TuxCare's Annual Open Source Survey is here! Share your opinions and help inform our yearly report. →
Start Survey

Using AvalonEdit (WPF Text Editor)

Oct 5, 2009 12 min read words

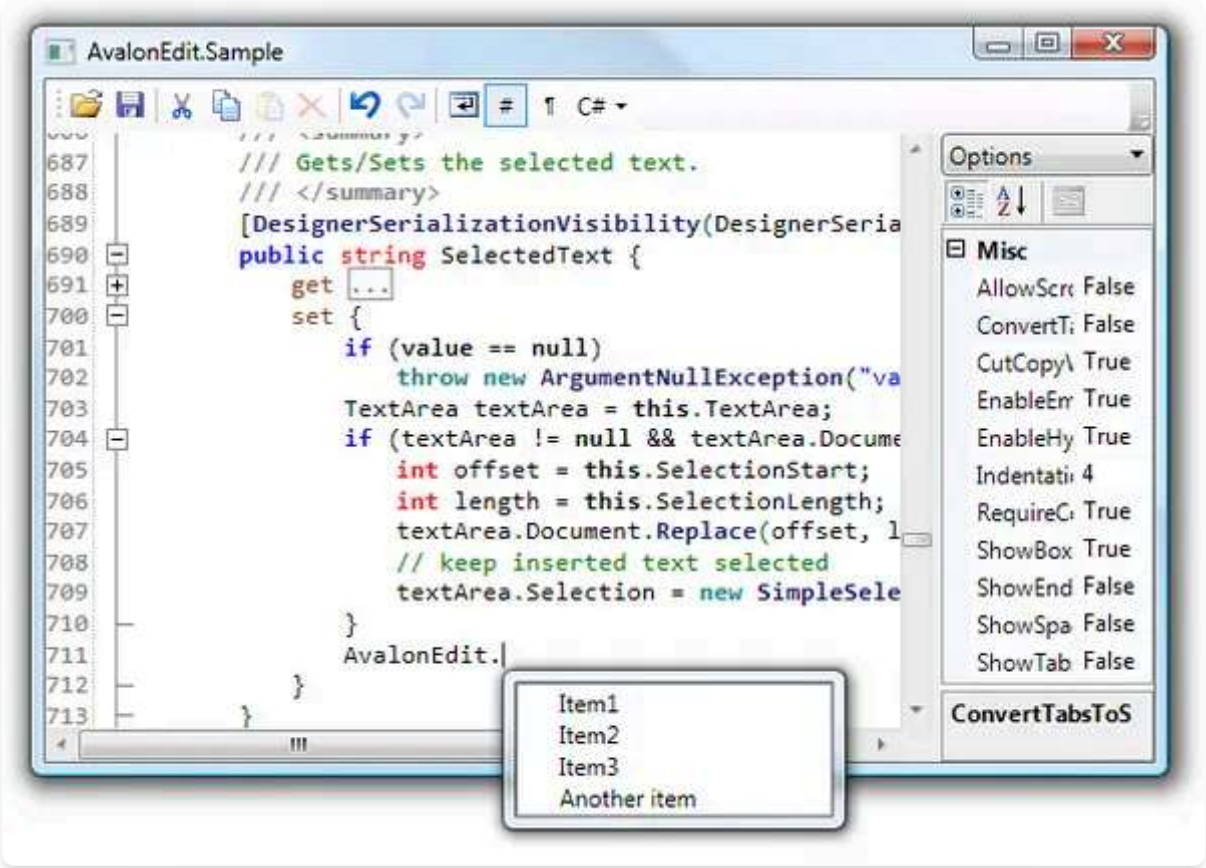
C#3.0 .NET3.5 C# .NET Dev XAML WPF Intermediate Advanced C#4.0 .NET4

by Daniel Grunwald
Contributor

2016k Views 4.95/ 5

- Download Source Code - 975.1 KB
- Download Binaries - 289.1 KB
- Download CHM documentation file - 2.9 MB
- NuGet package

The latest version of AvalonEdit can be found as part of the SharpDevelop project. For details on AvalonEdit, please see www.avalonedit.net.



