ComponentOne

RichTextBox for Windows Phone

ComponentOne, a division of GrapeCity

201 South Highland Avenue, Third 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)

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne RichTextBox for Windows Phone Overview

Add powerful rich text editing and viewing to your mobile apps. **ComponentOne RichTextBox™ for Windows Phone** supports common formatting tags, automatic line wrapping, HTML and RTF import/export, table support, images, annotations, and more.

Help with ComponentOne Studio for Windows Phone

Getting Started

For information on installing **ComponentOne Studio for Windows Phone**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Studio for Windows Phone.

What's New

For a list of the latest features added to **ComponentOne Studio for Windows Phone**, visit What's New in Studio for Windows Phone.

ComponentOne RichTextBox for Windows Phone Assemblies and Controls

Add powerful rich text editing and viewing to your mobile apps. **ComponentOne RichTextBox™ for Windows Phone** supports common formatting tags, automatic line wrapping, HTML and RTF import/export, table support, images, annotations, and more.

The following main classes are included in the **C1.Phone.RichTextBox.dll** assembly:

- **C1RichTextBox:** Control used to view and edit **C1Document** documents. The control can import and export HTML, and supports rich formatting including fonts, foreground and background colors, borders, paragraph alignment, images, hyperlinks, lists, arbitrary **UIElement** objects, and more.
- **C1Document:** Class that represents documents as a hierarchical list of elements that represent document components such as paragraphs, lists, images, and so on. The object model exposed by the **C1Document** object is similar to the one used in the **WPF FlowDocument** class and in the HTML DOM.

RichTextBox for Windows Phone Key Features

ComponentOne RichTextBox for Windows Phone allows you to create customized, rich applications. Make the most of **RichTextBox for Windows Phone** by taking advantage of the following key features:

- **RTF and HTML Formats**

  C1RichTextBox supports editing HTML, RTF and plain text. Load existing content into the **C1RichTextBox** control, edit the document and save it back. Or use the control in readonly mode to display blocks of rich content.

- **Apply Rich Formatting**

  Edit and format text containing multiple fonts, decorations, colors, tables, images, lists, and more. **C1RichTextBox** supports the following commands: Undo, Redo, Font Family, Font Size, Grow Font, Shrink Font, Bold, Italic, Underline, Change Case, Subscript, Superscript, Text Color, Text Highlight Color, Align Left, Align Center, Align Right, Justify, Bullets, Numbering, Text Wrapping, Border Thickness, Border Color, Paragraph Color, Margin, and Padding.

- **Application Buttons**

  C1RichTextBox comes with every button needed for formatting text. You can customize which commands to put in your Application Bar.

- **Hyperlinks**

  Take the user to another page within your app or on the Web by displaying hyperlinks within a block of text.

RichTextBox for Windows Phone Quick Start

In this quick start you'll create a Windows Phone application in Visual Studio 2010, add the **C1RichTextBox** to the application, add code to customize the application, and run the application to view possible run-time interactions.

Step 1 of 3: Creating a Windows Phone Application

In this step you'll create a new Windows Phone application, set the application up, and add the **C1RichTextBox** control to the application. After completing this step, you should have a mostly functional Rich Text editor.

Complete the following steps:

1. In Visual Studio, select **File | New | Project** to open the **New Project** dialog box.
2. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **Windows Phone Application**. Enter a **Name** for your project and click **OK**. The **New Windows Phone Application** dialog box will appear.
3. Click **OK** to close the **New Windows Phone Application** dialog box and create your project.
4. Right-click the project in the Solution Explorer and select **Add Reference**.
5. In the **Add Reference** dialog box, locate and select the **C1.Phone.dll** and **C1.Phone.RichTextBox.dll** assemblies and select **OK**.
6. Add the XAML namespace to the `<phone:PhoneApplicationPage>` tag by adding `xmlns:c1richtextbox="clr-namespace:C1.Phone.RichTextBox;assembly=C1.Phone.RichTextBox"` so it appears similar to the following:

```
<phone:PhoneApplicationPage x:Class="C1WP7.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:c1richtextbox="clr-
namespace:C1.Phone.RichTextBox;assembly=C1.Phone.RichTextBox"
mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="Portrait" Orientation="Portrait"
shell:SystemTray.IsVisible="True">
```

7. Edit the **TitlePanel** content to change the text in the **TextBlock** controls. It will appear similar to the following:

```
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="ComponentOne Studio for
Windows Phone" Style="{StaticResource PhoneTextNormalStyle}"/>
```

```
    <TextBlock x:Name="PageTitle" Text="RichTextBox" Margin="9,-7,0,0"
Style="{StaticResource PhoneTextTitle1Style}" FontSize="64"/>
</StackPanel>
```

8. In the XAML window of the project, place the cursor between the `<Grid x:Name="ContentPanel"></Grid>` tags and click once.
9. Add the following markup within the `<Grid>` tags to add the **C1RichTextBox** control:
```
<c1richtextbox:C1RichTextBox />
```

What You've Accomplished

If you run the application, you'll see an almost fully functional **C1RichTextBox** application. You can enter and edit text in the **C1RichTextBox** control. In the next step you'll customize the application further.

Step 2 of 3: Customizing the RichTextBox

In the previous step you created a new Windows Phone application, set the application up, and added the **C1RichTextBox** control to the application. In this step you'll customize the application further.

Complete the following steps:

1. In the Solution Explorer, right-click the **MainPage.xaml** file and select **View Code** to open the code file.
2. In the Code Editor, add the following code to import the following namespaces:
   - Visual Basic
   ```
   Imports C1.Phone.RichTextBox
   ```
   - C#
   ```
   using C1.Phone.RichTextBox;
   ```
3. Add the following code to the **MainPage** constructor:
   - Visual Basic
   ```
   Me.C1RTB.FontSize = 24
   Me.C1RTB.Text = "Hello World! Welcome to the most complete rich text
   editor available for Windows Phone. Load, edit, and save formatted text
   as HTML or RTF documents with ComponentOne RichTextBox for Windows
   Phone. The C1RichTextBox control provides rich formatting, automatic
   line wrapping, HTML and RTF import/export, table support, images,
   annotations, and more."
   ```
   - C#
   ```
   this.C1RTB.FontSize = 24;
   this.C1RTB.Text = "Hello World! Welcome to the most complete rich text
   editor available for Windows Phone. Load, edit, and save formatted text
   as HTML or RTF documents with ComponentOne RichTextBox for Windows
   Phone. The C1RichTextBox control provides rich formatting, automatic
   line wrapping, HTML and RTF import/export, table support, images,
   annotations, and more.";
   ```

   This code adds content to the **C1RichTextBox** control.

What You've Accomplished

In this step you added content to your **C1RichTextBox** application, now all that's left is to run it. In the next step you'll run the application and view some of the run-time interactions possible with the **C1RichTextBox** control.

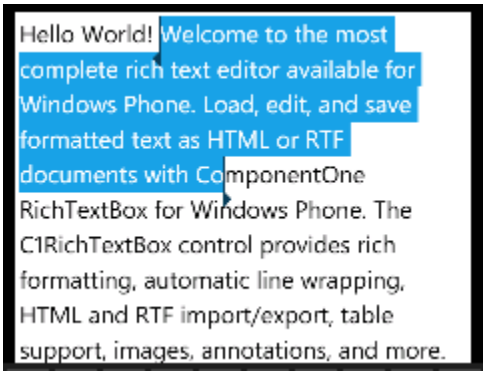
Step 3 of 3: Running the Application

In the previous steps you created a new Windows Phone application, added the **C1RichTextBox** control, and customized the application. All that's left now is to run the application and view some possible run-time interactions.

Complete the following steps:

1. In the menu select **Debug | Start Debugging** to run the application. The running application will appear in the Windows Phone emulator similar to the following image:

2. Tap inside the **C1RichTextBox** control. The on-screen keyboard will appear allowing you to enter text.
3. Enter text inside the **C1RichTextBox** using the on-screen keyboard.
4. Select a word, for example "Welcome". Notice that the word is highlighted and selected. To change you selection, you can select and drag the handles that appear on either side of the selected word. For example, select part of the next sentence:



What You've Accomplished

Congratulations, you've completed this tutorial! You learned a bit about using the **C1RichTextBox** control. In this tutorial you created a new Windows Phone application, added the **C1RichTextBox** control, and viewed some possible run-time interactions.

Working with RichTextBox for Windows Phone

The most complete rich text editor available for Windows Phone, load, edit, and save formatted text as HTML or RTF documents with **ComponentOne RichTextBox™ for Windows Phone**. The **C1RichTextBox** control provides rich formatting, automatic line wrapping, HTML and RTF import/export, table support, images, annotations, and more.

The **C1.Phone.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, and so on. 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.

C1RichTextBox Concepts and Main Properties

On the surface, the C1RichTextBox control appears just like a standard **TextBox**. It provides the same properties to control the font, colors, text, and selection. That can be an advantage – if you have an application that uses **TextBox** controls, you may be able to simply replace them with C1RichTextBox controls without any additional changes.

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

- Visual Basic

```
Private Sub SearchAndReplace(tb As TextBox, find As String, replace As
String)
    Dim start As Integer = 0
```

```
    While True
        Dim pos As Integer = tb.Text.IndexOf(find, start)
        If pos < 0 Then
            Exit While
        End If
        tb.[Select](pos, find.Length)
        ' Optionally show a dialog box to confirm the change.
        tb.SelectedText = replace
        start = pos + 1
    End While
End Sub
```

- C#

```
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 a 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:

- Visual Basic

```
Private Sub SearchAndReplace(tb As TextBox, find As String, replace As
String)
    Dim start As Integer = 0
    While True
        Dim pos As Integer = tb.Text.IndexOf(find, start)
        If pos < 0 Then
            Exit While
        End If
        tb.[Select](pos, find.Length)
        ' Optionally show a dialog box to confirm the change.
        tb.Selection.InlineBackground = New SolidColorBrush(Colors.Yellow)
        tb.SelectedText = replace
        start = pos + 1
    End While
End Sub
```

- C#

```
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 a dialog box 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.

C1RichTextBox Content

The content of the C1RichTextBox can be specified in two ways, using the Text property or the Html property. The Text property is used to assign and retrieve the control content as plain text.

