
ComponentOne

RichTextBox for Silverlight

Copyright © 1987-2009 ComponentOne LLC. All rights reserved.

Corporate Headquarters
ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com
Web site: <http://www.componentone.com>

Sales

E-mail: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

ComponentOne RichTextBox for Silverlight and the ComponentOne RichTextBox for Silverlight logo are trademarks of ComponentOne LLC. ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

Warranty

ComponentOne warrants that the original media is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no cost to you. After 90 days, you may obtain a replacement for a defective media by sending your request and a check for \$25 (to cover postage and handling) to ComponentOne at the above address.

Except for the express warranty of the original media set forth herein, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was made public. ComponentOne does not warrant and therefore shall not be liable for any errors or omissions that the software documentation may contain. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not and shall not be held liable for any special, punitive, incidental, consequential, or any other damages that may result from your use of the software.

Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of the software by anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.



Table of Contents

ComponentOne RichTextBox for Silverlight	1
Introduction to C1.Silverlight.RichTextBox	1
C1RichTextBox Concepts and Main Properties	1
Saving and Loading HTML	2
Implementing a Simple Formatting Toolbar	3
Hyperlinks	5
Spell-Checking	7
Modal Spell-Checking	7
Syntax Coloring	9
Overriding Styles	12
Hit-Testing	14
Working with the C1Document Object	15
Creating Documents and Reports	16
Implementing Split Views	21
Using the C1Document class	22
Understanding C1TextPointer	23

ComponentOne RichTextBox for Silverlight

The following topics describe how you can use the **C1.Silverlight.RichTextBox** assembly to implement rich formatting and editing to your Silverlight applications as well as create and manage documents directly.

Introduction to C1.Silverlight.RichTextBox

The **C1.Silverlight.RichTextBox** assembly contains two main objects: the **C1RichTextBox** control and the **C1Document** object.

C1RichTextBox is a powerful text editor that allows you to display and edit formatted text. **C1RichTextBox** supports all the usual formatting options, including fonts, background and foreground colors, lists, hyperlinks, images, borders, etc. **C1RichTextBox** also supports loading and saving documents in HTML format.

C1Document is the class that represents the contents of a **C1RichTextBox**. It is analogous to the **FlowDocument** class in WPF. As in WPF, a **C1Document** is composed of stacked elements (**C1Block** objects) which in turn are composed of inline elements (**C1Run** objects).

Many applications may deal only with the **C1RichTextBox** control, which provides a simple linear view of the document. Other applications may choose to use the rich object model provided by the **C1Document** class to create and manage documents directly, with full access to the document structure.

This document introduces the **C1RichTextBox** control by discussing basic concepts and showing how you can use the control to create different types of applications. It also discusses how you can use related components such as the **C1RichTextToolbar** and **C1SpellChecker**.

C1RichTextBox Concepts and Main Properties

On the surface, the **C1RichTextBox** is just like a regular **TextBox**. It provides the same properties to control the font, colors, text, and selection. If you have an application that uses **TextBox** controls, it is likely you can simply replace them with **C1RichTextBox** controls and it will work without any additional changes.

For example, the code implements a simple search and replace routine that works on **TextBox** and on **C1RichTextBox** controls:

```
void SearchAndReplace(TextBox tb, string find, string replace)
{
    for (int start = 0; ; )
    {
        int pos = tb.Text.IndexOf(find, start);
        if (pos < 0) break;
        tb.Select(pos, find.Length);
        // optionally show dialog box to confirm the change
        tb.SelectedText = replace;
        start = pos + 1;
    }
}
```

The code looks for matches in the **Text** property. It selects each match using the **Select** method, and then replaces the text using the **SelectedText** property. To convert this method for use with the **C1RichTextBox** control, you would simply change the type of the first argument to use a **C1RichTextBox** instead of a regular **TextBox**.

This is what the **C1RichTextBox** has in common with the regular **TextBox**. But of course it goes way beyond that. Suppose you wanted to highlight the replacements with a yellow background. This would be impossible with a regular **TextBox**. With the **C1RichTextBox**, you could accomplish that with one additional line of code:

```
void SearchAndReplace(C1RichTextBox tb, string find, string replace)
{
    for (int start = 0; ; )
    {
        int pos = tb.Text.IndexOf(find, start);
        if (pos < 0) break;
        tb.Select(pos, find.Length);
        // optionally show dialog to confirm the change
        tb.Selection.InlineBackground = new SolidColorBrush(Colors.Yellow);
        tb.SelectedText = replace;
        start = pos + 1;
    }
}
```

The **Selection** property provides properties that allow you to inspect and modify the formatting of the current selection. With this property and the ones in common with the **TextBox** control, you can easily create documents and add rich formatting.

You could use the technique described above to implement a toolbar or to add syntax coloring to documents. These topics are described in more detail in later sections.

Saving and Loading HTML

You can persist the contents of a simple **TextBox** control using the **Text** property. That also works with the **C1RichTextBox**, except any rich formatting would be lost. Instead, you can use the **Html** property to persist the content of a **C1RichTextBox** while preserving the formatting.

The **Html** property gets or sets the formatted content of a **C1RichTextBox** as an html string. The html filter built into the **C1RichTextBox** is fairly rich. It supports CSS styles, images, hyperlinks, lists, etc. But the filter does not support all of html; it is limited to features supported by the **C1RichTextBox** control itself. For example, the current version of **C1RichTextBox** does not support tables. Still, you can use the **Html** property to display simple html documents.

If you type “Hello world” into a **C1RichTextBox**, the **Html** property will return the following:

```
<html>
<head>
  <style type="text/css">
    .c0 { font-family:Portable User Interface;font-size:9pt; }
    .c1 { margin-bottom:7.5pt; }
  </style>
</head>
<body class="c0">
<p class="c1">Hello world.</p>
</body>
</html>
```

Note that the `Html` property is just a filter between `html` and the internal `C1Document` class. Any information in the `html` stream that is not supported by the `C1RichTextBox` (for example, comments and meta information) is discarded, and will not be preserved when you save the `html` document later.

Implementing a Simple Formatting Toolbar

Most rich editors include a toolbar with buttons that format the current selection, making it bold, italic, or underlined. The buttons also change state when you move the selection, to show that the selected text is bold, italic, underlined, etc.

Implementing a simple toolbar with the `C1RichTextBox` is easy. For example, the Xaml below defines three buttons (bold, italic, underline) and a `C1RichTextBox` that they control:

```
<UserControl x:Class="C1RichTextBoxIntro.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:clr="clr-
namespace:C1.Silverlight.RichTextBox;assembly=C1.Silverlight.RichTextBox">
  <Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <StackPanel Orientation="Horizontal" >
      <ToggleButton x:Name="_btnBold" Content="B" Click="_btnBold_Click" />
      <ToggleButton x:Name="_btnItalic" Content="I" Click="_btnItalic_Click"
/>
      <ToggleButton x:Name="_btnUnderline" Content="U"
Click="_btnUnderline_Click" />
    </StackPanel>
    <clr:C1RichTextBox x:Name="_rtb" Grid.Row="1"
      AcceptsReturn="True"
      SelectionChanged="_rtb_SelectionChanged"/>
  </Grid>
</UserControl>
```

When the buttons are clicked, the attached event handlers are responsible for updating the format of the selection. Here is the code that accomplishes that:

```
private void _btnBold_Click(object sender, RoutedEventArgs e)
{
  FontWeight? fw = _rtb.Selection.FontWeight;
  _rtb.Selection.FontWeight = fw.HasValue && fw.Value == FontWeights.Bold
    ? FontWeights.Normal
    : FontWeights.Bold;
}
```

The code starts by getting the value of the `FontWeight` property for the current selection. Note that the value returned is nullable (hence the `?` in the type declaration). If the selection contains a mix of different font weights, the value returned is null. The code above sets the font weight to “normal” if the whole selection has a single font weight and it is bold; otherwise, the code sets the font weight to “bold”.

