ComponentOne

Editor for WinForms

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

Table of Contents

Editor for WinForms Overview

Load, save, and edit your XHTML documents with **Editor for WinForms**. Choose from a WYSIWYG **Design** mode, **Source** mode, or **Preview** mode to edit or view your documents. The C1Editor control is bound to the XHTML document, so you can make changes directly to your XHTML document from within the control. Editing XHTML documents has never been so easy!

Getting Started

To get started, review the following topics:

- Key Features
- Quick Start
- Samples

Help with WinForms Edition

Getting Started

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

Key Features

The following are some of the main features of C1Editor that you may find useful:

- **"On the fly" synchronizing with XmlDocument**

  Bind the C1Editor control to a document specified in the Document property. If the document is edited within the C1Editor, the underlying XmlDocument syncs to match it. If the XmlDocument changes in code, these changes are visible in the C1Editor control at run time. See Binding C1Editor to a Document for more information.

- **C1Editor provides three edit modes: Design, Source, and Preview**

  The C1Editor control features three editor modes: **Design**, **Source**, and **Preview**. You can determine which of these views users will see initially by setting the Mode property. See the C1Editor Overview for more information.

- **Load and save Xhtml documents from or to a file or stream**

  You can load an XHTML document into C1Editor from a file, stream or XML string. You can save an XHTML document to a file or stream. See the Editor for WinForms Task-Based Help for examples on how to do this.

- **Editor for WinForms offers Cascading Style Sheet support, including easy-to-define custom CSS styles for Design and Preview mode**

  C1Editor fully supports cascading style sheets (CSS) in edited documents. In addition, you can specify external CSS rules in CSS files which will be used only in **Design** or **Preview** mode. The LoadDesignCSS and LoadPreviewCSS methods support this feature by loading a cascading style sheet from a file or stream.

  See Using a Cascading Style Sheet with C1Editor for an example on using the LoadDesignCSS method.

- **Code clean-up routines**

  On document loading, switching off **Source** mode, or on executing the ValidateSource or FixSource methods, C1Editor automatically removes empty tags, closes unclosed tags, and generally improves messy or unreadable HTML or XHTML code, converting it to valid XHTML.

- **Editor for WinForms' built-in spell checker allows you to check as you type**

  Spell-checking functionality is provided by ComponentOne's **C1SpellChecker** component.

  C1Editor fully supports **C1SpellChecker** so you can use all of its great features, including: modal **Dialog mode** (users can choose to correct or ignore errors through a dialog box), **As-you-type mode** (spelling errors are indicated by a red, wavy underline), and the **AutoReplace** feature (misspelled words are automatically corrected as you type). In **As-you-type mode**, the built-in C1Editor context menu merges with the **C1SpellChecker** context menu so you can see and select all available commands.

- **Ability to add custom tags in DTD**

  Advanced programming tasks sometimes require using additional DTD elements in the edited document. You can enter the elements, or tags, in the document specifying them using the XmlExtensions property in special XML format. See the **CustomTags** sample installed with this product for a complete example of this feature.

- **Access and manage data from code selections and the caret position**

  You can access content in the C1Editor by specifying a range of characters to select. For an example, see Selecting Characters in the C1Editor.

- **Text decoration and block formatting commands**

  You can easily set font and text decoration and block formatting properties in a text block without worrying about how to modify the underlying XmlDocument. Use the C1TextRange class to identify target text and the C1TextRange methods to apply the decoration or formatting: ApplyTag, ApplyClass, ApplyStyle, and **ApplyFormatting**.

- **Use built-in or custom dialog boxes to insert links, pictures, tables, and other objects**

  You can show built-in or custom dialog boxes to insert or edit various objects at the current selection. The dialog boxes allow you to specify all properties of the inserted or edited object. For example, the **Picture** dialog box contains fields to select the image source, or file name, alternate text, size, and so on. C1Editor also has a built-in **Find and Replace** dialog box that allows the user to specify a string to search for and a replacement string, as well as the options to use when searching for text in a document. If you prefer, you can create and use your own find and replace dialog box, specifying it in the CustomDialogs property of the C1Editor.