Introduction

`ICSharpCode.AvalonEdit` is the WPF-based text editor that I've written for SharpDevelop 4.0. It is meant as a replacement for ICSharpCode.TextEditor, but should be:

- Extensible
- Easy to use
- Better at handling large files

Extensible means that I wanted SharpDevelop add-ins to be able to add features to the text editor. For example, an add-in should be able to allow inserting images into comments – this way, you could put stuff like class diagrams right into the source code!

With, **Easy to use**, I'm referring to the programming API. It should just work™. For example, this means if you change the document text, the editor should automatically redraw without having to call `Invalidate()`. And, if you do something wrong, you should get a meaningful exception, not corrupted state and crash later at an unrelated location.

Better at handling large files means that the editor should be able to handle large files (e.g., the mscorlib XML documentation file, 7 MB, 74100 LOC), even when features like folding (code collapsing) are enabled.

Using the Code

The main class of the editor is `ICSharpCode.AvalonEdit.TextEditor`. You can use it just similar to a normal WPF `TextBox`:

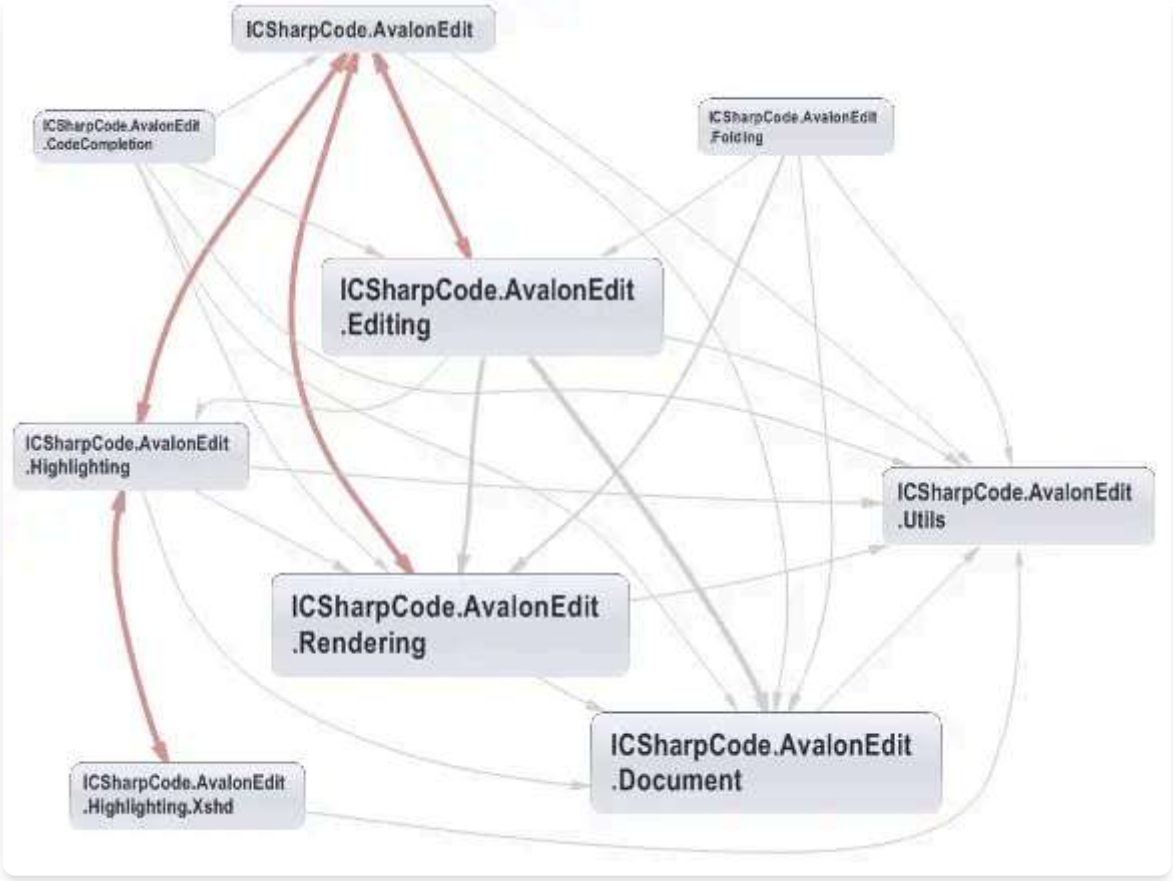
XML

```
<avalonEdit:TextEditor
    xmlns:avalonEdit="http://icsharpcode.net/sharpdevelop/avalonedit"
    Name="textEditor"
    FontFamily="Consolas"
    SyntaxHighlighting="C#"
    FontSize="10pt"/>
```

AvalonEdit has syntax highlighting definitions built in for: ASP.NET, Boo, Coco/R grammars, C++, C#, HTML, Java, JavaScript, Patch files, PHP, TeX, VB, and XML.