The code that handles the italics button is very similar, except it uses the **FontStyle** property instead of **FontWeight**:

```
private void _btnItalic_Click(object sender, RoutedEventArgs e)
{
    FontStyle? fs = _rtb.Selection.FontStyle;
    _rtb.Selection.FontStyle = fs.HasValue && fs.Value == FontStyles.Italic
        ? FontStyles.Normal
        : FontStyles.Italic;
}
```

Finally, the code that handles the underline button is similar, this time using the **TextDecorations** property. Note that **TextDecorations** property returns an actual object, and thus is not a nullable property:

```
private void _btnUnderline_Click(object sender, RoutedEventArgs e)
{
    bool underline = _rtb.Selection.TextDecorations ==
TextDecorations.Underline;
    _rtb.Selection.TextDecorations = underline
        ? null
        : TextDecorations.Underline;
}
```

This is all it takes to make the three buttons work.

Next, we need to implement the event handler for the SelectionChanged event. This event handler is responsible for changing the state of the buttons as the user moves the selection. For example, selecting a word that is bold and underlined would make the buttons appear pressed. Here is the code:

```
void _rtb_SelectionChanged(object sender, EventArgs e)
{
    FontWeight? fw = _rtb.Selection.FontWeight;
    _btnBold.IsChecked = fw.HasValue && fw.Value == FontWeights.Bold;

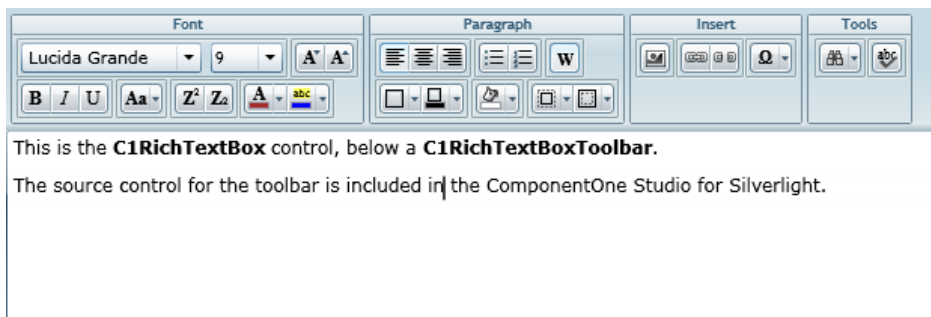
    FontStyle? fs = _rtb.Selection.FontStyle;
    _btnItalic.IsChecked = fs.HasValue && fs.Value == FontStyles.Italic;

    bool underline = _rtb.Selection.TextDecorations ==
TextDecorations.Underline;
    _btnUnderline.IsChecked = underline;
}
```

The code uses the **FontWeight**, **FontStyle**, and **TextDecorations** properties as before, and sets the **IsChecked** property on the corresponding buttons.

A full toolbar would include more buttons and controls, and would handle them in a similar fashion.

The ComponentOne Studio for Silverlight package includes a complete toolbox as a separate assembly. The source code for the **C1RichTextBoxToolbar** control is included so you can create your own customized version. The image below shows the **C1RichTextBoxToolbar** in action:



Hyperlinks

The **C1RichTextBox** supports hyperlinks. As in regular html documents, this feature allows you to make certain parts of the document active. When the user clicks them, the application receives a notification and takes some action.

The code below shows how you can create a hyperlink:

```
void MakeHyperlink()
{
    // set text
    _rtb.Text = "This is some text with a hyperlink in it.";

    // create hyperlink
    int pos = _rtb.Text.IndexOf("hyperlink");
    _rtb.Select(pos, 9);
    var uri = new Uri("http://www.componentone.com", UriKind.Absolute);
    _rtb.Selection.MakeHyperlink(uri);

    // handle navigation requests
    _rtb.NavigationMode = NavigationMode.OnControlKey;
    _rtb.RequestNavigate += _rtb_RequestNavigate;
}
```

The code starts by assigning some text to the **C1RichTextBox**. Next, it selects the word “hyperlink” and calls the **MakeHyperlink** method to make it a hyperlink. The parameter is a URI that is assigned to the new hyperlink's **NavigateUri** property.

Then, the code sets the **NavigationMode** property to determine how the **C1RichTextBox** should handle the mouse over hyperlinks. The default behavior is like the one in Microsoft Word and Visual Studio: moving the mouse over a hyperlink while holding down the control key causes the cursor to turn into a hand, and clicking while the control key is pressed fires the **RequestNavigate** event. This allows users to edit the hyperlink text as they would edit regular text.

The **RequestNavigate** event handler is responsible for handling the hyperlink navigation. In many cases this requires opening a new browser window and navigating to a different URL. This is illustrated below:

```
void _rtb_RequestNavigate(object sender, RequestNavigateEventArgs e)
{
    // open link in a new window ("_self" would use the current one)
    string target = "_blank";
    System.Windows.Browser.HtmlPage.Window.Navigate(e.Hyperlink.NavigateUri,
    target);
}
```

Note that hyperlink actions are not restricted to **URI** navigation. You could define a set of custom URI actions to be used as commands within your application. The custom URIs would be parsed and handled by the **RequestNavigate** handler. For example, the code below uses hyperlinks to show message boxes:

```
void MakeHyperlink()
{
    // set text
    _rtb.Text = "This is some text with a hyperlink in it.";

    // create hyperlink
    int pos = _rtb.Text.IndexOf("hyperlink");
    _rtb.Select(pos, 9);
    var uri = new Uri("msgbox:thanks for clicking!");
    _rtb.Selection.MakeHyperlink(uri);

    // handle navigation requests
    _rtb.NavigationMode = NavigationMode.OnControlKey;
    _rtb.RequestNavigate += _rtb_RequestNavigate;
}

void _rtb_RequestNavigate(object sender, RequestNavigateEventArgs e)
{
    Uri uri = e.Hyperlink.NavigateUri;
    if (uri.Scheme == "msgbox")
    {
        MessageBox.Show(uri.LocalPath);
    }
}
```

The only change in the **MakeHyperlink** code is the line that creates the URI. The **RequestNavigate** handler uses the URI members to parse the command and argument. You could use this technique to create documents with embedded menus for example.

Note that the **CreateHyperlink** method is just a quick and easy way to turn an existing part of a document into a hyperlink. You can also create hyperlinks by adding **C1Hyperlink** elements to **C1Document** objects. This is described in later sections.

Spell-Checking

Most rich editors implement two types of spell-checking:

- **Modal spell checking:** Shows a spell-dialog and selects each spelling mistake in the document. The user may choose to ignore the mistake, fix it by typing or picking from a list of suggestions, or add the word to a user dictionary.
- **As-you-type checking:** Highlights spelling mistakes as the user types, typically with a wavy red underline. The user may right-click the mistake in the document to see a menu with options that include ignore, add to dictionary, or pick a suggestion to correct the mistake automatically.