  For more information on **C1Editor**'s built-in dialog boxes, or for steps on how to use your own, see C1Editor Dialog Boxes.

- **Use the mouse to move or resize pictures or tables**

  You can interact with objects directly in**Design**mode by specifying their size or position with the mouse.

  Moving is a direct manipulation. You can move an object within a document in any direction. C1Editorwill move relative to the XmlNode and/or set the new position attributes of the object automatically.

  Resizing allows you to resize the selected object in the direction of the mouse pointer movement. **C1Editor** will set the new size attributes of the object automatically.

- **Printing support**

  Calling the Print method has the same effect as choosing **Print** from the Windows Internet Explorer **File** menu. The Print method can activate the **Print** dialog box, prompting the user to change print settings.

  The PrintPreview method lets you see the Web page before you print it so you can avoid printing mistakes.

  See the **PrintTemplate** sample installed with this product for a complete example of this feature.

- **Clipboard support**

  You can select text, tables or graphics and use the Cut or Copy methods of C1Editor to move your selection to the Clipboard. Then you can paste the selection into another program. You can copy HTML from a program or even a picture in Internet Explorer and then use the Paste method to put it into the C1Editor.

  The PasteAsText method automatically formats the text you paste into the C1Editor as plain text.

  C1Editor also supports keyboard shortcuts such as CTRL+C (copy), CTRL+X (cut), and CTRL+V (paste).

  To determine what clipboard operations are allowed, use the CanCut, CanCopy, CanPaste, and CanPasteAsText properties.

- **History of changes (undo/redo)**

  C1Editor has an unlimited undo history mechanism. You can programmatically access the editing history, calling the Undo or Redo methods. The **Undo** and **Redo** methods also support keyboard shortcuts: CTRL+Z (undo) and CTRL+Y (redo).

  By undoing repeatedly, a user can gradually work back to the point before the error was made.

Editor for WinForms Elements

Editor for WinForms includes the following controls:

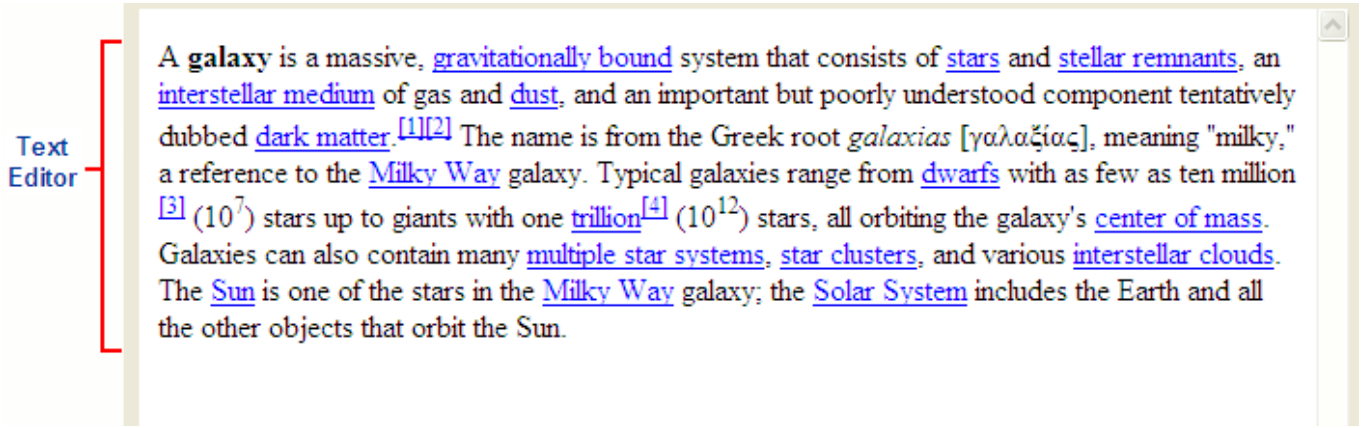
- C1Editor
- C1EditorToolStripMain
- C1EditorToolStripObjects
- C1EditorToolStripStyle
- C1EditorToolStripTable

The **C1EditorToolStrips** can be added to your form and linked to the C1Editor control through the Editor property. See Linking a ToolStrip to C1Editor for more information.

