TextDocument

TextEditor.Document  
TextArea.Document  
TextView.Document

public sealed class TextDocument : ITextSource

{

public event EventHandler UpdateStarted;

public event EventHandler<DocumentChangeEventArgs> Changing;

public event EventHandler<DocumentChangeEventArgs> Changed;

public event EventHandler TextChanged;

public event EventHandler UpdateFinished;

public TextAnchor CreateAnchor(int offset);

public ITextSource CreateSnapshot();

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 BeginUpdate();

public bool IsInUpdate { get; }

public void EndUpdate();

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; }

}

public interface ISegment

{

int Offset { get; }

int Length { get; } // must be non-negative

int EndOffset { get; } // must return Offset+Length

}

public struct TextLocation

{

public int Line { get; }

public int Column { get; }

}

public sealed partial class DocumentLine : ISegment

{

public bool IsDeleted { get; }

public int LineNumber { get; }

public int Offset { get; }

public int EndOffset { get; }

public int Length { get; }

/// Gets the length of this line, including the line delimiter

public int TotalLength { get; }

public DocumentLine NextLine { get; }

public DocumentLine PreviousLine { get; }

}

**BeginUpdate()**

* UpdateStarted event is raised

**Insert() / Remove() / Replace()**

* Changing event is raised
* The document is changed
* TextAnchor.Deleted events are raised if anchors were in the deleted text portion
* Changed event is raised

**EndUpdate()**

* TextChanged event is raised
* TextLengthChanged event is raised
* LineCountChanged event is raised
* UpdateFinished event is raised
* TextAnchor anchor = document.CreateAnchor(offset);
* ChangeMyDocument();
* int newOffset = anchor.Offset;

AnchorSegment class which implements ISegment using two text anchors

The TextSegmentCollection will automatically update the offsets of all TextSegment instances inside the collection; but it also has the useful methods FindOverlappingSegments and FindFirstSegmentWithStartAfter.

CreateSnapshot() returns an immutable snapshot of the document, and may be safely called even when the owner thread is concurrently modifying the document. This is very useful for features like a background parser that is running on its own thread.

The overload CreateSnapshot(out ChangeTrackingCheckpoint) also returns a ChangeTrackingCheckpoint for the document snapshot. Once you have two checkpoints, you can call GetChangesTo to retrieve the complete list of document changes that happened between those versions of the document.

Changed(this, DocumentChangeEventArgs);

public class DocumentChangeEventArgs : EventArgs

{

public int Offset { get}

public string RemovedText { get; }

public int RemovalLength { }

public string InsertedText { get; }

public int InsertionLength {

public OffsetChangeMap OffsetChangeMap {

public int GetNewOffset(int offset, AnchorMovementType movementType)

}

public class OffsetChangeMap : Collection<OffsetChangeMapEntry>

public struct OffsetChangeMapEntry : IEquatable<OffsetChangeMapEntry>

{

/// The offset at which the change occurs.

public int Offset {

/// The number of characters inserted.

/// Returns 0 if this entry represents a removal.

public int InsertionLength {

/// The number of characters removed.

/// Returns 0 if this entry represents an insertion.

public int RemovalLength {

SAMPLE :

foreach(DocumentChangeEventArgs change in changes) {

// Update offsets of all items

segments.UpdateOffsets(change);

// Removing will cause one of the ends to be set to change.Offset

// FindSegmentsContaining includes any segments touching

// so that conviniently takes care of the +1 byte

var segmentsContainingOffset = segments.FindOverlappingSegments(change.Offset, change.InsertionLength);

foreach(AXmlObject obj in segmentsContainingOffset.OfType<AXmlObject>().Where(o => o.IsCached)) {

InvalidateCache(obj, false);

TextView

public class TextView

{  
public IList<VisualLineElementGenerator> ElementGenerators {  
public IList<IVisualLineTransformer> LineTransformers {  
public ReadOnlyCollection<VisualLine> VisualLines {  
public event E<VisualLineConstructionStartEventArgs> VisualLineConstructionStarting  
public event EventHandler VisualLinesChanged;

public IList<IBackgroundRenderer> BackgroundRenderers {

before rendering the TextView starts to build visual lines and also immediately does the conversion to TextLine (word-wrapping).

The TextView.Redraw methods are used to remove visual lines from the cache.

AvalonEdit will redraw automatically on the affected lines when the document is changed

Calling Redraw does not cause immediate recreation of the lines. They are just removed from the cache so that the next rendering step will recreate them. All redraw methods will enqueue a new rendering step, using the WPF Dispatcher with a low priority.