- Visual Basic
```
Me.C1RichTextBox1.Text = "Hello World!"
```
- C#
```
this.c1RichTextBox1.Text = "Hello World!";
```

The Html property is used to assign and retrieve formatted text as HTML. The HTML text needs to be encoded in the XAML file, so, for example, instead of `<b>` for bold, tags are encoded as `&lt;b&gt;`.

- Visual Basic
```
Me.C1RichTextBox1.Html = "&lt;b&gt;Hello World!&lt;/b&gt;"
```
- C#
```
this.c1RichTextBox1.Html = "&lt;b&gt;Hello World!&lt;/b&gt;"
```

The C1RichTextBox exposes a TextWrapping property that specifies whether the control should wrap long lines or whether it should keep the lines together and provide a horizontal scrollbar instead.

- Visual Basic
```
Me.C1RichTextBox1.TextWrapping = TextWrapping.NoWrap
```
- C#
```
this.c1RichTextBox1.TextWrapping = TextWrapping.NoWrap;
```

The code above sets the C1RichTextBox control so that text content will not wrap in the control and will appear in a continuous line.

Configuring the Windows Phone Clipboard with C1RichTextBox

On the Windows Phone you get automatic copy and paste support for standard TextBoxes. For other controls you wish to enable copy/paste functionality, you typically have to use a TextBox in disguise. To put text to the clipboard you simply call the static **SetText** method on the **System.Windows.Clipboard** class like this:

```
Clipboard.SetText("some text");
```

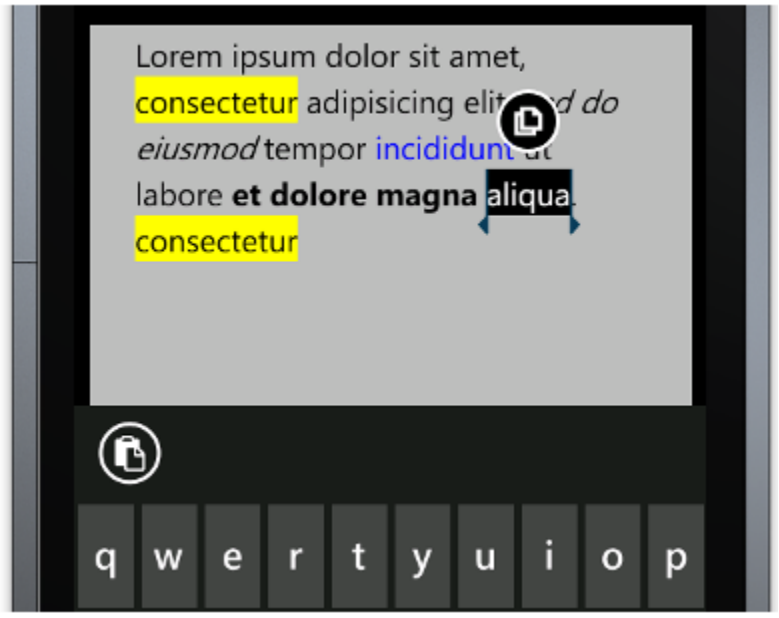
Copying Plain Text to the Clipboard

The **ComponentOne RichTextBox** control is part **TextBox** and part custom control. As of the 2012 v1 version, the control does not have built-in copying functionality, but if text resides on the clipboard, then you can paste it into C1RichTextBox for free. So in order to enable copying text, we must add it ourselves.

To put plain text on the clipboard the code would simply be:

```
// set clipboard text to plain text
Clipboard.SetText(c1RichTextBox1.SelectedText);
```

Next, we can create a custom round button that displays when the user selects some text in C1RichTextBox just as it would for a standard **TextBox**.



To create a round button we override the style to give it rounded corners (Border CornerRadius = 33).

```
<Style x:Key="RoundButtonStyle" TargetType="Button">
      <Setter Property="Background" Value="{StaticResource
PhoneBackgroundBrush}"/>
      <Setter Property="BorderBrush" Value="{StaticResource
PhoneForegroundBrush}"/>
      <Setter Property="Foreground" Value="{StaticResource
PhoneForegroundBrush}"/>
      <Setter Property="BorderThickness" Value="{StaticResource
PhoneBorderThickness}"/>
      <Setter Property="FontFamily" Value="{StaticResource
PhoneFontFamilySemiBold}"/>
      <Setter Property="FontSize" Value="{StaticResource
PhoneFontSizeMediumLarge}"/>
      <Setter Property="Padding" Value="0"/>
      <Setter Property="Template">
    <Setter.Value>
          <ControlTemplate TargetType="Button">
              <Grid Background="Transparent">
                  <VisualStateManager.VisualStateGroups>
                      <VisualStateGroup x:Name="CommonStates">
                          <VisualState x:Name="Normal"/>
                          <VisualState x:Name="MouseOver"/>
                          <VisualState x:Name="Pressed">
                              <Storyboard>
                                  <ObjectAnimationUsingKeyFr
ames
```

```
                Storyboard.TargetProperty="Foreg
                round"
                Storyboard.TargetName="ContentCo
                ntainer">
                      <DiscreteObjectKeyFrame
                KeyTime="0"
                Value="{StaticResource
                PhoneBackgroundBrush}"/>
                      </ObjectAnimationUsingKeyF
                rames>
                      <ObjectAnimationUsingKeyFr
                      ames
                      Storyboard.TargetProperty=
                      "Background"
                      Storyboard.TargetName="But
                      tonBackground">
                      <DiscreteObjectKeyFrame
                KeyTime="0"
                Value="{StaticResource
                PhoneForegroundBrush}"/>
                      </ObjectAnimationUsingKeyF
                rames>
                      <ObjectAnimationUsingKeyFr
                ames
                Storyboard.TargetProperty="Borde
                rBrush"
                Storyboard.TargetName="ButtonBac
                kground">
                      <DiscreteObjectKeyFrame
                KeyTime="0"
                Value="{StaticResource
                PhoneForegroundBrush}"/>
                      </ObjectAnimationUsingKeyF
                rames>
                      </Storyboard>
            </VisualState>
            <VisualState x:Name="Disabled">
                      <Storyboard>
                      <ObjectAnimationUsingKeyFr
                ames
                Storyboard.TargetProperty="Foreg
                round"
                Storyboard.TargetName="ContentCo
                ntainer">
                      <DiscreteObjectKeyFrame
                KeyTime="0"
                Value="{StaticResource
                PhoneDisabledBrush}"/>
                      </ObjectAnimationUsingKeyFrames>
                      <ObjectAnimationUsingKeyFrames
          Storyboard.TargetProperty="BorderBrush
          "
          Storyboard.TargetName="ButtonBackgroun
          d">
                      <DiscreteObjectKeyFrame
                KeyTime="0"
```

```
                                        Value="{StaticResource
                                        PhoneDisabledBrush}"/>
                                        </ObjectAnimationUsingKeyFrames>
                                        <ObjectAnimationUsingKeyFrames
                                    Storyboard.TargetProperty="Background"
                                    Storyboard.TargetName="ButtonBackgroun
                                    d">
                                            <DiscreteObjectKeyFrame
                                        KeyTime="0"
                                        Value="Transparent"/>
                                        </ObjectAnimationUsingKeyFrames>
                                    </Storyboard>
                                </VisualState>
                            </VisualStateGroup>
                        </VisualStateManager.VisualStateGroups>
                        <Border x:Name="ButtonBackground"
                        BorderBrush="{TemplateBinding BorderBrush}"
                        BorderThickness="{TemplateBinding BorderThickness}"
                        Background="{TemplateBinding Background}"
                        CornerRadius="33" Margin="{StaticResource
                        PhoneTouchTargetOverhang}">
                            <ContentControl x:Name="ContentContainer"
                        ContentTemplate="{TemplateBinding ContentTemplate}"
                        Content="{TemplateBinding Content}"
                        Foreground="{TemplateBinding Foreground}"
                        HorizontalContentAlignment="{TemplateBinding
                        HorizontalContentAlignment}" Padding="{TemplateBinding
                        Padding}" VerticalContentAlignment="{TemplateBinding
                        VerticalContentAlignment}">
                            <ContentControl.RenderTransform>
                                <ScaleTransform x:Name="buttonScale" />
                            </ContentControl.RenderTransform>
                        </ContentControl>
                    </Border>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
```

Then we can add a button to our page containing C1RichTextBox like this:

```
<Button x:Name="btnCopyPlainText"
        Width="72"
        Height="72"
        Click="btnCopyPlainText_Click"
        Margin="162,160, 0, 0"
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Style="{StaticResource RoundButtonStyle}"
        Visibility="Collapsed" >
    <Image x:Name="imgPlainText"
           Source="Resources\Copy.png"
           Stretch="Fill" />
</Button>
```

Notice that we are setting the **HorizontalAlignment**, **VerticalAlignment** and **Margin**. At run-time we will update the **Margin** to reflect the position of the selected text. To detect the selection changing we simply subscribe to the SelectionChanged event.

```
private void c1RichTextBox1_SelectionChanged(object sender, EventArgs
e)
{
     PositionCopyButton();
```

We can grab the **Rect** of the selected text by using the GetRectFromPosition method and passing in the **SelectedText Start** and **End** positions. From this **Rect** we set the **Left** and **Top** margin values of our button so that it appears to float above the selected text.

```
private void PositionCopyButton()
{
  if (!string.IsNullOrEmpty(c1RichTextBox1.SelectedText))
      {
   // get bounds of selected text
   Rect rtStart =
   c1RichTextBox1.GetRectFromPosition(c1RichTextBox1.Selection.Start);
   Rect rtEnd =
c1RichTextBox1.GetRectFromPosition(c1RichTextBox1.Selection.End);
   // center button above selected text
   btnCopyPlainText.Margin = new Thickness(rtStart.X + ((rtEnd.X -
rtStart.X) / 2) - btnCopyPlainText.Width / 2, rtStart.Y -
btnCopyPlainText.Height, 0, 0);
   btnCopyPlainText.Visibility = System.Windows.Visibility.Visible;
}
else
{
     // hide button if no text to copy
     btnCopyPlainText.Visibility =
System.Windows.Visibility.Collapsed;
}
```

Then after we set the selected text to the clipboard we must set the **Focus** from the copy button back to C1RichTextBox. We can do this by simply calling the **Focus** method.

```
private void btnCopyPlainText_Click(object sender, RoutedEventArgs e)
{
  // set clipboard text to plain text
  Clipboard.SetText(c1RichTextBox1.SelectedText);
  // keep focus on richtextbox
  c1RichTextBox1.Focus();
 }
```

That is it for copying plain text!