The **C1RichTextBox** supports both types of spell-checking using the **C1SpellChecker** component, also included in the ComponentOne Studio for Silverlight. The **C1SpellChecker** ships as a separate assembly because it can spell-check other controls as well.

Modal Spell-Checking

To implement modal spell checking, you start by adding to your project a reference to the **C1.Silverlight.SpellChecker** assembly. Then, add the following code to your project:

```
using C1.Silverlight.SpellChecker;

public partial class Page : UserControl
{
    // spell-checker used by all controls on this page
    C1SpellChecker _spell = new C1SpellChecker();

    // page constructor
    public Page()
    {
        // regular initialization
        InitializeComponent();

        // load main spell dictionary
        _spell.MainDictionary.LoadCompleted += MainDictionary_LoadCompleted;
        _spell.MainDictionary.LoadAsync("C1Spell_en-US.dct");

        // load user dictionary
        UserDictionary ud = _spell.UserDictionary;
        ud.LoadFromIsolatedStorage("Custom.dct");
        App.Current.Exit += App_Exit;

        // other initialization
        // ...
    }
}
```

The code creates a new **C1SpellChecker** object to be shared by all controls on the page that require spell-checking.

Later, the page constructor invokes the **LoadAsync** method to load the main spell dictionary. In this case, we are loading **C1Spell_en-US.dct**, the American English dictionary. This file must be present on the application folder on the server. **C1SpellChecker** includes over 20 other dictionaries which can be downloaded from our site.

The code adds a handler to the **LoadCompleted** event so it can detect when the main dictionary finishes loading and whether there were any errors. Here is a typical event handler:

```
void MainDictionary_LoadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (e.Error != null)
        MessageBox.Show("Error loading spell dictionary," +
            "spell-checking is disabled.");
}
```

The code also loads a user dictionary from isolated storage. This step is optional. The user dictionary stores words such as names and technical terms. The code attaches an event handler to the application's **Exit** event to save the user dictionary when the application finishes executing:

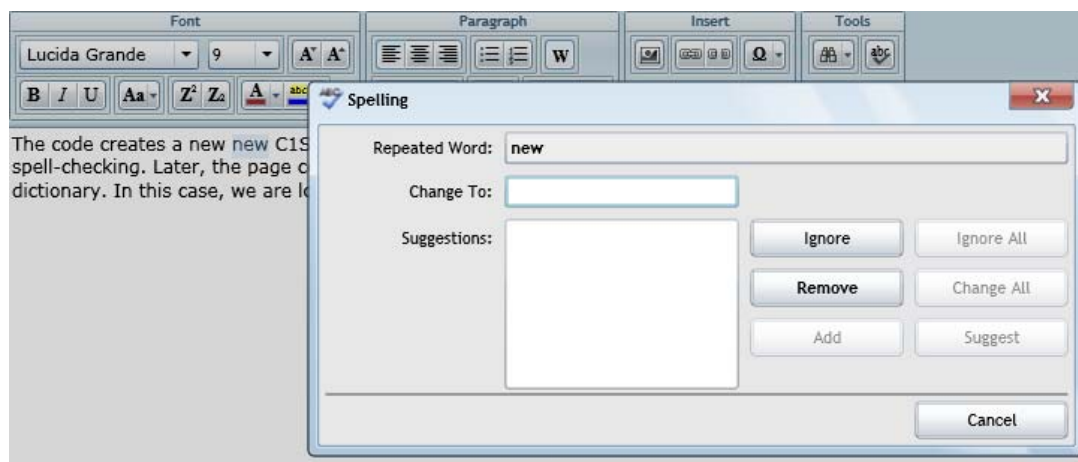
```
void App_Exit(object sender, EventArgs e)
{
    UserDictionary ud = _spell.UserDictionary;
    ud.SaveToIsolatedStorage("Custom.dct");
}
```

Once the dictionary has been loaded, you can invoke the modal spell-checker by calling the **CheckControlAsync** method. For example:

```
private void SpellCheck_Click(object sender, RoutedEventArgs e)
{
    _spell.CheckControlCompleted += _spell_CheckControlCompleted;
    _spell.CheckControlAsync(_rtb);
}
void _spell_CheckControlCompleted(object sender,
CheckControlCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        var msg = string.Format(
            "Spell-check complete. {0} error(s) found.", e.ErrorCount);
        MessageBox.Show(msg, "Spell-check complete", MessageBoxButton.OK);
    }
}
```

The code calls **CheckControlAsync**. When the modal checking is complete, the **CheckControlCompleted** event fires and shows a dialog to indicate that the spell-checking operation is complete.

The image below shows the spell-checking dialog box in action:



Syntax Coloring

An earlier section of this document described how you can use the **Selection** property to obtain a **C1TextRange** object that corresponds to the current selection, and how to use that object to inspect and apply custom formatting to parts of the document.

In some cases, however, you may want to inspect and apply formatting to ranges without selecting them. To do that using the **Selection** property, you would have to save the current selection, apply all the formatting, and then restore the original selection. Also, changing the selection may cause the document to scroll in order to keep the selection in view.

To handle these situations, the **C1RichTextBox** exposes a **GetTextRange** method. The **GetTextRange** method returns a **C1TextRange** object that may be used without affecting the current selection.

For example, you could use the **GetTextRange** method to add HTML syntax coloring to a **C1RichTextBox**. The first step is to detect any changes to the document. The changes will trigger the method that performs the actual syntax coloring:

```
// update syntax coloring on a timer
bool _updating;
Storyboard _syntax;

// start the timer whenever the document changes
void tb_TextChanged(object sender, C1TextChangedEventArgs e)
{
    if (!_updating)
    {
        // create storyboard if it's still null
        if (_syntax == null)
        {
            _syntax = new Storyboard();
            _syntax.Completed += _syntax_Completed;
            _syntax.Duration = new Duration(TimeSpan.FromMilliseconds(1000));
        }
    }
}
```

```

    // re-start storyboard
    _syntax.Stop();
    _syntax.Seek(TimeSpan.Zero);
    _syntax.Begin();
}
}
// timer elapsed, update syntax coloring
void _syntax_Completed(object sender, EventArgs e)
{
    _updating = true;
    UpdateSyntaxColoring(_rtb);
    _updating = false;
}

```

The code creates a timer that starts ticking whenever the user changes the document in any way. If the user changes the document while the timer is active, then the timer is reset. This prevents the code from updating the syntax coloring too often, while the user is typing quickly.

When the timer elapses, the code sets a flag to prevent the changes made while updating the syntax coloring from triggering the timer, then calls the **UpdateSyntaxColoring** method:

```

// perform syntax coloring
void UpdateSyntaxColoring(C1RichTextBox rtb)
{
    // initialize regular expression used to parse HTML
    string pattern =
        @"</?(?<tagName>[a-zA-Z0-9_:\-]+)" +
        @"(\s+(?<attName>[a-zA-Z0-9_:\-]+)(?<attValue>(=""[^"]+"")?)*)\s*/?>";

    // initialize brushes used to color the document
    Brush brDarkBlue = new SolidColorBrush(Color.FromArgb(255, 0, 0, 180));
    Brush brDarkRed = new SolidColorBrush(Color.FromArgb(255, 180, 0, 0));
    Brush brLightRed = new SolidColorBrush(Colors.Red);

    // remove old coloring
    var input = rtb.Text;
    var range = rtb.GetTextRange(0, input.Length);
    range.Foreground = rtb.Foreground;

    // highlight the matches
    foreach (Match m in Regex.Matches(input, pattern))
    {
        // select whole tag, make it dark blue
        range = rtb.GetTextRange(m.Index, m.Length);
        range.Foreground = brDarkBlue;

        // select tag name, make it dark red
        var tagName = m.Groups["tagName"];
        range = rtb.GetTextRange(tagName.Index, tagName.Length);
        range.Foreground = brDarkRed;
    }
}