Visual lines are created only for the visible part of the document.

public sealed class VisualLine

{

public TextDocument Document { get; private set; }

public DocumentLine FirstDocumentLine { get; private set; }

public DocumentLine LastDocumentLine { get; private set; }

public ReadOnlyCollection<VisualLineElement> Elements { get; private set;

public ReadOnlyCollection<TextLine> TextLines

public abstract class VisualLineElement

{

public int VisualLength { get; private set; }

public int DocumentLength { get; private set; }

public int VisualColumn { get; internal set; }

public int RelativeTextOffset { get; internal set; }

public VisualLineElementTextRunProperties TextRunProperties { get; public Brush BackgroundBrush { get; set; }

public abstract TextRun CreateTextRun(int startVisualColumn,

public virtual bool CanSplit {

public virtual void Split(int splitVisualColumn,

public virtual bool IsWhitespace(int visualColumn)

protected internal virtual void OnMouseDown(MouseButtonEventArgs e)

protected internal virtual void OnMouseUp(MouseButtonEventArgs e)

The main job of a visual line element is to implement the CreateTextRun method. This method should return a System.Windows.Media.TextFormatting.TextRun instance that can be rendered using the TextLine class.

DocumentLine

[Element Generator] => VisualLine -> VisualLineElement created (folding manager)

[Line Transformer] => VisualLineElement split and color (syntax highlighting)

[WPF TextFormatter] => WPF TextLine -> TextRun (word wrap)

[Backgd Renderer] => ? (current selection)

### Element Generators

You can extend the text view by registering a custom class deriving from VisualLineElementGenerator in the TextView.ElementGenerators collection. This allows you to add custom VisualLineElements.

Using the InlineObjectElement class, you can even put interactive WPF controls (anything derived from UIElement) into the text document.

For all document text not consumed by element generators, AvalonEdit will create VisualLineText elements.

### Line Transformers

Line transformers can modify the visual lines after they have been generated. The main usage of this is to colorize the text, as done both by syntax highlighting and the selection.

The base classes ColorizingTransformer and DocumentColorizingTransformer help with this task by providing helper methods for colorizing that split up visual line elements where necessary. The difference between the two classes is that one works using visual columns whereas the other one uses offsets into the document.

### Background renderers

Background renderers are simple objects that allow you to draw anything in the text view. They can be used to draw nice-looking backgrounds behind the text.

AvalonEdit contains the class BackgroundGeometryBuilder that helps with this task. You can use the static BackgroundGeometryBuilder.GetRectsForSegment to fetch a list of rectangles that contain text from the specified segment (you will get one rectangle per TextLine); or you can use the instance methods to build a PathGeometry for the text's outline. AvalonEdit also internally uses this geometry builder to create the selection with the rounded corners.

Inside SharpDevelop, the first option (getting list of rectangles) is used to render the squiggly red line that for compiler errors, while the second option is used to produce nice-looking breakpoint markers.

public enum KnownLayer

{

/// <summary>

/// This layer is in the background.

/// There is no UIElement to represent this layer, it is directly drawn in the TextView.

/// It is not possible to replace the background layer or insert new layers below it.

/// </summary>

/// <remarks>This layer is below the Selection layer.</remarks>

Background,

/// <summary>

/// This layer contains the selection rectangle.

/// </summary>

/// <remarks>This layer is between the Background and the Text layers.</remarks>

Selection,

/// <summary>

/// This layer contains the text and inline UI elements.

/// </summary>

/// <remarks>This layer is between the Selection and the Caret layers.</remarks>

Text,

/// <summary>

/// This layer contains the blinking caret.

/// </summary>

/// <remarks>This layer is above the Text layer.</remarks>

Caret

}

public interface IBackgroundRenderer

{

/// <summary>

/// Gets the layer on which this background renderer should draw.

/// </summary>

KnownLayer Layer { get; }

/// <summary>

/// Causes the background renderer to draw.

/// </summary>

void Draw(TextView textView, DrawingContext drawingContext);

SAMPLE

public void Draw(TextView textView, DrawingContext drawingContext)

{

int viewStart = visualLines.First().FirstDocumentLine.Offset;

int viewEnd = visualLines.Last().LastDocumentLine.EndOffset;

foreach (SearchResult result in currentResults.FindOverlappingSegments(viewStart, viewEnd - viewStart)) {

BackgroundGeometryBuilder geoBuilder = new BackgroundGeometryBuilder();

geoBuilder.AlignToMiddleOfPixels = true;

geoBuilder.CornerRadius = 3;

geoBuilder.AddSegment(textView, result);

Geometry geometry = geoBuilder.CreateGeometry();

if (geometry != null) {

drawingContext.DrawGeometry(markerBrush, markerPen, geometry);

}