If you need more of AvalonEdit than a simple text box with syntax highlighting, you will first have to learn more about the architecture of AvalonEdit.

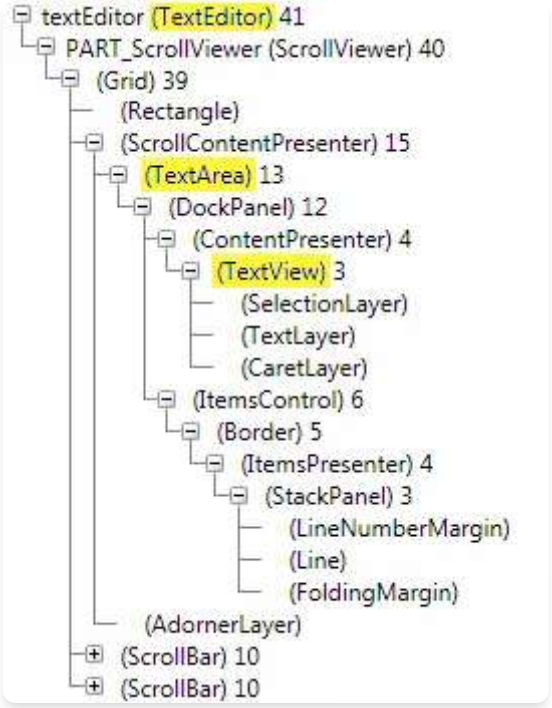
Architecture



As you can see in this dependency graph, AvalonEdit consists of a few sub-namespaces that have cleanly separated jobs. Most of the namespaces have a kind of 'main' class.

- `ICSharpCode.AvalonEdit.Utils`: Various utility classes
- `ICSharpCode.AvalonEdit.Document`: `TextDocument` — text model
- `ICSharpCode.AvalonEdit.Rendering`: `TextView` — extensible view onto the document
- `ICSharpCode.AvalonEdit.Editing`: `TextArea` — controls text editing (e.g., caret, selection, handles user input)
- `ICSharpCode.AvalonEdit.Folding`: `FoldingManager` — enables code collapsing
- `ICSharpCode.AvalonEdit.Highlighting`: `HighlightingManager` — highlighting engine
- `ICSharpCode.AvalonEdit.Highlighting.Xshd`: `HighlightingLoader` — XML syntax highlighting definition support (*.xshd* files)
- `ICSharpCode.AvalonEdit.CodeCompletion`: `CompletionWindow` — shows a drop-down list for code completion
- `ICSharpCode.AvalonEdit`: `TextEditor` — the main control that brings it all together

Here is the visual tree of the `TextEditor` control:



It's important to understand that AvalonEdit is a composite control with three layers: `TextEditor` (main control), `TextArea` (editing), `TextView` (rendering). While the main control provides some convenience methods for common tasks, for most advanced features, you have to work directly with the inner controls. You can access them using `textEditor.TextArea` or `textEditor.TextArea.TextView`.

Document (The Text Model)

The main class of the model is `ICSharpCode.AvalonEdit.Document.TextDocument`. Basically, the document is a `StringBuilder` with events. However, the `Document` namespace also contains several features that are useful to applications working with the text editor.

In the text editor, all three controls (`TextEditor`, `TextArea`, `TextView`) have a `Document` property pointing to the `TextDocument` instance. You can change the `Document` property to bind the editor to another document. It is possible to bind two editor instances to the same document; you can use this feature to create a split view.