```

```

// select attribute names, make them light red
var attGroup = m.Groups["attName"];
if (attGroup != null)
{
    var atts = attGroup.Captures;
    for (int i = 0; i < atts.Count; i++)
    {
        var att = atts[i];
        range = rtb.GetTextRange(att.Index, att.Length);
        range.Foreground = brLightRed;
    }
}
}
}
}

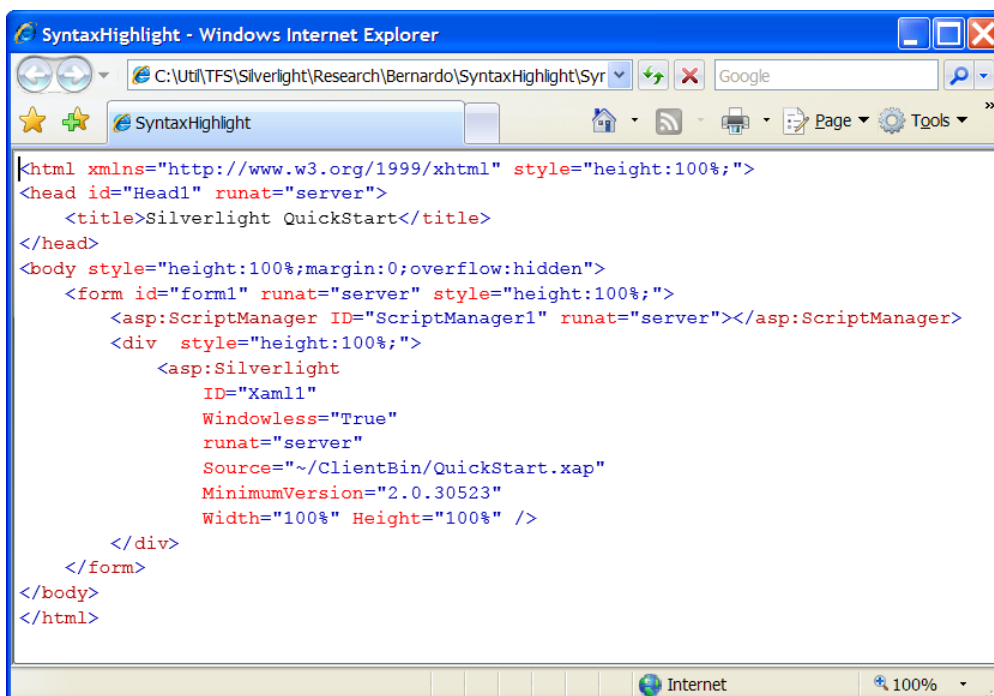
```

The code starts by defining a regular expression pattern to use for parsing the HTML. This is not the most efficient way to parse HTML, and the expression is not terribly easy to read or maintain. We don't recommend using regular expressions for parsing HTML except in sample code, where its compactness helps keep the code compact and easy to understand.

The next step is to remove any old coloring left over. This is done by creating a range that spans the whole document and setting its **Foreground** property to match the **Foreground** of the **C1RichTextBox** control.

Next, the regular expression is used to parse the document. The code scans each match, creates a **C1TextRange** object, and sets the **Foreground** property to the desired value. We use dark blue for the HTML tag, dark red for the tag name, and light red for the attribute names.

That's all the code that is required. The image below shows an HTML document viewed in the syntax-coloring **C1RichTextBox** we just created:



Test the application by typing or pasting some HTML text into the control. Notice that shortly after you stop typing, the new text is colored automatically.

A real application could optimize the syntax coloring process by detecting the type of text change and updating the coloring of small parts of the document. Also, it would detect additional elements such as style sheets and comments, and it probably would use a specialized parser instead of regular expressions.

The essential mechanism would be the same, however: detect ranges within the document, get **C1TextRange** objects, and apply the formatting.

Overriding Styles

The previous sections described how you can use **C1TextRange** objects to modify the style of parts of a document without moving the selection. In some cases, however, you may want to modify only the view, and not the document itself.

For example, the current selection is highlighted with different foreground and background colors. This style change does not belong to the document itself; it belongs to the view. Other examples are syntax coloring and as-you-type spell-checking.

The **C1RichTextBox** control supports these scenarios with the **StyleOverrides** property. This property contains a collection of objects that specify ranges and style modifications to be applied to the view only. This approach has two advantages over applying style modifications to **C1TextRange** objects as we did in the previous section:

1. The style overrides are not applied to the document, and therefore are not applied when you save a document as HTML (you would not normally want the current selection and spelling error indicators to be persisted to a file).
2. Because the changes are not added to the document, and only affect the part that is currently visible, this approach is much more efficient than changing **C1TextRange** objects directly.

The limitation of this approach is that the style changes cannot involve style elements that affect the document flow. You can use style overrides to change the background, foreground, and to underline parts of the document. But you cannot change the font size or style, for example, since that would affect the document flow.

Let us demonstrate the use of style overrides by modifying the previous syntax coloring example.

First, we need to declare a **C1RangeStyleCollection** object and add that to the control's **StyleOverrides** collection. Once that is done, any overrides added to our collection will be applied to the control. We will later populate the collection with the syntax-colored parts of the document.

```
C1RangeStyleCollection _rangeStyles = new C1RangeStyleCollection();
```

```
public Page()
{
    InitializeComponent();

    _rtb = new C1RichTextBox();
    LayoutRoot.Children.Add(_rtb);

    _rtb.TextChanged += tb_TextChanged;
    _rtb.FontFamily = new FontFamily("Courier New");
    _rtb.FontSize = 16;
    _rtb.Text = GetStringResource("w3c.htm");

    // add our C1RangeStyleCollection to the control's
    // StyleOverrides collection
    _rtb.StyleOverrides.Add(_rangeStyles);
}
```

Now, all we need to do is modify the **UpdateSyntaxColoring** method shown earlier and have it populate our collection of range styles (instead of applying the coloring to the document as we did before):

```
// perform syntax coloring using StyleOverrides collection
// (takes a fraction of a second to highlight the default document)
void UpdateSyntaxColoring(C1RichTextBox rtb)
{
    // initialize regular expression used to parse HTML
    string pattern =
        @"</?(?<tagName>[a-zA-Z0-9_:\-]+)" +
        @"(\s+(?<attName>[a-zA-Z0-9_:\-]+)" +
        "(?<attValue>(\s*=\s*" + "[^"]*" + ")?))*\s*/?>";

    // initialize styles used to color the document
    var key = C1TextElement.ForegroundProperty;
    var brDarkBlue = new C1TextElementStyle();
    brDarkBlue[key] = new SolidColorBrush(Color.FromArgb(255, 0, 0, 180));
    var brDarkRed = new C1TextElementStyle();
    brDarkRed[key] = new SolidColorBrush(Color.FromArgb(255, 180, 0, 0));
    var brLightRed = new C1TextElementStyle();
    brLightRed[key] = new SolidColorBrush(Colors.Red);

    // remove old coloring
    _rangeStyles.Clear();

    // highlight the matches
    var input = rtb.Text;
    foreach (Match m in Regex.Matches(input, pattern))
    {
        // select whole tag, make it dark blue
        var range = rtb.GetTextRange(m.Index, m.Length);
        _rangeStyles.Add(new C1RangeStyle(range, brDarkBlue));

        // select tag name, make it dark red
        var tagName = m.Groups["tagName"];
        range = rtb.GetTextRange(tagName.Index, tagName.Length);
        _rangeStyles.Add(new C1RangeStyle(range, brDarkRed));

        // select attribute names, make them light red
        var attGroup = m.Groups["attName"];
        if (attGroup != null)
        {
            foreach (Capture att in attGroup.Captures)
            {
                range = rtb.GetTextRange(att.Index, att.Length);
                _rangeStyles.Add(new C1RangeStyle(range, brLightRed));
            }
        }
    }
}
```

The revised code is very similar to the original. Instead of creating brushes to color the document, it creates **C1TextElementStyle** objects that contain an override for the foreground property. The code starts by clearing the override collection, then uses a regular expression to locate each HTML tag in the document, and finally

populates the overrides collection with **C1RangeStyle** objects that associate ranges with **C1TextElementStyle** objects.

