);
        }
    }
}
```

Integrate into your application status bar

As long as you use *StatusBar* component provide by JIDE Components product, you can easily add special status bar items to the common status bar used by your application. We currently provide three status bar items that you can use. They are

- CaretModelPositionStatusBarItem: This status bar item shows the caret position. We also have *CaretOffsetStatusBarItem* and *CaretOffsetStatusBarItem* to show the caret view position and offset position respectively. But the caret model position is probably the only position that makes sense to your end user.

- LineBreakStatusBarItem: This status bar item shows the line break type of the editing text.

- CaretOverwriteStatusBarItem: This status bar item shows the caret's override or insert status. It will change with *CodeEditor*'s *setOverwriteEnabled* method to indicate if the *CodeEditor* is in inserting mode or overwriting mode.

- EditableStatusBarItem: This status bar item shows the editable attribute of a *CodeEditor*.

All those status bar items can be added to the *StatusBar* you already have to make the code editor seamlessly integrate with your application.

Magin Area

Magin Area is a special area on the left side of a code editor. Usually the content in margin area will scroll vertically along with the text in code editor. A typical example is line number margin. In fact, you can add many margin components into the Margin Area. Line number is just one of those margins.

A *MarginArea* is created automatically by *CodeEditor*. You can call *editor.getMarginArea()* to get it. Then call *addMarginComponent(Margin magin)* to add a new margin to it. By default, we only add one margin which is *LineNumberMargin* which we will talk about in later. You can call *editor.setLineNumberVisible(true/false)* to show or hide the line number margin.

Create your own margin

All margins must implement an interface called *Margin*. We also provide *AbstractMargin* which implements most of the methods in *Margin*.

Margin should paint itself completely instead using any child components. You don't want to add child components to margin mainly because the margin will scroll along with the text in code editor and is usually very tall. You don't want to deal with any child components or layout manager for this particular component. Taking line number margin for example: say we have a code editor has 1000 lines and 100 lines are visible in the view port, you of course can create 1000 *JLabel*s and add them to the margin. Or you can paint the 100 strings on fly just for the visible 100 lines. Obviously the second approach is much more efficient. *Margin* has *paintMargin* method which you must implement in order to paint the margin.

There are two categories of margins – line margin or non-line margin.

The line margin paints its content line by line. A typical example in this category is the line number margin. Each painting code will just paint the rectangle area belong to that line. In this case, we created *AbstractLineMargin* to make it simple. You need to implement the *paintLineMargin* method in its subclass. Internally, we implemented the *paintMargin* method on *Margin* interface and delegate to *paintLineMargin* to paint line by line. As you can see below, all you need to do is to paint the content for a particular line at the specified rectangle. You should never paint it outside the rectangular area.

```
public void paintLineMargin(Graphics g, Rectangle rect, int line) {
    String lineNumber = "" + (line + 1);
    g.setColor(new Color(128, 0, 0));
```

```
        g.drawString(lineNumber, rect.x + rect.width –
getEditor().getPainter().getFontMetrics().stringWidth(lineNumber) - 3,
                rect.y + getEditor().getPainter().getFontMetrics().getAscent());
    }
```

There are many margins in this category. Breakpoint margin and bookmark margin are two more such examples.

The non-line margin is the opposite of the first category. It is not limited by one line but ranges from one line to another line. In this you don't want to paint line by line but paint it across many lines. A typical example is code folding margin. As the code folding ranges from a start line to an end line, you just paint the whole range in one shot. For the margins in this category, you need to extend *AbstractMargin* directly and implements the *paintMargin* method. For performance reason, you should check the first visible line of the code editor and the total visible line count. If you know you will paint outside the visible area, just don't paint.

LineNumberMargin

Now let's cover look at an example. Here is the *LineNumberMagin*. *LineNumberMagin* extends *AbstractLineMargin* as you may expect.

```
1   /*
2    * @(#)CodeEditorDemo.java 5/30/2006
3    *
4    * Copyright 2002 - 2006 JIDE Software Inc. All rights reserved.
5    */
6
7   import com.jidesoft.dialog.ButtonPanel;
8   import com.jidesoft.dialog.StandardDialog;
9   import com.jidesoft.document.DocumentComponent;
10  import com.jidesoft.editor.*;
11  import com.jidesoft.editor.action.DefaultInputHandler;
12  import com.jidesoft.editor.margin.AbstractLineMargin;
13  import com.jidesoft.editor.margin.CodeFoldingMargin;
14  import com.jidesoft.editor.margin.LineOffsetMargin;
15  import com.jidesoft.editor.margin.BraceMatchingMarginPainter;
16  import com.jidesoft.editor.marker.Marker;
17  import com.jidesoft.editor.marker.MarkerModel;
18  import com.jidesoft.editor.search.*;
19  import com.jidesoft.editor.search.FindAndReplaceAllDialig;
20  import com.jidesoft.editor.settings.FontPanel;
21  import com.jidesoft.editor.settings.StyleListPanel;
22  import com.jidesoft.icons.IconsFactory;
23  import com.jidesoft.plaf.LookAndFeelFactory;
24  import com.jidesoft.shortcut.ShortcutEditor;
25  import com.jidesoft.swing.JideSwingUtilities;
```

Figure 1 LineNumberMargin

Here is the code of *LineNumberMargin*.