Saving and Loading HTML

You can persist the contents of a simple **TextBox** control using the **Text** property. You can also use the Text property to persist content in the **C1RichTextBox** control, but you will lose any rich formatting. 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, and so on. But the filter does not support all 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 markup:

```
<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.

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:

- Visual Basic

```
Public Sub New()
    InitializeComponent()

    ' Set text
    _rtb.Text = "This is some text with a hyperlink in it."

    ' Create hyperlink
    Dim pos As Integer = _rtb.Text.IndexOf("hyperlink")
    _rtb.[Select](pos, 9)
    Dim uri = New Uri("http://www.componentone.com", UriKind.Absolute)
    _rtb.Selection.MakeHyperlink(uri)

    ' Handle navigation requests
    _rtb.NavigationMode = NavigationMode.Always
    AddHandler _rtb.RequestNavigate, AddressOf _rtb_RequestNavigate;
End Sub
```

- C#

```
public MainPage()
{
  InitializeComponent();

  // 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.Always;
  _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 moving the mouse over a hyperlink, and tapping 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 setting the Content by loading a new WebBrowser control that is navigating to a different URL. This is illustrated below:

- Visual Basic

```
Private Sub _rtb_RequestNavigate(sender As Object, e As
RequestNavigateEventArgs)
    ' Set the Content by loading a web browser control
    Dim browser = New WebBrowser
    browser.Navigate(e.Hyperlink.NavigateUri)
    Content = browser
End Sub
```

- C#

```
void _rtb_RequestNavigate(object sender, RequestNavigateEventArgs e)
{
    // Set the Content by loading a web browser control
    WebBrowser browser = new WebBrowser();
    browser.Navigate(e.Hyperlink.NavigateUri);
    Content = browser;
}
```

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:

- Visual Basic

```
Public Sub New()
    InitializeComponent()

    ' Set text
    _rtb.Text = "This is some text with a hyperlink in it."

    ' Create hyperlink
    Dim pos As Integer = _rtb.Text.IndexOf("hyperlink")
    _rtb.[Select](pos, 9)
    Dim uri = New Uri("msgbox:Thanks for clicking!")
    _rtb.Selection.MakeHyperlink(uri)

    ' Handle navigation requests
    _rtb.NavigationMode = NavigationMode.Always
    AddHandler _rtb.RequestNavigate, AddressOf _rtb_RequestNavigate
End Sub
Private Sub  rtb RequestNavigate(sender As Object, e As
RequestNavigateEventArgs)
    Dim uri As Uri = e.Hyperlink.NavigateUri
    If uri.Scheme = "msgbox" Then
        MessageBox.Show(uri.LocalPath)
    End If
End Sub
```

- C#

```
public MainPage()
{
  InitializeComponent();

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

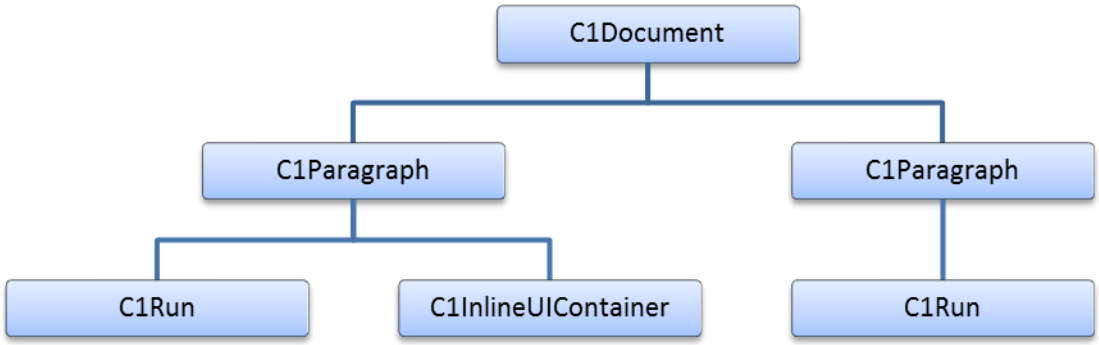
The only change in the MakeHyperlink code is the line that creates the URI. The RequestNavigate handler uses the URI members to parse the command and argument. You could use this technique to create documents with embedded menus for example.

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.

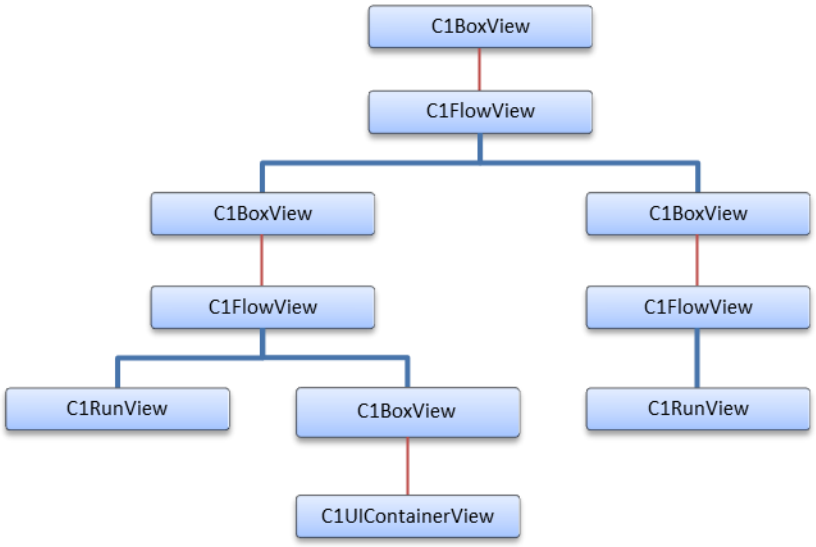
Accessing Layout Information

When C1RichTextBox creates the C1Document layout, it creates a parallel tree composed of C1TextElementView objects. For each C1TextElement in the C1Document tree, there is at least one C1TextElementView that is tasked with its layout and drawing.

Take this C1Document tree as an example:



Its corresponding view tree will look like the following:



Each C1TextElementView provides some basic layout information for its corresponding C1TextElement:

- Origin: this is the origin of the view in document coordinates.
- DesiredSize: this is the desired size of the view from the last time it was measured.

Multiple C1TextElementViews can be composed to handle layout and drawing for out C1TextElement. When this is done, the Content property contains the inner most C1TextElementView in the composition. The content view's children correspond to the Children collection of the associated C1TextElement.

View composition is used in C1BoxView to handle margin, padding and border for its Content view. This means that the origin of each C1BoxView is outside the margin, padding and border box, while the origin of its Content is inside.

C1FlowView takes care of flowing boxes and text into lines. Each line is represented by a C1Line object. Note that C1Lines not only contains single lines of text, but may also contain an entire paragraph. Each C1FlowView contains a list of C1Line, which are always vertically stacked. In turn, each C1Line is composed of C1LineFragments, which are horizontally stacked. C1LineFragments have a reference to the child element whose origin matches the position of the fragment.

For example, the following code counts the lines in a C1RichTextBox:

- Visual Basic

```
Private Function CountLines(rtb As C1RichTextBox) As Integer
    Dim root = rtb.ViewManager.GetView(rtb.Document)
    Return CountLines(root)
End Function

Private Function CountLines(view As C1TextElementView) As Integer
    Dim count As Integer = 0
    Dim flow = TryCast(view, C1FlowView)
    If flow IsNot Nothing Then
        For Each line As var In flow.Lines
            If TypeOf line.Fragments.First().Element Is C1Inline Then
                count += 1
            End If
        Next
```

```
        End If
        For Each child As var In view.Children
            count += CountLines(child)
        Next
        Return count
    End Function
```

- *C#*

```
int CountLines(C1RichTextBox rtb)
{
    var root = rtb.ViewManager.GetView(rtb.Document);
    return CountLines(root);
}

int CountLines(C1TextElementView view)
{
    int count = 0;
    var flow = view as C1FlowView;
    if (flow != null)
    {
        foreach (var line in flow.Lines)
        {
            if (line.Fragments.First().Element is C1Inline)
            {
                ++count;
            }
        }
    }
    foreach (var child in view.Children)
    {
        count += CountLines(child);
    }
    return count;
}
```

At first, the root view is obtained. That's the same as the view associated to root element, so GetView is used to get the view of **rtb.Document**. After that, the view tree is traversed counting the lines in each C1FlowView found. Note that you only count the lines with C1Inline elements; otherwise you would also count paragraphs and other container blocks.

Painters

Painters are a way of extending C1RichTextBox by displaying **UIElements** in the same canvas where C1RichTextBox displays text. This allows more general extensions than style overrides, but might be harder to use. Painters are used internally by C1RichTextBox to display the selection.

A painter is an implementation of the IRichTextPainter interface. This interface has two methods, Paint and PaintInline, which are called at different stages of **C1RichTextBox**'s painting pass. Each method receives a C1PaintingContext object that is used to check the viewport being painted, and has the methods that are used for painting custom **UIElements**.

Paint is called each time the entire screen is repainted. Note that each time this method is called, all **UIElements** must be painted by calling Paint, otherwise they will be removed. Passing the same **UIElement** each time is more efficient, as it is not removed from the visual tree.

PaintInline is called for each C1Line that is painted. This method allows finer control over the layer where custom **UIElements** are painted. For instance, it is possible to paint above the text background, but below the text itself. It has the same rule as Paint. All **UIElements** must be painted by calling PaintInline, otherwise they will be removed.

The **Annotations** sample uses painters to display sticky notes. Here is the implementation:

- Visual Basic

```
Class StickyPainter
    Implements IRichTextPainter
    Private _stickies As List(Of StickyNote)

    Public Sub New(stickies As List(Of StickyNote))
        _stickies = stickies
    End Sub