Here is the *simplified* definition of the `TextDocument`:

CS

```cs
public sealed class TextDocument : ITextSource
{
    public event EventHandler<DocumentChangeEventArgs> Changing;
    public event EventHandler<DocumentChangeEventArgs> Changed;
    public event EventHandler TextChanged;
```

```
        public IList<DocumentLine> Lines { get; }
        public DocumentLine GetLineByNumber(int number);
        public DocumentLine GetLineByOffset(int offset);
        public TextLocation GetLocation(int offset);
        public int GetOffset(int line, int column);

        public char GetCharAt(int offset);
        public string GetText(int offset, int length);

        public void Insert(int offset, string text);
        public void Remove(int offset, int length);
        public void Replace(int offset, int length, string text);

        public string Text { get; set; }
        public int LineCount { get; }
        public int TextLength { get; }
        public UndoStack UndoStack { get; }
    }
```

In AvalonEdit, an index into the document is called an **offset**.

Offsets usually represent the position between two characters. The first offset at the start of the document is 0; the offset after the first `char` in the document is 1. The last valid offset is `document.TextLength`, representing the end of the document. This is exactly the same as the 'index' parameter used by methods in the .NET `String` or `StringBuilder` classes.

Offsets are easy to use, but sometimes you need Line / Column pairs instead. AvalonEdit defines a `struct` called `TextLocation` for those.

The document provides the methods `GetLocation` and `GetOffset` to convert between offsets and `TextLocations`. Those are convenience methods built on top of the `DocumentLine` class.

The `TextDocument.Lines` collection contains a `DocumentLine` instance for every line in the document. This collection is read-only to user code, and is automatically updated to reflect the current document content.