```
public class LineNumberMargin extends AbstractLineMargin {
    private static Color DEFAULT_FOREGROUND = new Color(128, 0, 0);

    public LineNumberMargin(CodeEditor editor) {
        super(editor);
        setForeground(DEFAULT_FOREGROUND);
    }
```

```
    public void paintLineMargin(Graphics g, Rectangle rect, int line) {
        CodeEditor editor = getCodeEditor();
        String lineNumber = "" + (editor.viewToModelLine(line) + 1);
        g.setColor(getForeground());
        g.drawString(lineNumber, rect.x + rect.width - editor.getPainter().getFontMetrics().stringWidth(lineNumber) -
3,
                rect.y + editor.getPainter().getFontMetrics().getAscent());
    }

    public int getPreferredWidth() {
        CodeEditor editor = getCodeEditor();
        String maxLineNumber = "" + (editor.getLineCount() + 1);
        return editor.getPainter().getFontMetrics().stringWidth(maxLineNumber) + 6;
    }

    public String getToolTipText(int line) {
        return "" + (line + 1);
    }
}
```

MarginPainter and LineMarginPainter

At the beginning, you most likely use one margin for one purpose. But sooner or later, you will find out that's a waste of screen space. You will start to think if you can combine several margins that are not very busy into one margin. See screenshot below for an example. What you see here is a brace matching margin over the code folding margin. Code folding margin could be very busy but brace matching is not because there is only one brace matching at one time. So it's a perfect use case to paint the brace matching over the code folding to save some space.

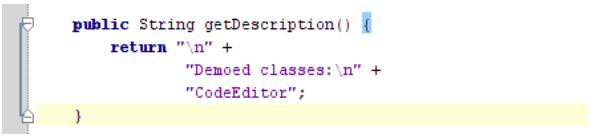


Figure 2 An example of MarginPainter

Now let's see how we do this. We added *addMarginPainter* method on *AbstractMargin*. Assuming you will extend *AbstractMargin* in your margin. If so, you can add your own *MarginPainter* to it.

MarginPainter is a painter interface which can paint the margin area. This painter is mainly used to add extra content to an existing margin. You can add many *MarginPainter*s to *AbstractMargin* In order to decide the order to paint them, each *MarginPainter* has layer index. The lower the layer index is, the earlier it gets painted. In the other word, the painter has a higher index will overwrite those that have lower index. The default layer index is defined as *LAYER_DEFAULT_INDEX* in *MarginPainter* interface. The original content of the margin is painted on this layer. For example, in code folding margin, code folding information is painted on the layer of index *LAYER_DEFAULT_INDEX*. If you want to your painter painted before code folding information, use a layer index smaller than *LAYER_DEFAULT_INDEX*. If you want it painted after the code folding, use an index larger than *LAYER_DEFAULT_INDEX*.

Here is the code of how to adding a margin painter. The result is what the screenshot above shows.