If you run this new version of the code, you should notice the dramatic performance increase. The new version is thousands of times faster than the original.

Hit-Testing

The **C1RichTextBox** supports hyperlinks, which provide a standard mechanism for implementing user interactivity. In some cases, you may want to go beyond that and provide additional, custom mouse interactions. For example, you may want to apply some custom formatting or show a context menu when the user clicks an element.

To enable these scenarios, the **C1RichTextBox** exposes **ElementMouse*** events and a **GetPositionFromPoint** method.

If all you need to know is the element that triggered the mouse event, you can get it from the **source** parameter in the event handler. If you need more detailed information (the specific word that was clicked within the element for example), then you need the **GetPositionFromPoint** method. **GetPositionFromPoint** takes a point in client coordinates and returns a **C1TextPosition** object that expresses the position in document coordinates.

The **C1TextPosition** object has two main properties: **Element** and **Offset**. The **Element** property represents an element within the document; **Offset** is a character index (if the element is a **C1Run**) or the index of the child element at the given point.

For example, the code below creates a **C1RichTextBox** and attaches a handler to the **ElementMouseDownLeftButtonDown** event:

```
public Page()
{
    // default initialization
    InitializeComponent();

    // create a C1RichTextBox and add it to the page
    _rtb = new C1RichTextBox();
    LayoutRoot.Children.Add(_rtb);

    // attach event handler
    _rtb.ElementMouseDownLeftButtonDown += rtb_ElementMouseDownLeftButtonDown;
}
```

The event handler below toggles the **FontWeight** property for the entire element that was clicked. This could be a word, a sentence, or a whole paragraph:

```
void _rtb_ElementMouseDownLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (Keyboard.Modifiers != 0)
    {
        var run = sender as C1Run;
        if (run != null)
        {
            run.FontWeight = run.FontWeight == FontWeights.Bold
                ? FontWeights.Normal
                : FontWeights.Bold;
        }
    }
}
```

The code gets the element that was clicked by casting the **sender** parameter to a **C1Run** object.

If you wanted to toggle the **FontWeight** value of a single word instead, then you would need to determine which character was clicked and expand the selection to the whole word. This is where the **GetPositionFromPoint** method becomes necessary. Here is a revised version of the event handler that accomplishes that:

```
void _rtb_ElementMouseDown(object sender, MouseButtonEventArgs e)
{
    if (Keyboard.Modifiers != 0)
    {
        // get position in control coordinates
        var pt = e.GetPosition(_rtb);

        // get text pointer at position
        var pointer = _rtb.GetPositionFromPoint(pt);

        // check that the pointer is pointing to a C1Run
        var run = pointer.Element as C1Run;
        if (run != null)
        {
            // get the word within the C1Run
            var text = run.Text;
            var start = pointer.Offset;
            var end = pointer.Offset;
            while (start > 0 && char.IsLetterOrDigit(text, start - 1))
                start--;
            while (end < text.Length - 1 && char.IsLetterOrDigit(text, end + 1))
                end++;

            // toggle the bold property for the run that was clicked
            var word = new C1TextRange(pointer.Element, start, end - start + 1);
            word.FontWeight =
                word.FontWeight.HasValue && word.FontWeight.Value == FontWeights.Bold
                ? FontWeights.Normal
                : FontWeights.Bold;
        }
    }
}
```

Notice that the **FontWeight** property returns a nullable value. If the range contains a mix of values for this attribute, the property returns null. The code used to toggle the **FontWeight** property is the same we used earlier when implementing the formatting toolbar.

The **GetPositionFromPoint** allows you to get a **C1TextPosition** object from a point on the screen. The **GetRectFromPosition** method performs the reverse operation, returning a **Rect** that represents the screen position of a **C1TextPosition** object. This is useful in situations where you want to present a UI element near a specific portion of a document.

Working with the C1Document Object

So far we have focused on the object model of the **C1RichTextBox** control. But the control is just an editable view of a **C1Document** object, which exposes a rich object model for creating and editing the underlying document. This architecture is similar to the one used by the WPF **RichTextBox** control, which provides a view of a **FlowDocument** object.

Programming directly against the `C1Document` object is the best way to perform many tasks, including report generation and the implementation of import and export filters. For example, the `Html` property exposes an HTML filter with methods that convert `C1Document` objects to and from HTML strings. You could implement a similar filter class to import and export other popular formats such as RTF or PDF.

The `C1RichTextBox` deals mainly with text. It provides a flat, linear view of the control content. The `C1Document`, on the other hand, exposes the structure of the document. The document model makes it easy to enumerate the runs within each paragraph, items within each list, and so on. This will be shown in a later section.

Creating Documents and Reports

To illustrate the process of creating a `C1Document`, we will walk through the steps required to implement a simple assembly documentation utility.

To start, create a new project and add a reference to the `C1.Silverlight` and `C1.Silverlight.RichTextBox` assemblies. Then edit the page constructor as follows:

```
using C1.Silverlight;
using C1.Silverlight.RichTextBox;
using C1.Silverlight.RichTextBox.Documents;

public partial class Page : UserControl
{
    // C1RichTextBox that will display the C1Document
    C1RichTextBox _rtb;

    public Page()
    {
        // default initialization
        InitializeComponent();

        // create the C1RichTextBox and add it to the page
        _rtb = new C1RichTextBox();
        LayoutRoot.Children.Add(_rtb);

        // create document and show it in the C1RichTextBox
        _rtb.Document = DocumentAssembly(typeof(C1RichTextBox).Assembly);
        _rtb.IsReadOnly = true;
    }
}
```

The code creates the `C1RichTextBox` and assigns its `Document` property to the result of a call to the `DocumentAssembly` method. It then makes the control read-only so users can't change the report.

The `DocumentAssembly` method takes an `Assembly` as argument and builds a `C1Document` containing the assembly documentation. Here is the implementation:

```
C1Document DocumentAssembly(Assembly asm)
{
    // create document
    C1Document doc = new C1Document();
    doc.FontFamily = new FontFamily("Tahoma");

    // assembly
    doc.Blocks.Add(new Heading1("Assembly\r\n" + asm.FullName.Split(',')[0]));
}
```

```

// types
foreach (Type t in asm.GetTypes())
    DocumentType(doc, t);

// done
return doc;
}

```

The method starts by creating a new **C1Document** object and setting its **FontFamily** property. This will be the default value for all text elements added to the document.