    Public Sub Paint(context As C1PaintingContext)
        For Each sticky As var In _stickies
            Dim rect =
context.ViewManager.GetRectFromPosition(sticky.Range.Start)
            context.Paint(Math.Round(rect.X), Math.Round(rect.Bottom),
False, sticky)
        Next
    End Sub

    Public Sub PaintInline(context As C1PaintingContext, line As
C1Line)
    End Sub

    Public Event PainterChanged As EventHandler(Of
RichTextPainterChangeEventArgs)
End Class
```

- C#

```
class StickyPainter : IRichTextPainter
{
    List<StickyNote> _stickies;

    public StickyPainter(List<StickyNote> stickies)
    {
        _stickies = stickies;
    }

    public void Paint(C1PaintingContext context)
    {
        foreach (var sticky in _stickies)
        {
            var rect =
context.ViewManager.GetRectFromPosition(sticky.Range.Start);
            context.Paint(Math.Round(rect.X), Math.Round(rect.Bottom),
false, sticky);
        }
    }

    public void PaintInline(C1PaintingContext context, C1Line line)
    {
    }

    public event EventHandler<RichTextPainterChangeEventArgs>
PainterChanged;
}
```

StickyPainter only uses the Paint method. For each sticky note it just gets the coordinates inside document and then calls Paint. Note that these are document coordinates; they are independent of paging, scrolling and zooming.

Syntax Coloring

The Understanding C1TextPointer (page 42) section describes 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:

- Visual Basic

```
' Update syntax coloring on a timer
Private _updating As Boolean
Private _syntax As Storyboard

' Start the timer whenever the document changes
Private Sub tb_TextChanged(sender As Object, e As
C1TextChangedEventArgs)
    If Not _updating Then
        ' Create storyboard if it's still null
        If _syntax Is Nothing Then
            _syntax = New Storyboard()
            AddHandler _syntax.Completed, AddressOf _syntax_Completed
            _syntax.Duration = New
Duration(TimeSpan.FromMilliseconds(1000))
        End If

        ' Re-start storyboard
        _syntax.[Stop]()
        _syntax.Seek(TimeSpan.Zero)
        _syntax.Begin()
    End If
End Sub
' Timer elapsed, update syntax coloring
Private Sub _syntax_Completed(sender As Object, e As EventArgs)
    _updating = True
    UpdateSyntaxColoring(_rtb)
    _updating = False
End Sub
```

- C#

```
// 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:

- Visual Basic

```
' Perform syntax coloring
Private Sub UpdateSyntaxColoring(rtb As C1RichTextBox)
    ' Initialize regular expression used to parse HTML
    Dim pattern As String = "</?(?<tagName>[a-zA-Z0-9_:\-]+)" &
"(\s+(?<attName>[a-zA-Z0-9_:\-]+)(?<attValue>(=""[^""]+"")?))*\s*/?>"


    ' Initialize brushes used to color the document
    Dim brDarkBlue As Brush = New SolidColorBrush(Color.FromArgb(255,
0, 0, 180))
    Dim brDarkRed As Brush = New SolidColorBrush(Color.FromArgb(255,
180, 0, 0))
    Dim brLightRed As Brush = New SolidColorBrush(Colors.Red)

    ' Remove old coloring
    Dim input = rtb.Text
    Dim range = rtb.GetTextRange(0, input.Length)
    range.Foreground = rtb.Foreground

    ' Highlight the matches
    For Each m As Match 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
        Dim tagName = m.Groups("tagName")
        range = rtb.GetTextRange(tagName.Index, tagName.Length)
        range.Foreground = brDarkRed

        ' Select attribute names, make them light red
        Dim attGroup = m.Groups("attName")
        If attGroup IsNot Nothing Then
            Dim atts = attGroup.Captures
            For i As Integer = 0 To atts.Count - 1
                Dim att = atts(i)
                range = rtb.GetTextRange(att.Index, att.Length)
                range.Foreground = brLightRed
            Next
        End If
    Next
End Sub
```

- *C#*

```
// 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 parse 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 it may help 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. 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 Syntax Coloring (page 19) section 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 you did in the previous section:

- 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).
- 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.

- Visual Basic

```
Private _rangeStyles As New C1RangeStyleCollection()

Public Sub New()
    InitializeComponent()

    _rtb = New C1RichTextBox()
    LayoutRoot.Children.Add(_rtb)

    AddHandler _rtb.TextChanged, AddressOf 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)
End Sub
```

- C#

```
C1RangeStyleCollection _rangeStyles = new C1RangeStyleCollection();

public MainPage()
{
  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):

- Visual Basic

```
' Perform syntax coloring using StyleOverrides collection
' (takes a fraction of a second to highlight the default document)
Private  Sub UpdateSyntaxColoring(ByVal rtb As C1RichTextBox)
  ' 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
  Dim key As var =  C1TextElement.ForegroundProperty
  Dim brDarkBlue As var =  New C1TextElementStyle()
  brDarkBlue(key) = New SolidColorBrush(Color.FromArgb(255, 0, 0, 180))
  Dim brDarkRed As var =  New C1TextElementStyle()
  brDarkRed(key)  = New SolidColorBrush(Color.FromArgb(255, 180, 0, 0))
```

```
  Dim brLightRed As var =  New C1TextElementStyle()
  brLightRed(key) = New SolidColorBrush(Colors.Red)

  ' Remove old coloring
  _rangeStyles.Clear()

  ' Highlight the matches
  Dim input As var =  rtb.Text
  Dim m As Match
  For Each m In Regex.Matches(input,pattern)
    ' Select whole tag, make it dark blue
    Dim range As var =  rtb.GetTextRange(m.Index,m.Length)
    _rangeStyles.Add(New C1RangeStyle(range,brDarkBlue))

    ' Select tag name, make it dark red
    Dim tagName As var =  m.Groups("tagName")
    range = rtb.GetTextRange(tagName.Index, tagName.Length)
    _rangeStyles.Add(New C1RangeStyle(range,brDarkRed))

    ' Select attribute names, make them light red
    Dim attGroup As var =  m.Groups("attName")
    If Not attGroup Is Nothing Then
      Dim att As Capture
      For Each att In attGroup.Captures
        range = rtb.GetTextRange(att.Index, att.Length)
        _rangeStyles.Add(New C1RangeStyle(range,brLightRed))
      Next
    End If
  Next
End Sub
```

- *C#*

```
// 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 (page 13), 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 ElementMouseLeftButtonDown event:

- Visual Basic

```
Public Sub New()
    ' Default initialization
    InitializeComponent()

    ' Create a C1RichTextBox and add it to the page
```

```
    _rtb = New C1RichTextBox()
    LayoutRoot.Children.Add(_rtb)

    ' Attach event handler
    Add Handler _rtb.ElementMouseLeftButtonDown AddressOf
rtb_ElementMouseLeftButtonDown
End Sub
```

- C#

```
public MainPage()
{
  // Default initialization
  InitializeComponent();

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

  // Attach event handler
  _rtb.ElementMouseLeftButtonDown += rtb_ElementMouseLeftButtonDown;
}
```

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:

- Visual Basic

```
Private Sub _rtb_ElementMouseLeftButtonDown(sender As Object, e As
MouseButtonEventArgs)
    If Keyboard.Modifiers <> 0 Then
        Dim run = TryCast(sender, C1Run)
        If run IsNot Nothing Then
            run.FontWeight = If(run.FontWeight = FontWeights.Bold,
FontWeights.Normal, FontWeights.Bold)
        End If
    End If
End Sub
```

- C#

```
void _rtb_ElementMouseLeftButtonDown(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:

- Visual Basic

```
Private Sub _rtb_ElementMouseLeftButtonDown(sender As Object, e As
MouseButtonEventArgs)
    If Keyboard.Modifiers <> 0 Then
        ' Get position in control coordinates
        Dim pt = e.GetPosition(_rtb)

        ' Get text pointer at position
        Dim pointer = _rtb.GetPositionFromPoint(pt)

        ' Check that the pointer is pointing to a C1Run
        Dim run = TryCast(pointer.Element, C1Run)
        If run IsNot Nothing Then
            ' Get the word within the C1Run
            Dim text = run.Text
            Dim start = pointer.Offset
            Dim [end] = pointer.Offset
            While start > 0 AndAlso Char.IsLetterOrDigit(text, start - 1)
                start -= 1
            End While
            While [end] < text.Length - 1 AndAlso
Char.IsLetterOrDigit(text, [end] + 1)
                [end] += 1
            End While

            ' Toggle the bold property for the run that was clicked
            Dim word = New C1TextRange(pointer.Element, start, [end] -
start + 1)
            word.FontWeight = If(word.FontWeight.HasValue AndAlso
word.FontWeight.Value = FontWeights.Bold, FontWeights.Normal,
FontWeights.Bold)
        End If
    End If
End Sub
```

- C#

```
void _rtb_ElementMouseLeftButtonDown(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.

HtmlFilter Customization

HtmlFilter is the component in C1RichTextBox that transforms HTML strings to C1Document**s** and back. It's also capable of transforming to and from an intermediate representation of an HTML document called C1HtmlDocument.

When transforming between C1HtmlDocument and C1Document, several events are fired allowing customization of each node that is transformed. These events are:

- ConvertingHtmlNode: this event is fired just before an HTML node is transformed. If marked as handled by the event handler then HtmlFilter assumes the node was transformed and skips it.
- ConvertedHtmlNode: this event is fired after a node was transformed. It can be used to make minor changes to the transformation result.
- ConvertingTextElement: this event is fired just before a C1TextElement is transformed. If marked as handled by the event handler then HtmlFilter assumes the element was transformed and skips it.
- ConvertedTextElement: this event is fired after a C1TextElement is transformed. It can be used to make minor changes to the transformation result.

As an example you can see how the **HtmlFilterCustomization** sample adds support for GIF images using C1.Phone.Imaging. It uses both ConvertingHtmlNode and ConvertingTextElement events. Here is the ConvertingHtmlNode event handler:

- Visual Basic