```
CodeFoldingMargin margin = new CodeFoldingMargin();
margin.addMarginPainter(new BraceMatchingMarginPainter());
editor.getMarginArea().addMarginComponent(margin);
```

MarginPainter is mainly for non-line margin. For line margin, there is *addLineMarginPainter* on *AbstractLineMargin*. It is almost the same as *MarginPainter* except it just paints the rectangle area of a particular line.

CodeFoldingMargin

CodeFoldingMargin is a special margin that paints the code folding information. We mentioned it several times when we cover the basic of margin. Now let's say how to use it.

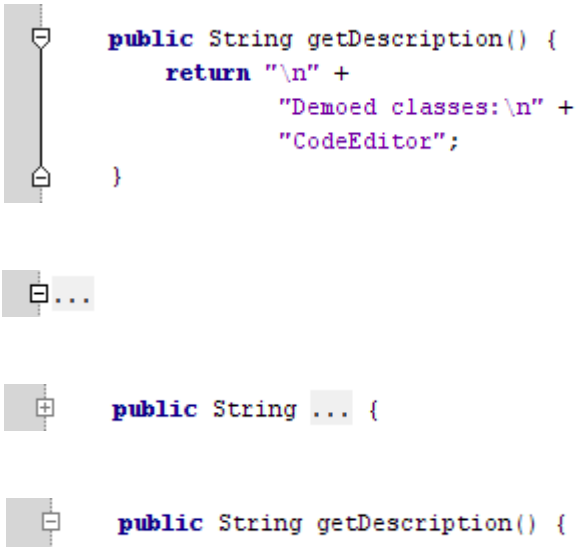


Figure 3 CodeFoldingMargin

As you can see from the screenshot above, each code folding has a start and an end (the 1st screenshot above). Then there is a line between them to connect them. When it is folded, there is just one icon to indicate where the folding is (the 2nd screenshot above), or when the code folding starts and ends on the same line, you will only see one icon when it either expanded or collapsed (the 3rd and 4th screenshots above).

In order to allow you to customize how the code folding is painted, we added *setCodeFoldingPainter* to allow you set a *CodeFoldingPainter*. *CodeFoldingPainter* has all methods you can implement in order to paint each part of the code folding. Here are a few examples after we set our own customized painter.

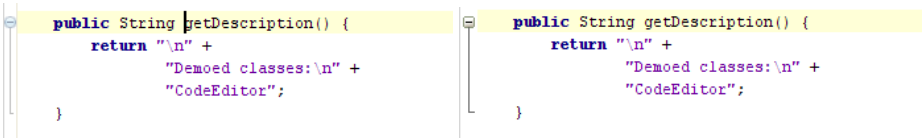
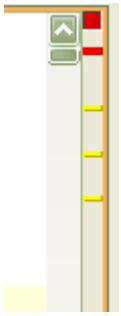


Figure 4 Different CodeFoldingPainter to mimic Eclipse and NetBeans

Marker Area and Code Inspection



Marker Area is a special area on the right side of a code editor, after the vertical scroll bar. See the picture on the right for an example. Since it appears outside the scroll bar, it doesn't scroll with the text in code editor. So you can use it to display the lines which have errors, todo's or highlights in the text. Users can see all those in one shot no matter how large the text is. They can also click on the line marker on the inspection area and jump right to the line.

Marker and MarkerModel

Marker represents a range of text in code editor. It has a start offset and an end offset. By default, there are two types of markers - error and warning. But you can always define your own types of markers. You can also associate a tool tip with a marker. The tool tip will be shown when user mouse moves over the marker stripe.

MarkerModel is the model class that stores all the markers in a code editor. You can use this class to add/remove/update markers. The change will be shown on the *MarkerArea* immediately.

Code Inspection

The main usage of *MarkerArea* is for code inspection.

You can add your own code inspector to inspect the code. The inspector could be a grammar inspector as most editors for IDEs have. But it also can be any kind of code inspectors that will scan the code to detect the information it is looking for. For example you can have TODO inspector to detect any TODO strings in the code.

See below for an example of grammar inspection. Please note, we used a *PHPParser* class that can parse php code and generate grammar errors and warnings in order to add them as *Marker*.

```
_editorForPhp.addCodeInspector(new CodeInspector() {
  public void inspect(final CodeEditor codeEditor, final MarkerModel markerModel) {
    PHPParser parser = new PHPParser();
    parser.setPhp5Enabled(true);
    parser.addParserListener(new PHPParserListener() {
      public void parseError(PHPParseErrorEvent e) {
        markerModel.addMarker(
            codeEditor.getLineStartOffset(e.getBeginLine() - 1) + e.getBeginColumn() - 1,
            codeEditor.getLineStartOffset(e.getEndLine() - 1) + e.getEndColumn() - 1,
            Marker.TYPE_ERROR, e.getMessage());
```

```
            }

            public void parseMessage(PHPParseMessageEvent e) {
                markerModel.addMarker(
                        codeEditor.getLineStartOffset(e.getBeginLine() - 1) + e.getBeginColumn() - 1,
                        codeEditor.getLineStartOffset(e.getEndLine() - 1) + e.getEndColumn() - 1,
                        Marker.TYPE_WARNING, e.getMessage());
            }
        });

        try {
            markerModel.setAdjusting(true);
            markerModel.clearMarkers();
            parser.parse(codeEditor.getText());
        }
        catch (ParseException e) {
            e.printStackTrace();
        }
        finally {
            Runnable runnable = new Runnable() {
                public void run() {
                    markerModel.setAdjusting(false);
                }
            };
            SwingUtilities.invokeLater(runnable);
        }
    }
});
```

In the current release, we didn't provide any build-in parsers except *PHPParser* which is packaged as part of the example. But you can find quite a few parser implementations from the web[1]. As long as the parser can generate a list of errors and warnings with their locations, you can use it along with *CodeInspector* to display them on marker area.

Marker Eye and Marker Stripes

MarkerEye is at the top of the marker area. It indicates the inspecting status. The paint of *MarkerEye* is done by a class called *MarkerEyePainter*. By default, *DefaultMarkerEyePainter* is used. You can always set your own painter by calling *setPainter(MarkerEyePainter)*.

[1] There are *PHPParser* and *HDLParser* from http://plugins.jedit.org/list.php?category=6. The php one is the one we used in our demo. You can also find more from sourceforge.net and java.net. You can also write one yourself using tools such as *JavaCC*.

MarkerStripe is a panel below the *MarkerEye* to display all the markers in a marker model. The paint of each stripe is done by a class called *MarkerStripePainter*. By default, *DefaultMarkerStripePainter* is used. You can always set your own painter by calling *setPainter(MarkerStripePainter)*.

Code Folding

JIDE Code Editor supports code folding. We talked about code folding margin which is used display code folding information in margin area. Here we will talk about the insider implementation of code folding in *CodeEditor*. All code folding information is kept in *FoldingModel*. You can get it using *CodeEditor*'s *getFoldingModel()* method.

DefaultFoldingModel is the default implementation of *FoldingModel*. It stores the code folding information as *FoldingSpan*. *FoldingSpan* has the start offset, the end offset and a description. Of course it also has a flag to indicate whether the folding is expanded. The description will be used to display in the editor when folding span is collapsed.

FoldingModel allows you to add your own folding information. In general, you should always call *setAdjusting* to true before you add code folding in batch mode. After you are done adding, you call *setAdjusting* and set it back to false.