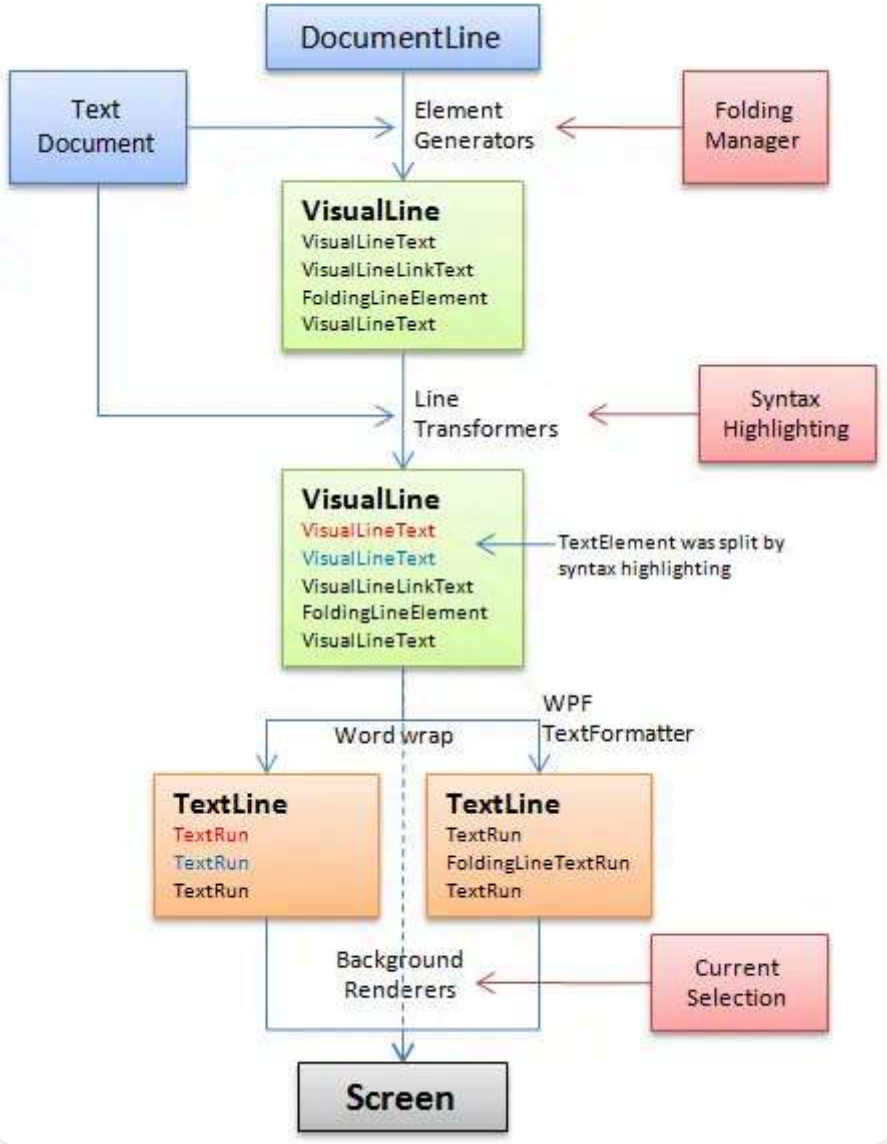
Rendering

In the whole 'Document' section, there was no mention of extensibility. The text rendering infrastructure now has to compensate for that by being completely extensible.

The `ICSharpCode.AvalonEdit.Rendering.TextView` class is the heart of AvalonEdit. It takes care of getting the document onto the screen.

To do this in an extensible way, the `TextView` uses its own kind of model: the `VisualLine`. Visual lines are created only for the visible part of the document.

The rendering process looks like this:



The last step in the pipeline is the conversion to one or more `System.Windows.Media.TextFormatting.TextLine` instances. WPF then takes care of the actual text rendering.

The "element generators", "line transformers", and "background renderers" are the extension points; it is possible to add custom implementations of them to the `TextView` to implement additional features in the editor.

Editing

The `TextArea` class handles user input and executes the appropriate actions. Both the caret and the selection are controlled by the `TextArea`.

You can customize the text area by modifying the `TextArea.DefaultInputHandler` by adding new or replacing existing WPF input bindings in it. You can also set `TextArea.ActiveInputHandler` to something different than the default, to switch the text area into another mode. You could use this to implement an "incremental search" feature, or even a VI emulator.

The text area has the `LeftMargins` property – use it to add controls to the left of the text view that look like they're inside the scroll viewer, but don't actually scroll. The `AbstractMargin` base class contains some useful code to detect when the margin is attached/detached from a text view; or when the active document changes. However, you're not forced to use it; any `UIElement` can be used as the margin.

Folding

Folding (code collapsing) is implemented as an extension to the editor. It could have been implemented in a separate assembly without having to modify the AvalonEdit code. A `VisualLineElementGenerator` takes care of the collapsed sections in the text document, and a custom margin draws the plus and minus buttons.

You could use the relevant classes separately; but, to make it a bit easier to use, the static `FoldingManager.Install` method will create and register the necessary parts automatically.

All that's left for you is to regularly call `FoldingManager.UpdateFoldings` with the list of foldings you want to provide. You could calculate that list yourself, or you could use a built-in folding strategy to do it for you.

Here is the full code required to enable folding:

CS

```
foldingManager = FoldingManager.Install(textEditor.TextArea);
foldingStrategy = new XmlFoldingStrategy();
foldingStrategy.UpdateFoldings(foldingManager, textEditor.Document);
```

If you want the folding markers to update when the text is changed, you have to repeat the `foldingStrategy.UpdateFoldings` call regularly.

Currently, only the `XmlFoldingStrategy` is built into AvalonEdit. The sample application to this article also contains the `BraceFoldingStrategy` that folds using { and }. However, it is a very simple implementation and does not handle { and } inside strings or comments correctly.

Syntax Highlighting

The highlighting engine in AvalonEdit is implemented in the class `DocumentHighlighter`. Highlighting is the process of taking a `DocumentLine` and constructing a `HighlightedLine` instance for it by assigning colors to different sections of the line.

The `HighlightingColorizer` class is the only link between highlighting and rendering. It uses a `DocumentHighlighter` to implement a line transformer that applies the highlighting to the visual lines in the rendering process.

Except for this single call, syntax highlighting is independent from the rendering namespace. To help with other potential uses of the highlighting engine, the `HighlightedLine` class has the method `ToHtml` to produce syntax highlighted HTML source code.