```
Private Sub HtmlFilter_ConvertingHtmlNode(sender As Object, e As
ConvertingHtmlNodeEventArgs)
    Dim htmlElement = TryCast(e.HtmlNode, C1HtmlElement)
    If htmlElement IsNot Nothing AndAlso htmlElement.Name = "img" Then
        Dim src As String
        If htmlElement.Attributes.TryGetValue("src", src) Then
            Dim uri = New Uri("/HtmlFilterCustomization;component/" &
src, UriKind.Relative)
            Dim resource = Application.GetResourceStream(uri)
            If resource IsNot Nothing Then
                Dim imageSource = New
C1Bitmap(resource.Stream).ImageSource
                Dim image = New Image() With { _
                    Key .Source = imageSource _
                }
                SetImageSource(image, src)
```

```
                e.Parent.Children.Add(New C1InlineUIContainer() With {
_
                    Key .Child = image _
                })
                e.Handled = True
            End If
        End If
    End If
End Sub
```

- C#

```
void HtmlFilter_ConvertingHtmlNode(object sender,
ConvertingHtmlNodeEventArgs e)
{
    var htmlElement = e.HtmlNode as C1HtmlElement;
    if (htmlElement != null && htmlElement.Name == "img")
    {
        string src;
        if (htmlElement.Attributes.TryGetValue("src", out src))
        {
            var uri = new Uri("/HtmlFilterCustomization;component/" + src,
UriKind.Relative);
            var resource = Application.GetResourceStream(uri);
            if(resource != null)
            {
                var imageSource = new
C1Bitmap(resource.Stream).ImageSource;
                var image = new Image { Source = imageSource };
                SetImageSource(image, src);
                e.Parent.Children.Add(new C1InlineUIContainer { Child =
image });
                e.Handled = true;
            }
        }
    }
}
```

The first thing the event handler does is cast **e.HtmlNode** to C1HtmlElement. There are two types that inherit from **C1HtmlNode: C1HtmlElement**, which represents an HTML element like `<img/>`, and C1HtmlText, which represents a text node.

Once the C1HtmlNode object has been cast to C1HtmlElement, it's possible to check the tag name, and access its attributes. This is done to see if the element is in fact an IMG tag, and to obtain the SRC attribute. The rest of the code takes core of creating the appropriate element, which is then added to **e.Parent**. Note that the SRC value is saved as an attached property, to be accessed when exporting.

Once the transformation is done, the handler can set **e.Handled** to **True** in order to prevent HtmlFilter from transforming this C1HtmlNode.

The ConvertingTextElement event handler looks like the following:

- Visual Basic

```
Private Sub HtmlFilter_ConvertingTextElement(sender As Object, e As
ConvertingTextElementEventArgs)
    Dim inlineContainer = TryCast(e.TextElement, C1InlineUIContainer)
    If inlineContainer IsNot Nothing Then
        Dim src = GetImageSource(inlineContainer.Child)
        If src IsNot Nothing Then
            Dim element = New C1HtmlElement("img")
```

```
            element.Attributes("src") = src
            e.Parent.Add(element)
            e.Handled = True
        End If
    End If
End Sub
```

- C#

```
void HtmlFilter_ConvertingTextElement(object sender,
ConvertingTextElementEventArgs e)
{
    var inlineContainer = e.TextElement as C1InlineUIContainer;
    if (inlineContainer != null)
    {
        var src = GetImageSource(inlineContainer.Child);
        if (src != null)
        {
            var element = new C1HtmlElement("img");
            element.Attributes["src"] = src;
            e.Parent.Add(element);
            e.Handled = true;
        }
    }
}
```

This is pretty similar to the other handler, only it transforms a C1TextElement to a C1HtmlElement. Note that the SRC value is recovered from the attached property, and a C1HtmlElement is created with that attribute. As before, the new element is added to **e.Parent**, and the event is marked as **Handled**.

Using the Application Bar

ComponentOne RichTextBox for Windows phone includes a **C1.Phone.RichTextBox.ApplicationBar** assembly containing an application bar that you can add buttons to for use in your application. There is no visible application bar control; however, you can add the C1 Tools included in the assembly to the application bar. All you need to do is specify the **IconUri** path for the button icons and set each image's **Build Action** property to **Content**.

To add buttons to the application bar, follow these steps:

1. Create an Images folder containing images to be used for the button icons.
2. In the Solution Explorer, click **Show All Files**, right-click the **Images** folder, and select **Include In Project**.
3. Select each image and in the Visual Studio Properties window, set the **Build Action** to **Content**.
4. Add a C1RichTextBox control to the page.
5. In the Solution Explorer, right-click the **References** folder, select **Add Reference** and select **C1.Phone.RichTextBox.ApplicationBar.dll**.
6. Add the following namespace to the list of namespaces at the top of the XAML page:

   xmlns:c1="clr-namespace:C1.Phone.RichTextBox;assembly=C1.Phone.RichTextBox.ApplicationBar"

7. Add XAML markup for the ApplicationBar and ApplicationBarIconButtons. Notice that the first button is simply an ApplicationBarIconButton and the second button creates a C1ItalicTool.

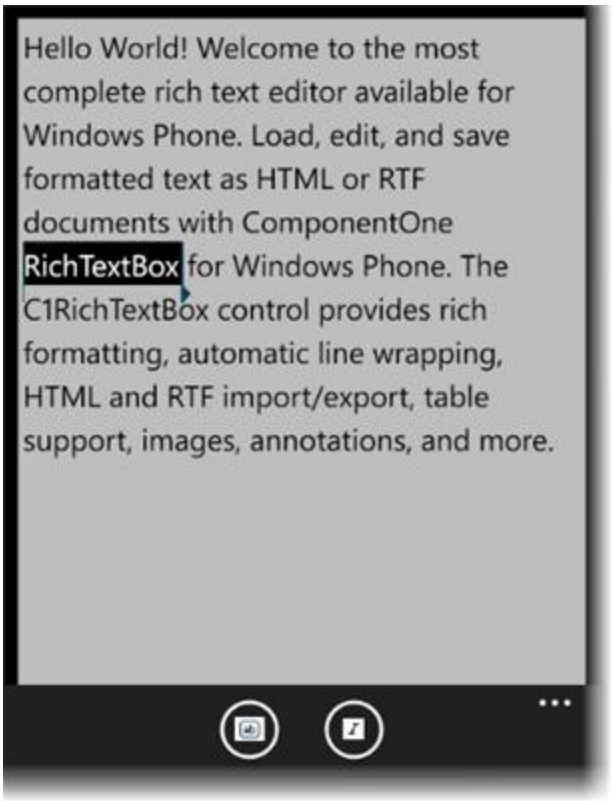
```
<!--Sample code showing usage of ApplicationBar-->
    <phone:PhoneApplicationPage.ApplicationBar>
      <shell:ApplicationBar>
```

```
        <shell:ApplicationBarIconButton
IconUri="Images/appbar_button1.png" Text="Button 1"/>
            <c1:C1ItalicTool IconUri="Images/ItalicButton.png" />

            </shell:ApplicationBar>
    </phone:PhoneApplicationPage.ApplicationBar>
```

8. Select the C1ItalicTool XAML, and in the Properties window, set the **RichTextBox** property to **C1RichTextBox**.
9. Select **View | Code** to switch to code view, and enter the following code so that when a user taps the italic button, italics are applied to the selected text:

```
public MainPage()
{
    InitializeComponent();

((C1ItalicTool)this.ApplicationBar.Buttons[1]).RichTextBox = c1RichTextBox
1;
}
```

10. Press **F5** to run the application, select some text, and tap the italics button.



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 Microsoft 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.Phone** and **C1.Phone.RichTextBox** assemblies. Then edit the page constructor as follows:

- Visual Basic

```
Imports C1.Phone
Imports C1.Phone.RichTextBox
Imports C1.Phone.RichTextBox.Documents

Public Partial Class MainPage
    Inherits UserControl
    ' C1RichTextBox that will display the C1Document
    Private _rtb As C1RichTextBox

    Public Sub New()
        ' 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(GetType(C1RichTextBox).Assembly)
        _rtb.IsReadOnly = True
    End Sub
End Class
```

- C#

```
using C1.Phone;
using C1.Phone.RichTextBox;
using C1.Phone.RichTextBox.Documents;

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

  public MainPage()
  {
    // 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:

- Visual Basic

```
Private Function DocumentAssembly(asm As Assembly) As C1Document
    ' Create document
    Dim doc As New C1Document()
    doc.FontFamily = New FontFamily("Tahoma")

    ' Assembly
    doc.Blocks.Add(New Heading1("Assembly" & vbCr & vbLf +
asm.FullName.Split(","C)(0)))

    ' Types
    For Each t As Type In asm.GetTypes()
        DocumentType(doc, t)
    Next

    ' Done
    Return doc
End Function
```

- C#

```
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:

- Visual Basic

```
Class Normal
    Inherits C1Paragraph
    Public Sub New(text As String)
        Me.Inlines.Add(New C1Run() With { _
            Key .Text = text _
        })
        Me.Padding = New Thickness(30, 0, 0, 0)
        Me.Margin = New Thickness(0)
    End Sub
End Class
```

- C#

```
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:

- Visual Basic

```
Class Heading
    Inherits Normal
    Public Sub New(text As String)
        MyBase.New(text)
        Me.FontWeight = FontWeights.Bold
    End Sub
End Class
```

- C#

```
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:

- Visual Basic

```
Class Heading1
    Inherits Heading
    Public Sub New(text As String)
        MyBase.New(text)
```

```
        Me.Background = New SolidColorBrush(Colors.Yellow)
        Me.FontSize = 24
        Me.Padding = New Thickness(0, 10, 0, 10)
        Me.BorderBrush = New SolidColorBrush(Colors.Black)
        Me.BorderThickness = New Thickness(3, 1, 1, 0)
    End Sub
End Class
Class Heading2
    Inherits Heading
    Public Sub New(text As String)
        MyBase.New(text)
        Me.FontSize = 18
        Me.FontStyle = FontStyles.Italic
        Me.Background = New SolidColorBrush(Colors.Yellow)
        Me.Padding = New Thickness(10, 5, 0, 5)
        Me.BorderBrush = New SolidColorBrush(Colors.Black)
        Me.BorderThickness = New Thickness(3, 1, 1, 1)
    End Sub
End Class
Class Heading3
    Inherits Heading
    Public Sub New(text As String)
        MyBase.New(text)
        Me.FontSize = 14
        Me.Background = New SolidColorBrush(Colors.LightGray)
        Me.Padding = New Thickness(20, 3, 0, 0)
    End Sub
End Class
Class Heading4
    Inherits Heading
    Public Sub New(text As String)
        MyBase.New(text)
        Me.FontSize = 14
        Me.Padding = New Thickness(30, 0, 0, 0)
    End Sub
End Class
```

- C#