```
_editorForJava.getFoldingModel().setAdjusting(true);

_editorForJava.getFoldingModel().addFoldingSpan(120, 1279, "...");
_editorForJava.getFoldingModel().addFoldingSpan(2013, 2053, "...");
..... // add more folding spans

_editorForJava.getFoldingModel().setAdjusting(false);
```

FoldingModel also supports *FoldingSpanListener* so that you can listen to any changes that happen to the *FoldingModel*. *FoldingSpanEvent* is the event that is used to deliver the folding changes. Please note, in general, you shouldn't do any update or repaint if the isAdjusting is true in the event. For the efficiency, only when you get either *FOLDING_SPAN_END_ADJUSTING* event or the individual event's *isAdjusting* is false, you should update.

Find and Replace

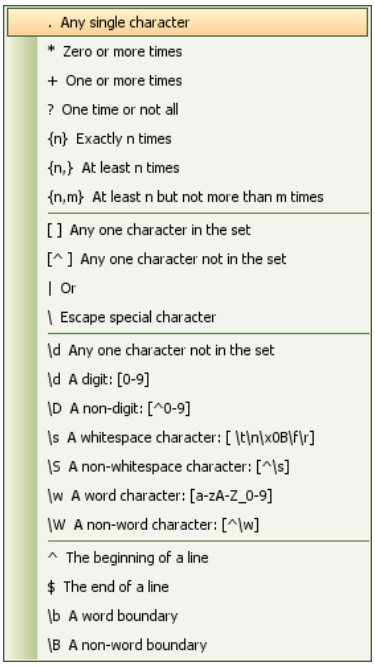
JIDE Code Editor provides a very flexible find and replace feature. As we can't anticipate how users will use *CodeEditor,* we designed it so that it allows you to fully customize the way find and replace works. It can perform find and replace on one *CodeEditor*, on several *CodeEditor*s, on part of the text within a *CodeEditor* (such as the selected text in a *CodeEditor*), or even on any JTextComponent (such as JTextPane, JTextArea etc), a file, a list of files.

FindAndReplace

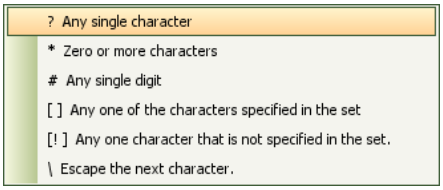
FindAndReplace is the main class in this module. It actually does the matching of the searching text with the text to be searched. It keeps track of the find/replace text history. It has

several flags such as matching cases, matching whole word. It also allows you to use regular expression or wildcards during searching.

Here is the list of commonly used regular expressions.



Here is the list of wild card expressions.



FindAndReplaceTarget

Although it's part of *JIDE Code Editor*, *FindAndReplace* doesn't depend on *CodeEditor*. It works on an interface called *FindAndReplaceTarget*. You can think *FindAndReplaceTarget* as something that has a piece of several pieces of text that need to be searched. Having this abstraction level allows *FindAndReplace* works independently any concrete of text components. There are several concrete implementations of *FindAndReplaceTarget*. *CodeEditorFindAndReplaceTarget* is the one for *CodeEditor*. *CodeEditorSelectionFindAndReplaceTarget* is the one for the selected text of a *CodeEditor*. We also have *CodeEditorDocumentPaneFindAndReplaceTarget* which is for the DocumentPane that contains several *CodeEditors*.

FindAndReplace can support many *FindAndReplaceTargets* if you want. A typical example is you want to search on the whole *CodeEditor* or the selected text of a *CodeEditor*. As we mentioned above, they are actually two different targets. So we will add both targets to *FindAndReplace*. But when user does a searching, only one target will be selected and used. This is done by the next class we will cover – the *FindAndReplacePanel*.

FindAndReplacePanel

FindAndReplacePanel is the configuration panel for *FindAndReplace*. All options provided by *FindAndReplace* can be configured in this panel.

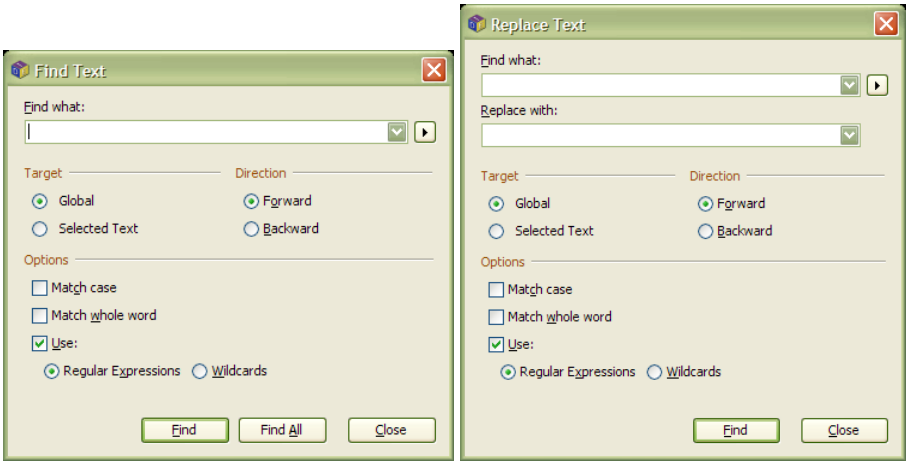


Figure 5 Find and Replace Panel used in a dialog

There are three sections in this panel. The first section is where user input Find text and Replace text. As you can see, it keeps the history of the find and replace history in a JComboBox. The left arrow button next to "Find what" JCombBox is to show a popup menu to help user to input the regular expression or wild card.

The second section is the target panel. As we mentioned, any number of targets can be added to *FindAndReplace*. *FindAndReplacePanel* will display them vertically using BoxLayout and put a radio button before each one. User can select one target as the active target.

The last section is the option panel.

FindResult and FindResults

FindAndReplace can do interactive searching. If it works on a *CodeEditor*, you will see the matching text is highlighted in *CodeEditor*. Pressing F3 will highlight the next occurrence (or shift-F3 for the previous occurrence), until it reaches the end. If it is in replace mode, a prompt dialog will ask user to replace or skip the current occurrence. Opposite to interactive searching, there is find all. It will find all the occurrences in one shot without user interaction. In this case, we used two classes *FindResult* and *FindResults* to store the results. *FindResults* can have one or more *FindResult*(s) or *FindResults(*es) so that it can form a tree structure. *FindResultTree* can display *FindResults* in a JTree.

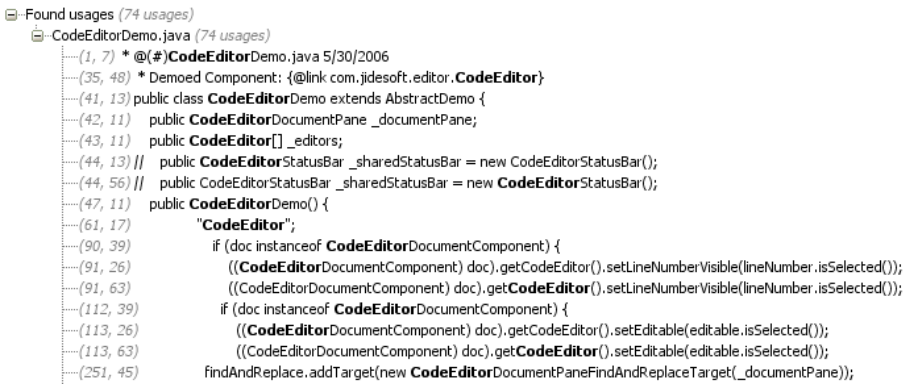


Figure 6 FindResults displayed in a tree

Listener Support

To perform a find all operation could take a long time. So we need to provide users the necessary feedback so that they know what is happening. That's why we added *FindAndReplaceListener* to *FindAndReplace*. This listener will use *FindAndReplaceEvent* to inform you when searching starts, ends, when a text is found and when a text is replaced.

Each event will have a status id and some optional fields depending on what kind of status it is. There are five possible statuses in *FindAndReplaceEvent*.

See below for a list of possible statuses and which optional fields are available in each status.

SEARCH_STARTED: this event is fired when searching or replacing starts. There is no data in this event.

SEARCH_STARTED_TEXT: this event is fired when it starts to work on a new piece of text. getTargetName() will give you the new name. This event will fire even if you just have one piece of text, so you can always use this event to find out the name of the text. During searching-all, you can use the name to display a status message so that user knows