The rules for highlighting are defined using an "extensible syntax highlighting definition" (*.xshd*) file. Here is a complete highlighting definition for a sub-set of C#:

XML

```
<SyntaxDefinition name="C#"
        xmlns="http://icsharpcode.net/sharpdevelop/syntaxdefinition/2008">
    <Color name="Comment" foreground="Green" />
    <Color name="String" foreground="Blue" />

    <!--
```

The highlighting engine works with "spans" and "rules" that each have a color assigned to them. In the XSHD format, colors can be both referenced (`color="Comment"`) or directly specified (`fontWeight="bold" foreground="Blue"`).

Spans consist of two Regular Expressions (begin+end); while rules are simply a single regex with a color. The `<Keywords>` element is just a nice syntax to define a highlighting rule that matches a set of words; internally, a single regex will be used for the whole keyword list.

The highlighting engine works by first analyzing the spans: whenever a begin regex matches some text, that span is pushed onto a stack. Whenever the end regex of the current span matches some text, the span is popped from the stack.

Each span has a nested rule set associated with it, which is empty by default. This is why keywords won't be highlighted inside comments: the span's empty ruleset is active there, so the keyword rule is not applied.

This feature is also used in the string span: the nested span will match when a backslash is encountered, and the character following the backslash will be consumed by the end regex of the nested span (`.` matches any character). This ensures that `\"` does not denote the end of the string span; but `\\"` still does.

What's great about the highlighting engine is that it highlights only on-demand, works incrementally, and yet usually requires only a few KB of memory even for large code files.

On-demand means that when a document is opened, only the lines initially visible will be highlighted. When the user scrolls down, highlighting will continue from the point where it stopped the last time. If the user scrolls quickly, so that the first visible line is far below the last highlighted line, then the highlighting engine still has to process all the lines in between – there might be comment starts in them. However, it will only scan that region for changes in the span stack; highlighting rules will not be tested.

The stack of active spans is stored at the beginning of every line. If the user scrolls back up, the lines getting into view can be highlighted immediately because the necessary context (the span stack) is still available.

Incrementally means that even if the document is changed, the stored span stacks will be reused as far as possible. If the user types `/*`, that would theoretically cause the whole remainder of the file to become highlighted in the comment color. However, because the engine works on-demand, it will only update the span stacks within the currently visible region and keep a notice 'the highlighting state is not consistent between line X and line X+1', where X is the last line in the visible region. Now, if the user would scroll down, the highlighting state would be updated and the 'not consistent' notice would be moved down. But usually, the user will continue typing and type `*/` only a few lines later. Now, the highlighting state in the visible region will revert to the normal 'only the main ruleset is on the stack of active spans'. When the user now scrolls down below the line with the 'not consistent' marker, the engine will notice that the old stack and the new stack are identical, and will remove the 'not consistent' marker. This allows reusing the stored span stacks cached from before the user typed `/*`.

While the stack of active spans might change frequently inside the lines, it rarely changes from the beginning of one line to the beginning of the next line. With most languages, such changes happen only at the start and end of multiline comments. The highlighting engine exploits this property by storing the list of span stacks in a special data structure (`ICSharpCode.AvalonEdit.Utils.CompressingTreeList`). The memory usage of the

highlighting engine is linear to the number of span stack changes; not to the total number of lines. This allows the highlighting engine to store the span stacks for big code files using only a tiny amount of memory, especially in languages like C# where sequences of `//` or `///` are more popular than `/* */` comments.

Code Completion

AvalonEdit comes with a code completion drop down window. You only have to handle the text entering events to determine when you want to show the window; all the UI is already done for you.

Here's how you can use it:

CS

```
        // in the constructor:
        textEditor.TextArea.TextEntering += textEditor_TextArea_TextEntering;
        textEditor.TextArea.TextEntered += textEditor_TextArea_TextEntered;
    }

    CompletionWindow completionWindow;

    void textEditor_TextArea_TextEntered(object sender, TextCompositionEventArgs
    e)
    {
        if (e.Text == ".") {
            // Open code completion after the user has pressed dot:
            completionWindow = new CompletionWindow(textEditor.TextArea);
            IList<ICompletionData> data =
    completionWindow.CompletionList.CompletionData;
            data.Add(new MyCompletionData("Item1"));
            data.Add(new MyCompletionData("Item2"));
            data.Add(new MyCompletionData("Item3"));
            completionWindow.Show();
            completionWindow.Closed += delegate {
                completionWindow = null;
            };
        }
    }

    void textEditor_TextArea_TextEntering(object sender, TextCompositionEventArgs
    e)
```

```
    {
        if (e.Text.Length > 0 && completionWindow != null) {
            if (!char.IsLetterOrDigit(e.Text[0])) {
                // Whenever a non-letter is typed while the completion window is
    open,
                // insert the currently selected element.
                completionWindow.CompletionList.RequestInsertion(e);
            }
        }
        // Do not set e.Handled=true.
        // We still want to insert the character that was typed.
    }
```