```
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:

- Visual Basic

```
Private Sub DocumentType(doc As C1Document, t As Type)
    ' Skip non-public/generic
    If Not t.IsPublic OrElse t.ContainsGenericParameters Then
        Return
    End If

    ' Type
    doc.Blocks.Add(New Heading2("Class " & Convert.ToString(t.Name)))

    ' Properties
    doc.Blocks.Add(New Heading3("Properties"))
    For Each pi As PropertyInfo In t.GetProperties()
        If pi.DeclaringType = t Then
            DocumentProperty(doc, pi)
        End If
    Next

   ' Methods
        doc.Blocks.Add(New Heading3("Methods"))
    For Each mi As MethodInfo In t.GetMethods()
        If mi.DeclaringType = t Then
            DocumentMethod(doc, mi)
        End If
    Next

     ' Events
     doc.Blocks.Add(New Heading3("Events"))
     For Each ei As EventInfo In t.GetEvents()
        If ei.DeclaringType = t Then
            DocumentEvent(doc, ei)
        End If
    Next
```

```
End Sub
```

- C#

```
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:

- Visual Basic

```
Private Sub DocumentProperty(doc As C1Document, pi As PropertyInfo)
    If pi.PropertyType.ContainsGenericParameters Then
        Return
    End If

    doc.Blocks.Add(New Heading4(pi.Name))

    Dim text = String.Format("public {0} {1} {{ {2}{3} }}",
pi.PropertyType.Name, pi.Name, If(pi.CanRead, "get; ", String.Empty),
If(pi.CanWrite, "set; ", String.Empty))
    doc.Blocks.Add(New Normal(text))
End Sub
```

- C#

```
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:

- Visual Basic

```
Private Sub DocumentMethod(doc As C1Document, mi As MethodInfo)
    If mi.IsSpecialName Then
        Return
    End If

    doc.Blocks.Add(New Heading4(mi.Name))
    Dim parms = New StringBuilder()
    For Each parm As var In mi.GetParameters()
        If parms.Length > 0 Then
            parms.Append(", ")
        End If
        parms.AppendFormat("{0} {1}", parm.ParameterType.Name,
parm.Name)
    Next
    Dim text = String.Format("public {0} {1}({2})", mi.ReturnType.Name,
mi.Name, parms.ToString())

    doc.Blocks.Add(New Normal(text))
End Sub

Private Sub DocumentEvent(doc As C1Document, ei As EventInfo)
    doc.Blocks.Add(New Heading4(ei.Name))

    Dim text = String.Format("public {0} {1}",
ei.EventHandlerType.Name, ei.Name)

    doc.Blocks.Add(New Normal(text))
End Sub
```

- C#

```
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));
}
```

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:

- Visual Basic

```
Public Sub New()
    ' 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(GetType(C1RichTextBox).Assembly)
    _rtb.IsReadOnly = True

    ' Attach event handler
    AddHandler _rtb.ElementMouseLeftButtonDown, AddressOf
_rtb_ElementMouseLeftButtonDown

    ' Add a second C1RichTextBox to the page
    LayoutRoot.RowDefinitions.Add(New RowDefinition())
    LayoutRoot.RowDefinitions.Add(New RowDefinition())
    Dim rtb2 = New C1RichTextBox()
    rtb2.SetValue(Grid.RowProperty, 1)
    LayoutRoot.Children.Add(rtb2)

    ' Bind the second C1RichTextBox to the same document
    rtb2.Document = _rtb.Document
End Sub
```

- C#

```
public MainPage()
{
  // 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.ElementMouseLeftButtonDown += _rtb_ElementMouseLeftButtonDown;

  // 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 that 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 you wanted to add some functionality to the previous sample: when the user presses the CTRL key, you 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:

- Visual Basic

```
Public Sub New()
    ' Default initialization
    InitializeComponent()

    ' No changes here...

    ' Bind the second C1RichTextBox to the same document
    rtb2.Document = _rtb.Document
    AddHandler rtb2.KeyDown, AddressOf rtb2_KeyDown
End Sub
Private Sub rtb2_KeyDown(sender As Object, e As KeyEventArgs)
    If e.Key = Key.Ctrl Then
        For Each heading2 As var In _rtb.Document.Blocks.OfType(Of
Heading2)()
            Dim text = heading2.ContentRange.Text
            heading2.ContentRange.Text = text.ToUpper()
        Next
    End If
End Sub
```

- C#

```
public MainPage()
{
  // 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();
        }
    }
}
```

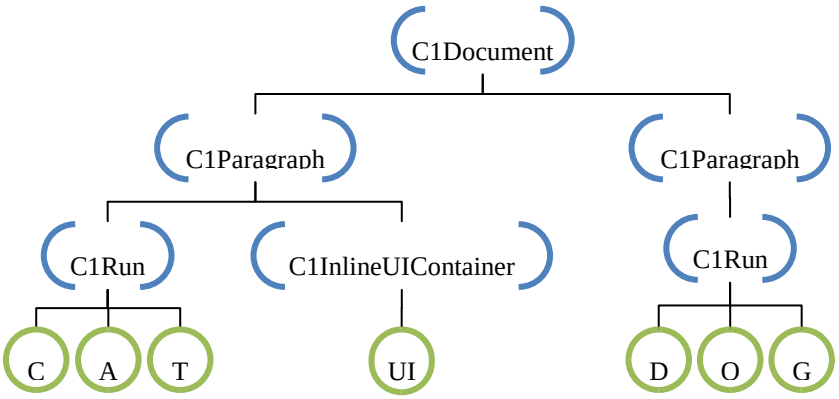
The code monitors the keyboard. When the user presses the CTRL key, it enumerates all **Heading2** elements in the document and replaces their content with capitals.

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, and so on. As an example, let's say you want to get the C1TextRange for the word contained under a C1TextPointer. You can do the following:

- Visual Basic

```
Private Function ExpandToWord(pos As C1TextPointer) As C1TextRange
    ' Find word start
    Dim wordStart = If(pos.IsWordStart, pos,
pos.Enumerate(LogicalDirection.Backward).First(Function(p)
p.IsWordStart))

    ' Find word end
    Dim wordEnd = If(pos.IsWordEnd, pos,
pos.Enumerate(LogicalDirection.Forward).First(Function(p) p.IsWordEnd))

    ' Return new range from word start to word end
    Return New C1TextRange(wordStart, wordEnd)
End Function
```

- C#

```
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 the code first checks if the parameter position is a word start, and if not, searches backward for a position that is a word start. Likewise for the word end, it checks the parameter position and then searches 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:

- Visual Basic
```
document.ContentStart.Enumerate().Count(Function(p) p.IsWordStart
AndAlso TypeOf p.Symbol Is Char)
```
- C#
```
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:

- Visual Basic
```
Private Function FindWordFromPosition(position As C1TextPointer, word
As String) As C1TextRange
    ' Get all ranges whose text length is equal to word.Length
    Dim ranges = position.Enumerate().[Select](Function(pos)
    ' Get a position that is at word.Length offset
    ' but ignoring tags that do not change the text flow
    Dim [end] = pos.GetPositionAtOffset(word.Length,
C1TextRange.TextTagFilter)
    Return New C1TextRange(pos, [end])

End Function)
    ' returned value will be null if word is not found.
    Return ranges.FirstOrDefault(Function(range) range.Text = word)
End Function
```
- C#
```
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:

- Visual Basic

```
Dim wordRange = FindWordFromPosition(document.ContentStart, "cat")
If wordRange IsNot Nothing Then
    wordRange.Text = "dog"
End If
```