SEARCH_FINISHED: this event is fired when searching or replacing is done. If you are doing a searching-all, getFindResults() will tell you all the find results. If you are doing a interactive searching/replacing, getFindResults() will be null.

SEARCH_FINISHED_TEXT: this event is fired when searching or replacing is done with a piece of text. If you are doing a searching-all, getFindResults() will tell you all the find results. If you are doing an interactive searching/replacing, getFindResults() will be null. In either case, getTargetName() will always tell you the text name, just like in SEARCH_STARTED_TEXT.

SEARCH_FOUND: This event is fired when a matching text is found. getFindResult() will always be a non-null value which tells you the start and end offset of the matching text. This event is fired for both interactive search and search-all.

SEARCH_REPLACED: This event is fired when a matching text is found and replaced with the replacement text. getFindResult() will always be a non-null value which tells you the start and end offset of the matching text. getReplaceText() will tell you the replacement text. This event is fired for both interactive replace and replace-all.

Lazy Loading

Sometimes, you may feel the need to read huge files, for example, system logs. In this case, *CodeEditor#setText()* would not be a good choice as it takes too much memory to read everything into a String. JIDE *CodeEditor* offers *LazyLoadDocument* to read/edit those huge files.

Loading large file

To load a large file, please use *CodeEditor#setFileName()* in this case instead of using *setText*. Internally, we will use *LazyLoadDocument* as the *Document* class for *CodeEditor*. *LazyLoadDocument* does not read all the contents from the file immediately. Instead, it reads the first page and display it to user right away. While the user scrolls down the pages, the *LazyLoadDocument* will read in more pages when needed. It will also unload pages that are not displaying and not edited when the loaded page reaching the maximum pages. You could invoke *setMaximumPages* and *setPageLineSize* to balance the memory usage and file read chances. By default, the maximum pages are 5 and the line size in each page is 10000.

Saving large file

After editing, please invoke *exportToOutputStream* to export the edited text to the designated output stream. Please make sure you will not invoke *getText* or *getRawText* if the file is large.

PageLoadEvent

PageLoadEvent is fired each time a new page is being loaded into memory or has been finished loading. You can register this listener to *LazyLoadDocument* to control the UI behavior. It could take a few seconds to a few minutes to load a page, depending on how large the file is and how fast your computer is. When loading a page, we will use *Overlayable* feature in JIDE Common Layer to display a spinning circle on the editor to give user a hint that something is loading. To use this feature, all you need to do is to add *codeEditor.createOverlay()* to parent container instead of adding *codeEditor* to its parent container.

Performance

With the lazy loading feature, the memory usage will be reduced to several pages and it's possible to spend more time while loading new pages. Below are some performance benchmark FYI.

To open a log file with its size as 1.54G bytes (5.9 million lines), the UI will be shown up within 1 second. It should take very few milliseconds to perform any actions, like select all, insert a char, remove a string, undo, redo, etc. However, if the page to be displayed is not loaded yet, a page loading process will be triggered. The time spent for page loading depends on the page location within the file and the I/O speed of your machine.

The memory usage would be 8~9M bytes constantly if nothing edited. The memory usage could be increased if you edit any page. Any edited page would not be released for undo/redo purpose.

Potential disabled features

To avoid blocking the UI, we have to disable some features to support lazy loading if the target page is not loaded yet. Those features include bracket highlighting, folding span, TAB support.

Shortcut Keys

We provided a rich set of shortcut keys for *CodeEditor*. See below for a table of all available shortcut keys.

Name	Shortcut Key	Description	Method Name on CodeEditor
Basic Editing			
Backspace	BACK_SPACE	Delete the previous char at the caret position	backspaceChar
Delete to Word Start	control BACK_SPACE	Delete previous chars until it see a non-word char.	backspaceWord
Delete	DELETE	Delete the next char at the caret position	deleteChar
Delete to Word End	control DELETE	Delete next chars until it see a non-word char	deleteWord
Delete Current Line	control Y	Delete current line	deleteLine
Insert Line Break	ENTER	Insert a line break at the caret position	insertBreak
Split Line	control ENTER	Insert a line break at the caret position and keep current caret position	splitLine
Start New Line	shift ENTER	Start a new line next to the caret line and put caret at the beginning of the new line.	startNewLine
Indent Selection	TAB	Indent the caret line if no selection and all selected lines if there is selection.	indentSelection
Unindent Selection	shift TAB	Indent the caret line if no selection and all selected lines if there is selection.	unindentSelection
Join Lines	control shift J	Join the next line with caret line if no selection, or join all selected lines as one line if	joinLines

		there is selection.	
Toggle Insert/Overwrite	INSERT	Toggle the override/insert status.	toggleOverwrite