Next, the method adds a **Heading1** paragraph containing the assembly name to the new document's **Blocks** collection. Blocks are elements such as paragraphs and list items that flow down the document. They are similar to “div” elements in HTML. Some document elements contain an **Inlines** collection instead. These collections contain elements that flow horizontally, similar to “span” elements in HTML.

The **Heading1** class inherits from **C1Paragraph** and adds some formatting. We will add several such classes to the project, for normal paragraphs and headings 1 through 4.

The **Normal** paragraph is a **C1Paragraph** that takes a content string in its constructor:

```

class Normal : C1Paragraph
{
    public Normal(string text)
    {
        this.Inlines.Add(new C1Run() { Text = text });
        this.Padding = new Thickness(30, 0, 0, 0);
        this.Margin = new Thickness(0);
    }
}

```

The **Heading** paragraph extends **Normal** and makes the text bold:

```

class Heading : Normal
{
    public Heading(string text) : base(text)
    {
        this.FontWeight = FontWeights.Bold;
    }
}

```

Heading1 through **Heading4** extend **Heading** to specify font sizes, padding, borders, and colors:

```

class Heading1 : Heading
{
    public Heading1(string text) : base(text)
    {
        this.Background = new SolidColorBrush(Colors.Yellow);
        this.FontSize = 24;
        this.Padding = new Thickness(0, 10, 0, 10);
        this.BorderBrush = new SolidColorBrush(Colors.Black);
        this.BorderThickness = new Thickness(3, 1, 1, 0);
    }
}

```

```
class Heading2 : Heading
{
    public Heading2(string text): base(text)
    {
        this.FontSize = 18;
        this.FontStyle = FontStyles.Italic;
        this.Background = new SolidColorBrush(Colors.Yellow);
        this.Padding = new Thickness(10, 5, 0, 5);
        this.BorderBrush = new SolidColorBrush(Colors.Black);
        this.BorderThickness = new Thickness(3, 1, 1, 1);
    }
}
class Heading3 : Heading
{
    public Heading3(string text) : base(text)
    {
        this.FontSize = 14;
        this.Background = new SolidColorBrush(Colors.LightGray);
        this.Padding = new Thickness(20, 3, 0, 0);
    }
}
class Heading4 : Heading
{
    public Heading4(string text): base(text)
    {
        this.FontSize = 14;
        this.Padding = new Thickness(30, 0, 0, 0);
    }
}
```

Now that we have classes for all paragraph types in the document, it's time to add the content. Recall that we used a **DocumentType** method in the first code block. Here is the implementation for that method:

```
void DocumentType(C1Document doc, Type t)
{
    // skip non-public/generic
    if (!t.IsPublic || t.ContainsGenericParameters)
        return;

    // type
    doc.Blocks.Add(new Heading2("Class " + t.Name));

    // properties
    doc.Blocks.Add(new Heading3("Properties"));
    foreach (PropertyInfo pi in t.GetProperties())
    {
        if (pi.DeclaringType == t)
            DocumentProperty(doc, pi);
    }
}
```

```

// methods
doc.Blocks.Add(new Heading3("Methods"));
foreach (MethodInfo mi in t.GetMethods())
{
    if (mi.DeclaringType == t)
        DocumentMethod(doc, mi);
}

// events
doc.Blocks.Add(new Heading3("Events"));
foreach (EventInfo ei in t.GetEvents())
{
    if (ei.DeclaringType == t)
        DocumentEvent(doc, ei);
}
}

```

The method adds a **Heading2** paragraph with the class name and then uses reflection to enumerate all the public properties, events, and methods in the type. The code for these methods is simple:

```

void DocumentProperty(C1Document doc, PropertyInfo pi)
{
    if (pi.PropertyType.ContainsGenericParameters)
        return;

    doc.Blocks.Add(new Heading4(pi.Name));

    var text = string.Format("public {0} {1} {{ {2}{3} }}",
        pi.PropertyType.Name,
        pi.Name,
        pi.CanRead ? "get; " : string.Empty,
        pi.CanWrite ? "set; " : string.Empty);
    doc.Blocks.Add(new Normal(text));
}

```

The method adds a **Heading4** paragraph containing the property name, then some **Normal** text containing the property type, name, and accessors.

The methods used for documenting events and properties are analogous:

```

void DocumentMethod(C1Document doc, MethodInfo mi)
{
    if (mi.IsSpecialName)
        return;

    doc.Blocks.Add(new Heading4(mi.Name));
    var parms = new StringBuilder();
    foreach (var parm in mi.GetParameters())
    {
        if (parms.Length > 0)
            parms.Append(", ");
        parms.AppendFormat("{0} {1}", parm.ParameterType.Name, parm.Name);
    }
}

```

```

var text = string.Format("public {0} {1}({2})",
    mi.ReturnType.Name,
    mi.Name,
    parms.ToString());

doc.Blocks.Add(new Normal(text));
}

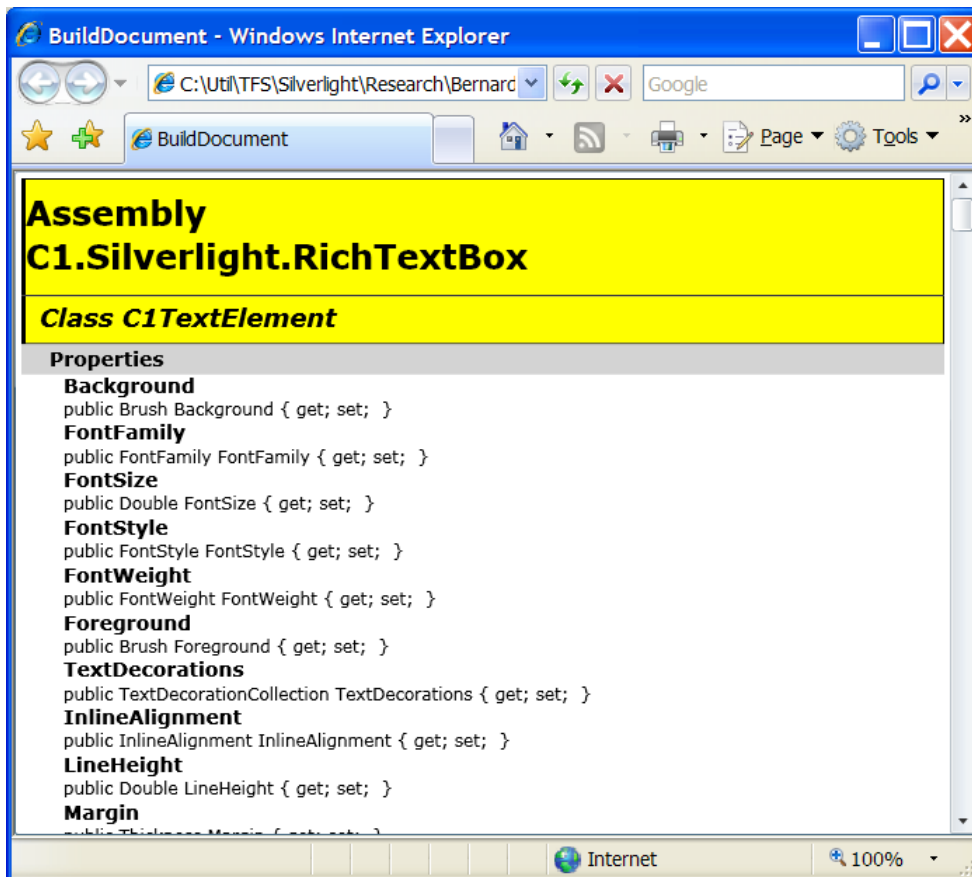
void DocumentEvent(C1Document doc, EventInfo ei)
{
    doc.Blocks.Add(new Heading4(ei.Name));

    var text = string.Format("public {0} {1}",
        ei.EventHandlerType.Name,
        ei.Name);

    doc.Blocks.Add(new Normal(text));
}

```

If you run the project now, you will see a window like the one shown below:



The resulting document can be viewed and edited in the **C1RichTextBox** like any other. It can also be exported to HTML using the **Html** property in the **C1RichTextBox**, or copied through the clipboard to applications such as Microsoft Word or Excel.