- C#

```
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.

Elements Supported in RichTextBox

The following sections detail what HTML 4 elements, HTML attributes, CSS2 properties, and CSS selectors are supported in **RichTextBox for Windows Phone**.

HTML Elements

RichTextBox for Windows Phone supports several HTML 4 elements. The following table lists HTML elements by names and notes whether they are supported by **RichTextBox for Windows Phone**.

Name	Supported
A	Yes
ABBR	Yes
ACRONYM	Yes
ADDRESS	Yes
APPLET	No
AREA	No
B	Yes
BASE	No
BASEFONT	Yes
BDO	No
BIG	Yes
BLOCKQUOTE	Yes
BODY	Yes
BR	Yes

BUTTON	No
CAPTION	No
CENTER	Yes
CITE	Yes
CODE	Yes
COL	Yes
COLGROUP	Yes
DD	Yes
DEL	Yes
DFN	Yes
DIR	Yes
DIV	Yes
DL	Yes
DT	Yes
EM	Yes
FIELDSET	No
FONT	Yes
FORM	No
FRAME	No
FRAMESET	No
H1	Yes
H2	Yes
H3	Yes
H4	Yes
H5	Yes
H6	Yes
HEAD	Yes
HR	Yes
HTML	Yes
I	Yes
IFRAME	No
IMG	Yes

INPUT	No
INS	Yes
ISINDEX	No
KBD	Yes
LABEL	Yes
LEGEND	No
LI	Yes
LINK	No
MAP	No
MENU	Yes
META	No
NOFRAMES	No
NOSCRIPT	No
OBJECT	No
OL	Yes
OPTGROUP	No
OPTION	No
P	Yes
PARAM	No
PRE	Yes
Q	No
S	Yes
SAMP	Yes
SCRIPT	No
SELECT	No
SMALL	Yes
SPAN	Yes
STRIKE	Yes
STRONG	Yes
STYLE	Yes
SUB	Yes
SUP	Yes

TABLE	Yes
TBODY	Yes
TD	Yes
TEXTAREA	No
TFOOT	Yes
TH	Yes
THEAD	Yes
TITLE	No
TR	Yes
TT	Yes
U	Yes
UL	Yes
VAR	Yes

HTML Attributes

RichTextBox for Windows Phone supports several HTML 4 attribute. The following table lists HTML attributes by name and element and notes whether they are supported by **RichTextBox for Windows Phone**.

Name	Elements	Supported
abbr	TD, TH	No
accept-charset	FORM	No
accept	FORM, INPUT	No
accesskey	A, AREA, BUTTON, INPUT, LABEL, LEGEND, TEXTAREA	No
action	FORM	No
align	CAPTION	No
align	APPLET, IFRAME, IMG, INPUT, OBJECT	No
align	LEGEND	No
align	TABLE	No
align	HR	No
align	DIV, H1, H2, H3, H4, H5, H6, P	Yes
align	COL, COLGROUP, TBODY, TD, TFOOT, TH, THEAD, TR	No
alink	BODY	No
alt	APPLET	No
alt	AREA, IMG	No

alt	INPUT	No
archive	APPLET	No
archive	OBJECT	No
axis	TD, TH	No
background	BODY	No
bgcolor	TABLE	Yes
bgcolor	TR	Yes
bgcolor	TD, TH	Yes
bgcolor	BODY	Yes
border	TABLE	Yes
border	IMG, OBJECT	Yes
cellpadding	TABLE	No
cellspacing	TABLE	Yes
char	COL, COLGROUP, TBODY, TD, TFOOT, TH, THEAD, TR	No
charoff	COL, COLGROUP, TBODY, TD, TFOOT, TH, THEAD, TR	No
charset	A, LINK, SCRIPT	No
checked	INPUT	No
cite	BLOCKQUOTE, Q	No
cite	DEL, INS	No
class	All elements but BASE, BASEFONT, HEAD, HTML, META, PARAM, SCRIPT, STYLE, TITLE	Yes
classid	OBJECT	No
clear	BR	No
code	APPLET	No
codebase	OBJECT	No
codebase	APPLET	No
codetype	OBJECT	No
color	BASEFONT, FONT	No
cols	FRAMESET	No
cols	TEXTAREA	No
colspan	TD, TH	Yes
compact	DIR, DL, MENU, OL, UL	No
content	META	No

coords	AREA	No
coords	A	No
data	OBJECT	No
datetime	DEL, INS	No
declare	OBJECT	No
defer	SCRIPT	No
dir	All elements but APPLET, BASE, BASEFONT, BDO, BR, FRAME, FRAMESET, IFRAME, PARAM, SCRIPT	No
dir	BDO	No
disabled	BUTTON, INPUT, OPTGROUP, OPTION, SELECT, TEXTAREA	No
enctype	FORM	No
face	BASEFONT, FONT	Yes
for	LABEL	No
frame	TABLE	Yes
frameborder	FRAME, IFRAME	No
headers	TD, TH	No
height	IFRAME	No
height	TD, TH	No
height	IMG, OBJECT	Yes
height	APPLET	No
href	A, AREA, LINK	Yes
href	BASE	No
hreflang	A, LINK	No
hspace	APPLET, IMG, OBJECT	Yes
http-equiv	META	No
id	All elements but BASE, HEAD, HTML, META, SCRIPT, STYLE, TITLE	Yes
ismap	IMG, INPUT	No
label	OPTION	No
label	OPTGROUP	No
lang	All elements but APPLET, BASE, BASEFONT, BR, FRAME, FRAMESET, IFRAME, PARAM, SCRIPT	No
language	SCRIPT	No

link	BODY	No
longdesc	IMG	No
longdesc	FRAME, IFRAME	No
marginheight	FRAME, IFRAME	No
marginwidth	FRAME, IFRAME	No
maxlength	INPUT	No
media	STYLE	No
media	LINK	No
method	FORM	No
multiple	SELECT	No
name	BUTTON, TEXTAREA	No
name	APPLET	No
name	SELECT	No
name	FORM	No
name	FRAME, IFRAME	No
name	IMG	No
name	A	No
name	INPUT, OBJECT	No
name	MAP	No
name	PARAM	No
name	META	No
nohref	AREA	No
noresize	FRAME	No
noshade	HR	No
nowrap	TD, TH	No
object	APPLET	No
onblur	A, AREA, BUTTON, INPUT, LABEL, SELECT, TEXTAREA	No
onchange	INPUT, SELECT, TEXTAREA	No
onclick	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
ondblclick	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No

onfocus	A, AREA, BUTTON, INPUT, LABEL, SELECT, TEXTAREA	No
onkeydown	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onkeypress	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onkeyup	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onload	FRAMESET	No
onload	BODY	No
onmousedown	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onmousemove	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onmouseout	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onmouseover	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onmouseup	All elements but APPLET, BASE, BASEFONT, BDO, BR, FONT, FRAME, FRAMESET, HEAD, HTML, IFRAME, ISINDEX, META, PARAM, SCRIPT, STYLE, TITLE	No
onreset	FORM	No
onselect	INPUT, TEXTAREA	No
onsubmit	FORM	No
onunload	FRAMESET	No
onunload	BODY	No
profile	HEAD	No
prompt	ISINDEX	No
readonly	TEXTAREA	No
readonly	INPUT	No
rel	A, LINK	No

rev	A, LINK	No
rows	FRAMESET	No
rows	TEXTAREA	No
rowspan	TD, TH	Yes
rules	TABLE	Yes
scheme	META	No
scope	TD, TH	No
scrolling	FRAME, IFRAME	No
selected	OPTION	No
shape	AREA	No
shape	A	No
size	HR	No
size	FONT	No
size	INPUT	No
size	BASEFONT	No
size	SELECT	No
span	COL	No
span	COLGROUP	No
src	SCRIPT	No
src	INPUT	No
src	FRAME, IFRAME	No
src	IMG	No
standby	OBJECT	No
start	OL	Yes
style	All elements but BASE, BASEFONT, HEAD, HTML, META, PARAM, SCRIPT, STYLE, TITLE	Yes
summary	TABLE	No
tabindex	A, AREA, BUTTON, INPUT, OBJECT, SELECT, TEXTAREA	No
target	A, AREA, BASE, FORM, LINK	No
text	BODY	No
title	All elements but BASE, BASEFONT, HEAD, HTML, META, PARAM, SCRIPT, TITLE	Yes
type	A, LINK	No

type	OBJECT	No
type	PARAM	No
type	SCRIPT	No
type	STYLE	No
type	INPUT	No
type	LI	No
type	OL	No
type	UL	No
type	BUTTON	No
usemap	IMG, INPUT, OBJECT	No
valign	COL, COLGROUP, TBODY, TD, TFOOT, TH, THEAD, TR	Yes
value	INPUT	No
value	OPTION	No
value	PARAM	No
value	BUTTON	No
value	LI	No
valuetype	PARAM	No
version	HTML	No
vlink	BODY	No
vspace	APPLET, IMG, OBJECT	Yes
width	HR	No
width	IFRAME	No
width	IMG, OBJECT	Yes
width	TABLE	No
width	TD, TH	No
width	APPLET	No
width	COL	Yes
width	COLGROUP	No
width	PRE	No

CSS2 Properties

RichTextBox for Windows Phone supports several CSS2 properties. The following table lists CSS2 properties by name and media group and notes whether they are supported by **RichTextBox for Windows Phone**.

Name	Media groups	Supported	Comments
azimuth	aural	No	
background-attachment	visual	No	
background-color	visual	Yes	
background-image	visual	Yes	The image is not repeated.
background-position	visual	No	
background-repeat	visual	No	
background	visual	Yes	Only supports color and image.
border-collapse	visual	Yes	
border-color	visual	Yes	
border-spacing	visual	Yes	
border-style	visual	Yes	Supports 'none', 'hidden', and solid. Other values are treated as solid.
border-top border-right border-bottom border-left	visual	Yes	
border-top-color border-right-color border-bottom-color border-left-color	visual	Yes	
border-top-style border-right-style border-bottom-style border-left-style	visual	Yes	
border-top-width border-right-width border-bottom-width border-left-width	visual	Yes	
border-width	visual	Yes	
border	visual	Yes	
bottom	visual	No	
caption-side	visual	No	
clear	visual	No	
clip	visual	No	
color	visual	Yes	
content	all	No	

counter-increment	all	No	
counter-reset	all	No	
cue-after	aural	No	
cue-before	aural	No	
cue	aural	No	
cursor	visual, interactive	Yes	All values except crosshair, move, progress, help, and <uri>.
direction	visual	No	
display	all	Yes	All values except run-in, inline-block, inline-table, and table-caption.
elevation	aural	No	
empty-cells	visual	No	
float	visual	No	
font-family	visual	Yes	
font-size	visual	Yes	
font-style	visual	Yes	
font-variant	visual	No	
font-weight	visual	Yes	
font	visual	Yes	
height	visual	Yes	Only in img elements.
left	visual	No	
letter-spacing	visual	No	
line-height	visual	Yes	
list-style-image	visual	Yes	
list-style-position	visual	No	
list-style-type	visual	Yes	All values except georgian, armenian and, lower-greek.
list-style	visual	Yes	
margin-right margin-left	visual	Yes	
margin-top margin-bottom	visual	Yes	
margin	visual	Yes	

max-height	visual	No	
max-width	visual	No	
min-height	visual	No	
min-width	visual	No	
orphans	visual, paged	No	
outline-color	visual, interactive	No	
outline-style	visual, interactive	No	
outline-width	visual, interactive	No	
outline	visual, interactive	No	
overflow	visual	No	
padding-top padding-right padding-bottom padding-left	visual	Yes	
padding	visual	Yes	
page-break-after	visual, paged	No	
page-break-before	visual, paged	No	
page-break-inside	visual, paged	No	
pause-after	aural	No	
pause-before	aural	No	
pause	aural	No	
pitch-range	aural	No	
pitch	aural	No	
play-during	aural	No	
position	visual	No	
quotes	visual	No	
richness	aural	No	
right	visual	No	
speak-header	aural	No	
speak-numeral	aural	No	
speak-punctuation	aural	No	

speak	aural	No	
speech-rate	aural	No	
stress	aural	No	
table-layout	visual	No	
text-align	visual	Yes	
text-decoration	visual	Yes	
text-indent	visual	Yes	
text-transform	visual	No	
top	visual	No	
unicode-bidi	visual	No	
vertical-align	visual	Yes	All values except <percentage> and <length>.
visibility	visual	Yes	
voice-family	aural	No	
volume	aural	No	
white-space	visual	Yes	Nowrap and pre are treated like normal and pre-wrap.
widows	visual, paged	No	
width	visual	Yes	Only in img elements.
word-spacing	visual	No	
z-index	visual	No	

CSS2 Selectors

RichTextBox for Windows Phone supports several CSS2 selectors. The following table lists CSS2 selectors by pattern and CSS level and notes whether they are supported by **RichTextBox for Windows Phone**.

Pattern	CSS level	Supported
*	2	Yes
E	1	Yes
E[foo]	2	Yes
E[foo="bar"]	2	Yes
E[foo~="bar"]	2	Yes
E[foo^="bar"]	3	Yes

E[foo$="bar"]	3	Yes
E[foo*="bar"]	3	Yes
E[foo|="en"]	2	Yes
E:root	3	No
E:nth-child(n)	3	No
E:nth-last-child(n)	3	No
E:nth-of-type(n)	3	No
E:nth-last-of-type(n)	3	No
E:first-child	2	No
E:last-child	3	No
E:first-of-type	3	No
E:last-of-type	3	No
E:only-child	3	No
E:only-of-type	3	No
E:empty	3	No
E:link	1	No
E:visited	1	No
E:active	2	No
E:hover	2	No
E:focus	2	No
E:target	3	No
E:lang(fr)	2	No
E:enabled	3	No
E:disabled	3	No
E:checked	3	No
E::first-line	1	No
E::first-letter	1	No
E::before	2	No
E::after	2	No
E.warning	1	No
E#myid	1	Yes
E:not(s)	3	No

E F	1	Yes
E > F	2	Yes
E + F	2	Yes
E ~ F	3	Yes

RichTextBox for Windows Phone Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the C1RichTextBox control in general. If you are unfamiliar with the **ComponentOne RichTextBox for Windows Phone** product, please see the RichTextBox for Windows Phone Quick Start (page 3) first.

Each topic in this section provides a solution for specific tasks using the **ComponentOne RichTextBox for Windows Phone** product.

Each task-based help topic also assumes that you have created a new Windows Phone project.

Setting the Text Content

The Text property determines the text content of the C1RichTextBox control. By default the C1RichTextBox control starts blank and without content but you can customize this at design time, in XAML, and in code. Note that you can also set HTML markup as the control's content. See Setting the HTML Content (page 61) for details.

At Design Time

To set the Text property, complete the following steps:

1. Click the C1RichTextBox control once to select it.
2. Navigate to the Properties window, and enter text, for example "Hello World!", in the text box next to the Text property.

   This will set the Text property to the value you chose.

In XAML

For example, to set the Text property add `Text="Hello World!"` to the `<c1rtb:C1RichTextBox>` tag so that it appears similar to the following:

```
<c1rtb:C1RichTextBox HorizontalAlignment="Left" Margin="10,10,0,0"
Name="C1RichTextBox1" VerticalAlignment="Top" Height="83" Width="208"
Text="Hello World!" />
```

In Code

For example, to set the Text property add the following code to your project:

- Visual Basic

```
C1RichTextBox1.Text = "Hello World!"
```

- C#

```
c1RichTextBox1.Text = "Hello World!";
```

What You've Accomplished

You've set the text content of the C1RichTextBox control. Run your application and observe that initially "Hello World!" (or the text you chose) will appear in the control:

Note that you can also set HTML markup as the control's content. See Setting the HTML Content (page 61) for details.

Setting the HTML Content

The Html property determines the HTML markup content of the C1RichTextBox control. By default the C1RichTextBox control starts blank and without content but you can customize this at design time, in XAML, and in code. Note that you can also set text as the control's content. See Setting the Text Content (page 60) for details.

At Design Time

To set the Html property, complete the following steps:

1. Click the C1RichTextBox control once to select it.
2. Navigate to the Properties window, and enter text, for example "<h1>Hello World!</h1>", in the text box next to the Html property.

   This will set the Html property to the value you chose.

In XAML

For example, to set the Html property add `Html="Hello World!"` to the `<c1rtb:C1RichTextBox>` tag so that it appears similar to the following:

```
<c1rtb:C1RichTextBox HorizontalAlignment="Left" Margin="10,10,0,0"
Name="C1RichTextBox1" VerticalAlignment="Top" c1:C1NagScreen.Nag="True"
Height="83" Width="208" Html="&lt;h1&gt;Hello World!&lt;/h1&gt;" />
```

In Code

For example, to set the Html property add the following code to your project:

- Visual Basic

```
Me.C1RichTextBox1.Html = "&lt;h1&gt;Hello World!&lt;/h1&gt;"
```

- C#

```
this.c1RichTextBox1.Html = "&lt;h1&gt;Hello World!&lt;/h1&gt;";
```

What You've Accomplished

You've set the text content of the C1RichTextBox control. Note that you had to represent angle brackets (< >) as "<" and ">". Run your application and observe that initially "Hello World!" (or the text you chose) will appear in the control in large text:

Note that you can also set text as the control's content. See Setting the Text Content (page 60) for details. For an example of adding a hyperlink to the content of the C1RichTextBox control, see the Hyperlinks (page 13) topic.

Implementing a Simple Formatting Toolbar

While you can use the C1RichTextBox control to add a full toolbar to use with the C1RichTextBox control, you can also easily create your own simple 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, and so on.

Implementing a simple toolbar with the C1RichTextBox is easy. For example, complete the following steps:

1. In the Solution Explorer, right-click the project and, from the context menu, choose **Add Reference**.
2. In the **Add Reference** dialog box, select the **C1.Phone.RichTextBox** assembly and click **OK**.
3. Update the XAML on the page, so it appears similar to the following:

```
<UserControl x:Class="C1RichTextBoxIntro.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1rtb=
"clr-namespace:C1.Phone.RichTextBox;assembly=C1.Phone.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>
    <c1rtb:C1RichTextBox x:Name="_rtb" Grid.Row="1"
       AcceptsReturn="True"
       SelectionChanged="_rtb_SelectionChanged"/>
    </Grid>
</UserControl>
```

   This markup adds a C1RichTextBox control and three buttons (bold, italic, and underline) that control its formatting. When a button is clicked, the attached event handlers are responsible for updating the format of the selection. The code in the next step accomplishes that.

4. Right-click the page and choose **View Code** to switch to the Code Editor.
5. Add the following code to your application:

   - Visual Basic

```
Private Sub _btnBold_Click(sender As Object, e As RoutedEventArgs)
    Dim fw As System.Nullable(Of FontWeight) =
_rtb.Selection.FontWeight
    _rtb.Selection.FontWeight = If(fw.HasValue AndAlso fw.Value =
FontWeights.Bold, FontWeights.Normal, FontWeights.Bold)
End Sub
```

   - C#

```
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 is bold; otherwise, the code sets the font weight to "bold".

6. Add the following code to initialize the italics button:

- Visual Basic

```
Private Sub _btnItalic_Click(sender As Object, e As RoutedEventArgs)
    Dim fs As System.Nullable(Of FontStyle) = _rtb.Selection.FontStyle
    _rtb.Selection.FontStyle = If(fs.HasValue AndAlso fs.Value =
FontStyles.Italic, FontStyles.Normal, FontStyles.Italic)
End Sub
```

- C#

```
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;
}
```

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

7. Add the following code to initialize the underline button:

- Visual Basic

```
Private Sub _btnUnderline_Click(sender As Object, e As RoutedEventArgs)
    If _btnUnderline.IsChecked.HasValue Then
        rtb.Selection.TextDecorations =
If(_btnUnderline.IsChecked.Value, C1TextDecorations.Underline, Nothing)
    End If
End Sub
```

- C#

```
private void _btnUnderline_Click(object sender, RoutedEventArgs e)
{
    if (_btnUnderline.IsChecked.HasValue)
    {
        _rtb.Selection.TextDecorations = _btnUnderline.IsChecked.Value
          ? C1TextDecorations.Underline
          : null;
    }
}
```

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. The above code is all it takes to make the three buttons work.

8. Add the following code to implement the event handler for the SelectionChanged event:

- Visual Basic

```
Private Sub _rtb_SelectionChanged(sender As Object, e As EventArgs)
    Dim fw As System.Nullable(Of FontWeight) =
_rtb.Selection.FontWeight
    _btnBold.IsChecked = fw.HasValue AndAlso fw.Value =
FontWeights.Bold

    Dim fs As System.Nullable(Of FontStyle) = _rtb.Selection.FontStyle
    _btnItalic.IsChecked = fs.HasValue AndAlso fs.Value =
FontStyles.Italic

    _btnUnderline.IsChecked = (sel.TextDecorations IsNot Nothing)
End Sub
```

- C#

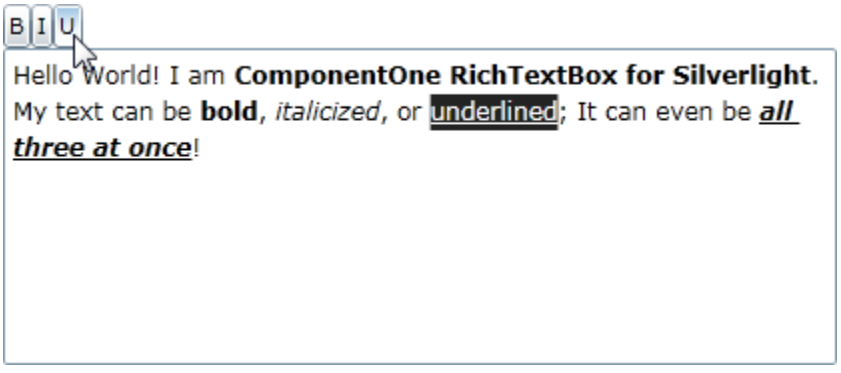
```
void _rtb_SelectionChanged(object sender, EventArgs e)
{
  var sel = _rtb.Selection;

  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;

  _btnUnderline.IsChecked = (sel.TextDecorations != null);
}
```

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.

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

What You've Accomplished

You've created a simple toolbar. When run, the application would appear similar to the following image:



You can enter text and press the bold, italic, and underline buttons to format the text as in the above image. A full toolbar would include more buttons and controls, and would handle them in a similar fashion.