Toggle Rectangular Selection	control BACK_SLASH alt shift INSERT	Toggle from regular selection to rectangular selection	setColumnSelectionMode
Toggle Case	control shift U	Toggle the case of the selection	toggleCase
Change Caret Position and Selection			
Select All	control A	Select all the text in the editor	selectAll
Move Caret to Line Start	HOME	Move caret to the start of the current line[2]	moveToLineStart(false)
Move Caret to Line End	END	Move caret to the end of the current line	moveToLineEnd(false)
Select to Line Start	shift HOME	Select from the current caret position all the way to the line start	moveToLineStart(true)
Select to Line End	shift END	Select from the current caret position all the way to the line end	moveToLineEnd(true)
Move Caret to Document Start	control HOME	Moves caret to the start of the code editor	moveToDocumentStart(false)
Move Caret to Document End	control END	Moves caret to the end of the code editor	moveToDocumentEnd(false)
Select to Document Start	control shift HOME	Select from the current caret position all the way to the document start	moveToDocumentStart(true)
Select to Document End	control shift END	Select from the current caret position all the way to the document end	moveToDocumentEnd(true)
Page Up	PAGE_UP	Move caret one page up	moveToPreviousPage(false)
Page Down	PAGE_DOWN	Move caret one page down	moveToNextPage(false)

[2] It supports smart home feature. If there are leading spaces, the first Home key will move caret to the first non-space char. The second Home key will move caret to the first position. If you press Home again and again, it will toggle between the first non-space position and the first position.

Select to Previous Page	shift PAGE_UP	Select from the current caret position up by one page	moveToPreviousPage(true)
Select to Next Page	shift PAGE_DOWN	Select from the current caret position down by one page	moveToNextPage(true)
Move Caret to Previous Char	LEFT	Move caret one char left	moveToPreviousChar(false)
Move Caret to Next Char	RIGHT	Move caret one char right	moveToNextChar(false)
Select Previous Char	shift LEFT	Select the previous char of the current caret position	moveToPreviousChar(true)
Select Next Char	shift RIGHT	Select the current char of the current caret position	moveToNextChar(true)
Move Caret to Previous Word	control LEFT	Move caret to the previous word – the first space char before the current caret position.	moveToPreviousWord(false)
Move Caret to Next Word	control RIGHT	Move caret to the next word – the first space char after the current caret position.	moveToNextWord(false)
Select Previous Word	control shift LEFT	Select the current caret position to the first space char before it	moveToPreviousWord(true)
Select Next Word	control shift RIGHT	Select the current caret position to the first space char after it	moveToNextWord(true)
Move Caret to Previous Line	UP	Move caret up by one line	moveToPreviousLine(false)
Move Caret to Next Line	DOWN	Move caret down by one line	moveToNextLine(false)
Select Previous Line	shift UP	Select from the caret position up by one line	moveToPreviousLine(true)
Select Next Line	shift DOWN	Select from the caret position down by one line	moveToNextLine(true)
Goto Line	control G	Prompt a dialog to let user type in a line index and scroll to it	promptGotoLine
Select Word at Caret	control W	Select the current word	selectWord
Select to Matching	control B	Select the current block that	selectToMatchingBracket

Bracket		starts and ends with two matching brackets.	
Duplication Selection	control D	Duplication the selection. If no selection, the caret line will be duplicated	duplicateSelection
Line Comments	control SLASH	Using line style comment for comment a line or several lines.	lineComments
Block Comments	control shift SLASH	Using block style comment a line or several lines.	blockComments
Undo/Redo			
Undo	control Z	Undo the last editing operation	undo
Redo	control shift Z	Redo the last undone editing operation	redo
Clipboard Operations			
Clipboard Cut	control X shift DELETE	Cut the currently selected text. If nothing is selected, cut the current line	clipboardCut
Clipboard Copy	control C control INSERT	Copy the currently selected text. If nothing is select, copy the current line	clipboardCopy
Clipboard Paste	control V shift INSERT	Paste whatever on the clipboard to the current caret position	clipboardPaste
Clipboard Paste with Dialog	control shift V control shift INSERT	Prompt a dialog to allow user to select one of the previous clipboards and paste it	pasteWithDialog
Find And Replace			
Find	control F	Prompt a dialog to allow user to type in a text to search for	Find
Find Next Occurrence	F3	Find the next occurrence of the searching text.	findNext
Find Previous Occurrence	shift F3	Find the previous occurrence of the searching text.	findPrevious
Replace	control R	Prompt a dialog to allow user to type in a text to replace with another text	replace

Quick Search	alt F3	Using the Searchable feature to show a popup where user can type in a text without using a dialog	quickSearch
Folding Operations			
Fold Selection	control PERIOD	Create a folding which contains the currently selected text.	toggleFoldingSelection
Expand Folding	control EQUALS	Expand the current folding	expandFolding
Collapse Folding	control MINUS	Collpse the current folding	collapseFolding
Expand All	control shift EQUALS	Expand all the foldings	expandAll
Collapse All	control shift MINUS	Collapse all the foldings.	collapseAll

The table above shows the default shortcut keys for the action. You can always define your own action. We used JIDE Shortcut Editor as the editor to configure the shortcut keys. Here is the code to create an editor for it.