You also have the option of implementing your own ToolStrip using the C1EditorToolStripBase as the base class. You can then add an C1EditorToolStripButton or C1EditorToolStripComboBox to a standard ToolStrip and set the Editor and Command properties. See Creating a Custom ToolStrip for more information.

C1Editor Overview

The C1Editor control is a rich text editor that can be bound to an Xhtml document. Any document bound to the C1Editor control can be loaded, saved or edited.



Compared to the RichTextBox that ships with Visual Studio, the C1Editor has the following advantages:

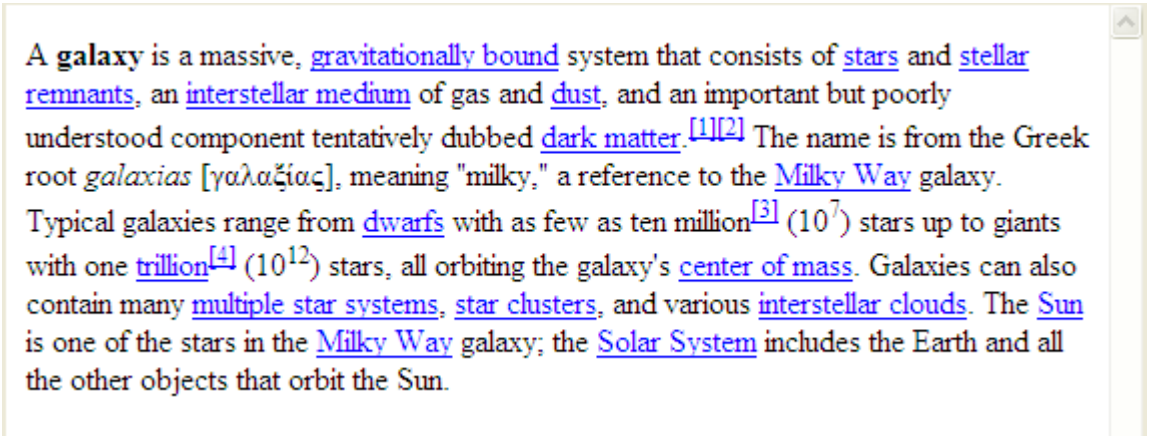
1. C1Editor uses XHTML as its basic format (instead of RTF or proprietary formats). XHTML is a specialized version of XML, which means it is extremely easy to automate all kinds of document handling tasks.
2. C1Editor fully supports cascading style sheets that separate content from presentation and facilitate document handling even further. You can choose to edit documents in preview or source mode, editing the XML tags directly or seeing the document as it will appear to the final users.
3. C1Editor has a powerful and simple object model based on the **XmlDocument** class. If you manipulate XML documents using the XmlDocument class, you already know how to create and edit documents programmatically with the C1Editor.
4. C1Editor supports advanced features such as tables, lists, images, and as-you-type spell-checking. This is the same editor we use in our own award-winning Doc-To-Help documentation tool. (Spell-checking is based on the **C1SpellChecker** component which works great with the C1Editor).
5. C1Editor ships with four specialized ToolStrip controls that allow you to build complete user interfaces without writing any code. Add the toolstrip to the form, connect it to a C1Editor by setting the toolstrip's Editor property, and you have a professional quality XHTML editor.

C1Editor Modes

The C1Editor control features three editor modes: **Design**, **Source**, and **Preview**. You can determine which of these views users will see initially by setting the Mode property.