This code will open the code completion window whenever '.' is pressed. By default, the `CompletionWindow` only handles key presses like Tab and Enter to insert the currently selected item. To also make it complete when keys like '.' or ';' are pressed, we attach another handler to the `TextEntering` event and tell the completion window to insert the selected item.

The `CompletionWindow` will actually never have focus - instead, it hijacks the WPF keyboard input events on the text area and passes them through its `ListBox`. This allows selecting entries in the completion list using the keyboard and normal typing in the editor at the same time.

For the sake of completeness, here is the implementation of the `MyCompletionData` class used in the code above:

CS

```
    /// Implements AvalonEdit ICompletionData interface to provide the entries in
    the
    /// completion drop down.
    public class MyCompletionData : ICompletionData
    {
        public MyCompletionData(string text)
        {
            this.Text = text;
        }

        public System.Windows.Media.ImageSource Image {
            get { return null; }
```

```
        }

        public string Text { get; private set; }

        // Use this property if you want to show a fancy UIElement in the list.
        public object Content {
            get { return this.Text; }
        }

        public object Description {
            get { return "Description for " + this.Text; }
        }

        public void Complete(TextArea textArea, ISegment completionSegment,
            EventArgs insertionRequestEventArgs)
        {
            textArea.Document.Replace(completionSegment, this.Text);
        }
    }
```

Both the content and the description shown may be any content acceptable in WPF, including custom UIElements. You may also implement custom logic in the `Complete` method if you want to do more than simply insert text. The `insertionRequestEventArgs` can help decide which kind of insertion the user wants - depending on how the insertion was triggered, it is an instance of `TextCompositionEventArgs`, `KeyEventArgs`, or `MouseEventArgs`.

History

- August 13, 2008: Work on AvalonEdit started
- November 7, 2008: First version of AvalonEdit added to the SharpDevelop 4.0 trunk
- June 14, 2009: The SharpDevelop team switches to SharpDevelop 4 as their IDE for working on SharpDevelop; AvalonEdit starts to get used for real work
- October 4, 2009: This article first published on The Code Project
- June 13, 2010: Updated downloads to AvalonEdit 4.0.0.5950 (SharpDevelop 4.0 Beta 1)
- September 13, 2011: Updated downloads to AvalonEdit 4.1.0.7916 (SharpDevelop 4.1 RC)
    - Lots of bugs fixed
    - Improved Performance

    - Now targeting .NET 4.0
- May 12, 2012: Updated downloads to AvalonEdit 4.2.0.8783 (SharpDevelop 4.2)
    - Added SearchPanel
    - Added support for virtual space
    - Some bugfixes
- March 3, 2013: Updated downloads to AvalonEdit 4.3.0.9390 (SharpDevelop 4.3)
    - Added support for Input Method Editors (IME)
    - Fixed a major bug that sometimes caused "InvalidOperationException: Trying to build visual line from collapsed line" when updating existing foldings.
- April 2, 2013: Updates downloads to AvalonEdit 4.3.1.9430 (SharpDevelop 4.3.1)
    - Fixed a bug in IME support - the previous version did not properly re-enable the IME if it was disabled by another WPF control.

Note: Although my sample code is provided under the MIT license, `ICSharpCode.AvalonEdit` itself is provided under the terms of the GNU LGPL.

License

This article, along with any associated source code and files, is licensed under The GNU Lesser General Public License (LGPLv3)


by Daniel Grunwald
Contributor

2016k Views
4.95/ 5

Comments And Discussions (419)

Sort: Newest

Share your thoughts

You need to be signed in to participate in the discussion. Sign in

xiaoxstz
Jul 24, 2023

It's useful