```
DefaultInputHandler inputHandler = (DefaultInputHandler) DefaultSettings.getDefaults().getInputHandler();
ShortcutEditor shortcutEditor = new ShortcutEditor(inputHandler.getShortcutSchemaManager(), true);
```

See below for a screenshot of the shortcut editor. We put it in a dialog as a demo but you can use it anywhere you want as it's just a *JPanel*.

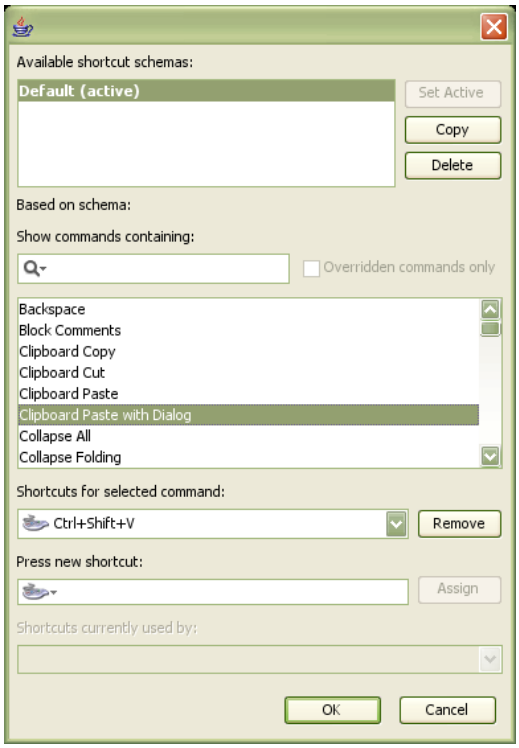


Figure 7 Shortcut Editor for CodeEditor

Line Break

There are three kinds of line breaks. If you need to work with different platforms, such as a PC running Windows, a web server running Linux, you will have to deal with all these kinds of line breaks.

Windows, and DOS before it and OS/2, uses a pair of CR[3] and LF characters to terminate lines. UNIX (Including Linux, FreeBSD, AIX, Mac OS X, BeOS, and Amiga) uses an LF character only. The Apple Macintosh OS through version 9, finally, uses a CR character only.

Problems arise when transferring text files between different operating systems and using software that is not smart enough to detect the line break style used by a file. E.g. if you open a UNIX file in Microsoft Notepad, it will display the text as if the file contained no line breaks at all. If you open a Windows file in a UNIX editor, you will see a control character (the CR) at the end of each line. Older versions of Perl on Linux would refuse to run any script that used Windows line breaks, aborting with an unhelpful error message.

JIDE Code Editor addressed all those issues by allowing reading and writing using any of the line break styles above.

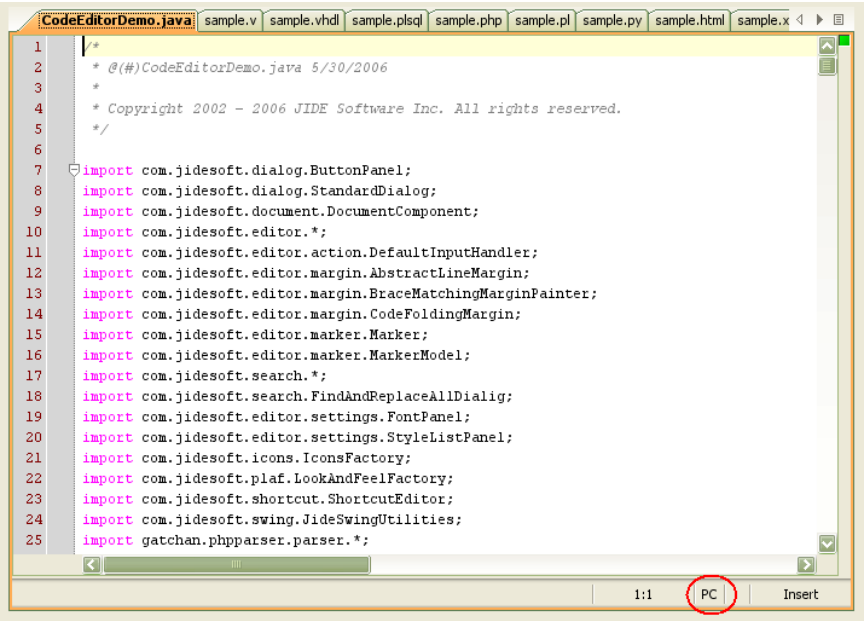
[3] CR == '\r' whose value is 0xA and LF == '\n' whose value is 0x0D

When *CodeEditor*'s *setText()* method is called, we will parse the text and see what line break the text uses. If the same line break is consistently used all over the text, we will set the *lineBreakStyle* property to one of the following three values (LINE_BREAK_PC, LINE_BREAK_UNIX, LINE_BREAK_MAC). When you call *getText()*, we will make sure the same line breaks are used in the returned text.

If the line break is not consistently used in the text, we will set the *lineBreakStyle* property to LINE_BREAK_MIXED. You can call *setLineBreakStyle* to set it to the one you want. If you don't, *getText()* will use the line break style returned from *getDefaultLineBreakStyle()*.