- **Design View**

  This view displays the text editor's content in a What-You-See-Is-What-You-Get (WYSIWYG) format. Here you can add content without markup. The following image shows the C1Editor in **Design** view:

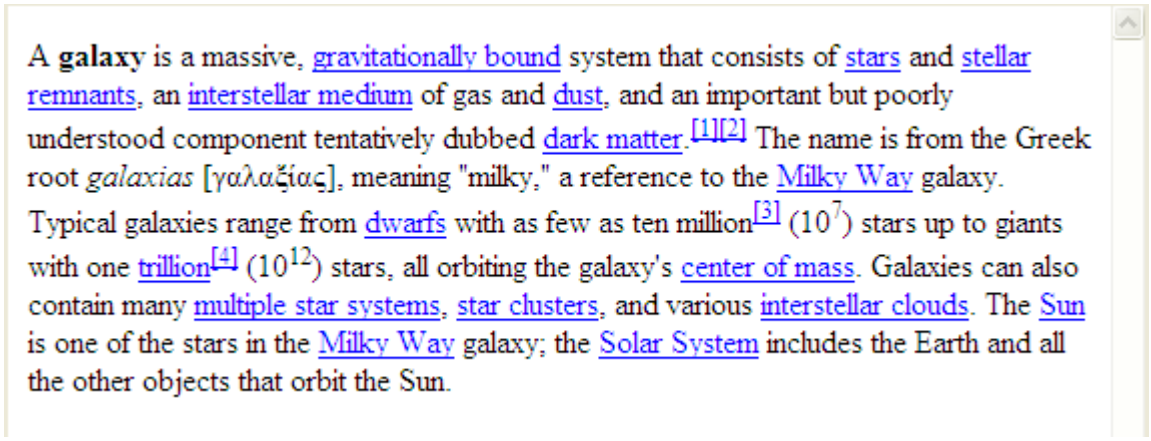


- **Source View**

  This view provides a hand-coding environment for writing and editing HTML markup. The following image shows the text editor in **Source** view:

- **Preview View**

  In this mode, no editing is allowed; **Preview** mode is strictly for viewing the content. The following image shows the text editor in **Preview** mode:



You can set the initial run-time view of the text editor by setting the Mode property. The EditorMode enumeration can be set to one of three settings: **Design**, **Source**, or **Preview**. For more information on how to set the editing mode, see Changing the C1Editor Mode.

Scrollbars will automatically appear if content added to the text editor exceeds the available screen space.

C1EditorToolStripMain Overview

The C1EditorToolStripMain control is a standard text-style ToolStrip made up of C1EditorToolStripButtons allowing you to: create, open, save or print a new file; cut, copy, and paste text; undo or redo an action; select all text or find and replace text; switch between design and source view; or preview the Xml document.

C1EditorToolStripObjects Overview

The C1EditorToolStripObjects control is made up of C1EditorToolStripButtons allowing you to insert objects, including tables, pictures, hyperlinks, bookmarks, and flash movies.

C1EditorToolStripStyle Overview

The C1EditorToolStripStyle control is made up of a C1EditorToolStripComboBox and C1EditorToolStripButtons allowing you to: set paragraph or heading styles; bold, italic, underline, or strikethrough your text; clear formatting; create subscript or superscript text; change the font size to small or big; change the font color; highlight text; add background shading; change paragraph indentation or alignment; or create bulleted or numbered lists.

C1EditorToolStripTable Overview

The C1EditorToolStripTable control is made up of C1EditorToolStripButtons allowing you to: insert a table; set table, row, column, and cell properties; insert a row above or below the selected row; insert a column before or after the

selected column; and delete a table, row, or column.