You could use the same technique to create reports based on data from a database. In addition to formatted text, the **C1Document** object model supports the following features:

- **Lists**
Lists are created by adding **C1List** objects to the document. The **C1List** object has a **ListItems** property that contains **C1ListItem** objects, which are also blocks.
- **Hyperlinks**
Hyperlinks are created by adding **C1Hyperlink** objects to the document. The **C1Hyperlink** object has an **Inlines** property that contains a collection of runs (typically **C1Run** elements that contain text), and a **NavigateUrl** property that determines the action to be taken when the hyperlink is clicked.
- **Images**
Images and other **FrameworkElement** objects are created by adding **C1BlockUIContainer** objects to the document. The **C1BlockUIContainer** object has a **Child** property that can be set to any **FrameworkElement** object.
Note that not all objects can be exported to HTML. Images are a special case that the HTML filter knows how to handle.

Implementing Split Views

Many editors offer split-views of a document, allowing you to keep a part of the document visible while you work on another part.

You can achieve this easily by connecting two or more **C1RichTextBox** controls to the same underlying **C1Document**. Each control acts as an independent view, allowing you to scroll, select, and edit the document as usual. Changes made to one view are reflected on all other views.

To show how this works, let's extend the previous example by adding a few lines of code to the page constructor:

```
public Page()
{
    // default initialization
    InitializeComponent();

    // create the C1RichTextBox and add it to the page
    _rtb = new C1RichTextBox();
    LayoutRoot.Children.Add(_rtb);

    // create document and show it in the C1RichTextBox
    _rtb.Document = DocumentAssembly(typeof(C1RichTextBox).Assembly);
    _rtb.IsReadOnly = true;

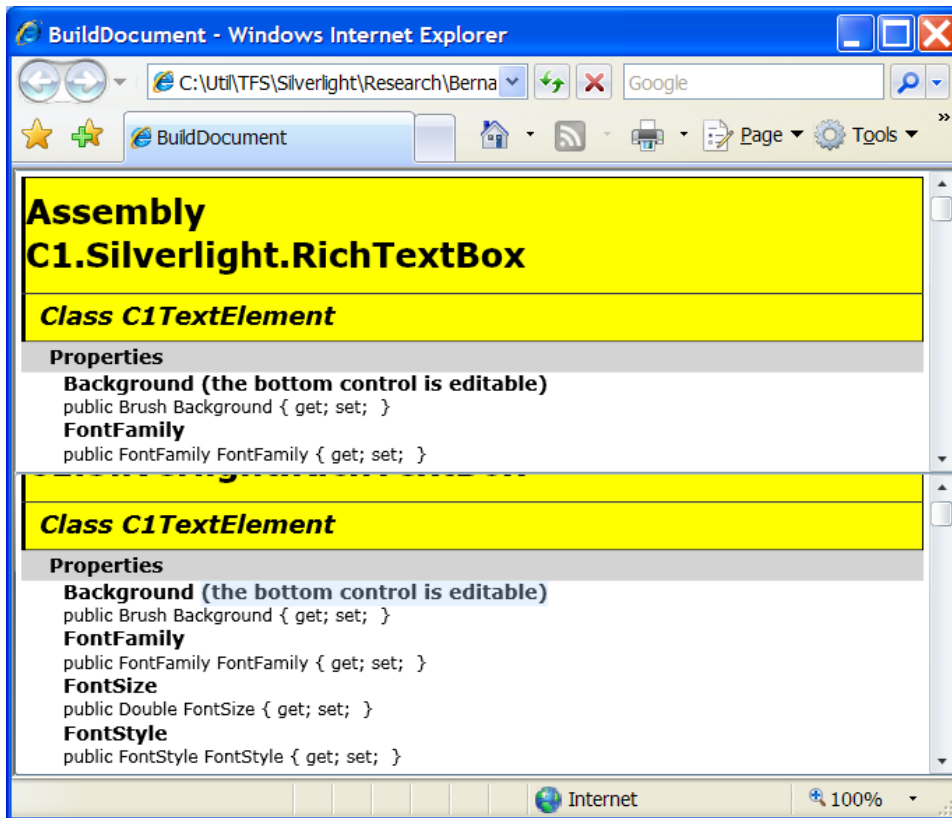
    // attach event handler
    _rtb.ElementMouseDown += _rtb_ElementMouseDown;

    // add a second C1RichTextBox to the page
    LayoutRoot.RowDefinitions.Add(new RowDefinition());
    LayoutRoot.RowDefinitions.Add(new RowDefinition());
    var rtb2 = new C1RichTextBox();
    rtb2.SetValue(Grid.RowProperty, 1);
    LayoutRoot.Children.Add(rtb2);

    // bind the second C1RichTextBox to the same document
    rtb2.Document = _rtb.Document;
}
```

The new code adds a new **C1RichTextBox** to the page and then sets its **Document** property to the document being shown by the original **C1RichTextBox**.

If you run the project again, you will see a window like the one shown below:



The bottom control is editable (we did not set its **IsReadOnly** property to **False**). If you type into it, you will see the changes on both controls simultaneously.

The mechanism is general; we could easily attach more views of the same document. Moreover, any changes you make to the underlying document are immediately reflected on all views.

Using the C1Document class

As we mentioned earlier, the **C1RichTextBox** provides a linear, flat view of the control content, while **C1Document** class exposes the document structure.

To illustrate the advantages of working directly with the document object, suppose we wanted to add some functionality to the previous sample: when the user presses the Control key, we want to capitalize the text in all paragraphs of type **Heading2**.

The object model exposed by the **C1RichTextBox** is not powerful enough to do this reliably. You would have to locate spans based on their formatting, which would be inefficient and unreliable (what if the user formatted some plain text with the same format used by **Heading2**?).

Using the C1Document object model, this task becomes trivial:

```
public Page()
{
    // default initialization
    InitializeComponent();

    // no changes here...

    // bind the second C1RichTextBox to the same document
    rtb2.Document = _rtb.Document;
    rtb2.KeyDown += rtb2_KeyDown;
}
void rtb2_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Ctrl)
    {
        foreach (var heading2 in _rtb.Document.Blocks.OfType<Heading2>())
        {
            var text = heading2.ContentRange.Text;
            heading2.ContentRange.Text = text.ToUpper();
        }
    }
}
```

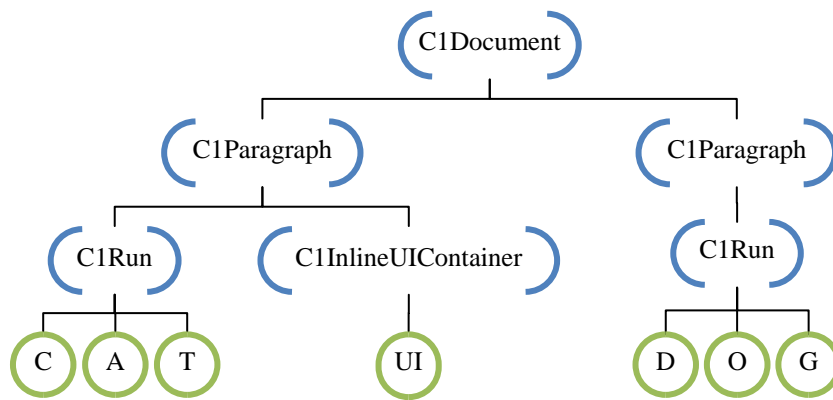
The code monitors the keyboard. When the user presses the control key, it enumerates all **Heading2** elements in the document and replaces their content with capitals.