```
public int getDefaultLineBreakStyle() {
    if (SystemInfo.isWindows() || SystemInfo.isMacOSX()) {
        return LINE_BREAK_PC;
    } else if (SystemInfo.isMacClassic()) {
        return LINE_BREAK_MAC;
    } else if (SystemInfo.isUnix()) {
        return LINE_BREAK_UNIX;
    } else {
        return LINE_BREAK_PC;
    }
}
```

To make it easy for end user to control this, we also provided *LineBreakStatusBarItem*, an optional status bar item you can add to your status bar.



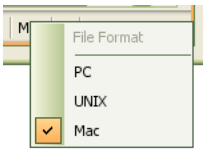


Figure 8 Line Break Status Bar Item

As you can see in the screenshots above, whenever you open a file, the line break style will be displayed on the status bar. If user clicks on it, they will see a popup menu which allows them to change the line break style. This will affect the line break styles used by *getText()* method.

Internally, we only use a single CR as the line break so that *CodeEditor* doesn't need to deal with various line break styles during runtime. If you call *getRawText()* or *getText(int start, int len)*, you will get the text we used internally which only uses CR as line break. There is also *importText(String in, StringBuffer out)* method which can convert from whatever line break style to CR only line break style. The opposite direction is done by *exportText(String in, StringBuffer out)* method which converts CR only line break style to the line break style specified by *lineBreakStyle* property.

Clipboard operation will be affected by line break too. The *clipboardCopy()* and *clipboardCut()* method will always put the text on the clipboard using line break style specified by *lineBreakStyle* property. On the other hand, *clipboardPaste()* method can accept a piece of text with any line break styles. But it will affect the *lineBreakStyle* property as *setText()* does.