Doing the same thing using the **C1RichTextBox** object model would require a lot more code and the result would not be as reliable.

Understanding C1TextPointer

The **C1TextPointer** class represents a position inside a **C1Document**. It is intended to facilitate traversal and manipulation of **C1Documents**. The functionality is analogous to WPF's **TextPointer** class, although the object model has many differences.

A **C1TextPointer** is defined by a **C1TextElement** and an offset inside of it. Let's take this document as an example:



The blue bracketed nodes above are **C1TextElements**; the offset of a **C1TextPointer** indicates between which children the position is located. For instance, a position that points to the **C1Document** above with offset 0 is just before the first **C1Paragraph**, offset 1 is between the two paragraphs, and offset 2 is after the second paragraph. When a **C1TextPointer** points to a **C1Run**, each character in its text is considered a child of **C1Run**, so the offset indicates a position inside the text. A **C1InlineUIContainer** is considered to have a single child (the **UIElement** it displays) so it has two positions, one before and one after the child.

An alternative way to visualize a document is as a sequence of symbols, where a symbol can be either an element tag or some type of content. An element tag indicates the start or end of an element. In XML, the above document would be written as:

```
<C1Document>
  <C1Paragraph>
    <C1Run>CAT</C1Run>
    <C1InlineUIContainer><UI/></C1InlineUIContainer>
  </C1Paragraph>
  <C1Paragraph>
    <C1Run>DOG</C1Run>
  </C1Paragraph>
</C1Document>
```

Viewing the document like this, a **C1TextPointer** points to a position between tags or content. This view also gives a clear order to **C1TextPointer**. In fact, **C1TextPointer** implements **IComparable**, and also overloads the comparison operators for convenience.

The symbol that is after a **C1TextPointer** can be obtained using the **Symbol** property. This property returns an object that can be of type **StartTag**, **EndTag**, **char** or **UIElement**.

If you want to iterate through the positions in a document, there are two methods available:

GetPositionAtOffset and **Enumerate**. **GetPositionAtOffset** is the low-level method; it just returns the position at a specified integer offset. **Enumerate** is the recommended way to iterate through positions. It returns an **IEnumerable<C1TextPointer>** that iterates through all the positions in a specified direction. For instance, this returns all the positions in a document:

```
document.ContentStart.Enumerate()
```

Note that **ContentStart** returns the first **C1TextPointer** in a **C1TextElement**; there is also a **ContentEnd** property that returns the last position.

The interesting thing about **Enumerate** is that it returns a lazy enumeration. That is, **C1TextPointer** objects are only created when the **IEnumerable** is iterated. This allows for efficient use of LINQ extensions methods for filtering, finding, selecting, etc. As an example, let's say we want to get the **C1TextRange** for the word contained under a **C1TextPointer**. We can do this:

```
C1TextRange ExpandToWord(C1TextPointer pos)
{
    // find word start
    var wordStart = pos.IsWordStart
        ? pos
        : pos.Enumerate(LogicalDirection.Backward).First(p => p.IsWordStart);
```

```

// find word end
var wordEnd = pos.IsWordEnd
    ? pos
    : pos.Enumerate(LogicalDirection.Forward).First(p => p.IsWordEnd);

// return new range from word start to word end
return new C1TextRange(wordStart, wordEnd);
}

```

The **Enumerate** method returns the positions in a specified direction, but it doesn't include the current position. So we first check if the parameter position is a word start, and if not, we search backward for a position that is a word start. Likewise for the word end, we check the parameter position and then we search forward. We want to find the word that contains the parameter position, so we need the first word end moving forward and the first word start moving backward. **C1TextPointer** already contains the properties **IsWordStart** and **IsWordEnd** that tells you whether a position is a word start or end depending on the surrounding symbols. We use the **First** LINQ extension method to find the first position that satisfies our required predicate. And finally we create a **C1TextRange** from the two positions.

LINQ extension methods can be very useful when working with positions. As another example we can count the words in a document like this:

```
document.ContentStart.Enumerate().Count(p => p.IsWordStart && p.Symbol is char)
```

Note that we need to check that the symbol following a word start is a char because **IsWordStart** returns **True** for positions that are not exactly at the start of a word. For instance the position just before a **C1Run** start tag is considered a word start if the first position of the **C1Run** is a word start.

Let's implement a **Find** method as another example:

```

C1TextRange FindWordFromPosition(C1TextPointer position, string word)
{
    // get all ranges whose text length is equal to word.Length
    var ranges = position.Enumerate().Select(pos =>
    {
        // get a position that is at word.Length offset
        // but ignoring tags that do not change the text flow
        var end = pos.GetPositionAtOffset(word.Length,
C1TextRange.TextTagFilter);
        return new C1TextRange(pos, end);
    });
    // returned value will be null if word is not found.
    return ranges.FirstOrDefault(range => range.Text == word);
}

```

We want to find the word from a specified position, so we enumerate all positions forward, and select all ranges whose text length is **word.Length**. For each position we need to find the position that is at **word.Length** distance. For this we use the **GetPositionAtOffset** method. This method returns a position at a specified offset, but it also counts all inline tags as valid positions, we need to ignore this to account for the case when a word is split between two **C1Run** elements. That is why we use **C1TextRange.TextTagFilter**; this is the same filter method used by the internal logic that translates document trees into text. As a final step we search for the range whose text matches the searched word.

As a last example let's replace the first occurrence of a word:

```
var wordRange = FindWordFromPosition(document.ContentStart, "cat");
if (wordRange != null)
{
    wordRange.Text = "dog";
}
```

We can use the previous example to first find the word, and then replace the text by just assigning to Text property.

As we mentioned earlier, **C1TextPointer** is defined by a **C1TextElement** and an offset in it. A consequence of this is when the Document changes, positions may become invalid. Let's say we have a **C1Run** with the text "Cat" and a position at the end of this text. If some substring of the text is removed, our position will become invalid because its offset will be greater than the length of the text. As another example, let's say we add "The" to the beginning of the text. Then the position that was at the end of "Cat" will be pointing at the end of "The", which is also incorrect. To work around this problem, the argument of the **TextChanged** event in **C1RichTextBox** and **C1TextElement** includes a Fix method. This method transforms any position that was valid before the change, to a new position that is valid in the changed document. Even if a position is valid in the new document, it transforms the position so that it remains between the same symbols as before. For instance, in the case of inserting "The" before "Cat", the position that was at the end of "Cat" will be fixed to be at the end of "TheCat". Let's illustrate how the **C1TextChangedEventArgs.Fix** method is used:

```
C1TextPointer pos = SomeTextPointer();
document.TextChanged += (s, e) =>
{
    pos = e.Fix(pos);
    // do something with the fixed pos
};
```