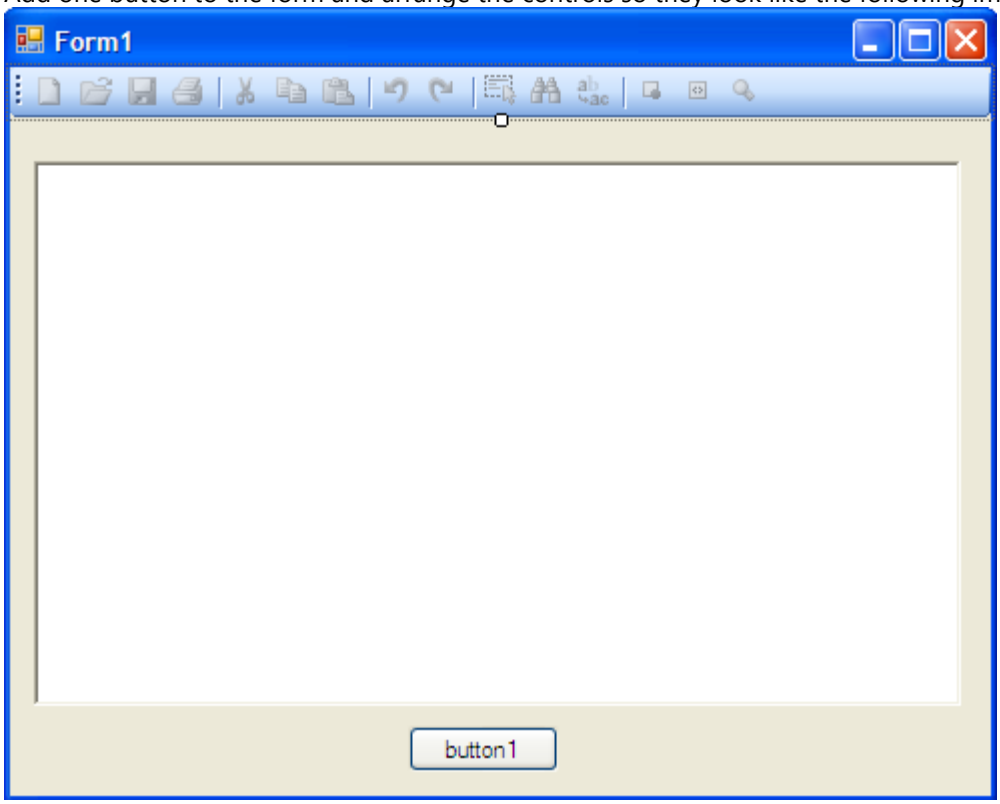
Editor for WinForms Quick Start

This section details some of the features of **Editor for WinForms**. This quick start will walk through the steps of adding a C1Editor and C1EditorToolStripMain control to your project, binding the editor to a document, applying a cascading style sheet, and using some of the buttons on the C1EditorToolStripMain toolbar.

Step 1 of 4: Adding Editor for WinForms Components to the Form

In this topic you will add the C1Editor and C1EditorToolStripMain controls to your form and create a basic application.

1. Create a new Windows application.
2. While in Design view, double-click the C1Editor and C1EditorToolStripMain controls in the Visual Studio Toolbox to add them to your form.
3. Add one button to the form and arrange the controls so they look like the following image:



4. Right-click the C1EditorToolStripMain control on your form and select **Properties**.
5. In the Visual Studio Properties window, click the drop-down arrow next to the Editor property and select **c1Editor1**. This links the toolstrip to the C1Editor control.
6. Select **button1** and enter **Apply CSS** next to the **Text** property in the Properties window.

In the next step you will bind C1Editor to a document.

Step 2 of 4: Binding C1Editor to a Document

Now you can bind C1Editor to a document that can be saved to a file and loaded later when needed. If this document is edited within the C1Editor, the underlying XmlDocument syncs to match it.

1. Click the **View** menu and select **Code** to switch to code view.
2. Add the following Imports (Visual Basic) or using (C#) statements to your project so you can use abbreviated names.

   To write code in Visual Basic

   ```
   Visual Basic
   Imports System.Xml
   Imports C1.Win.C1Editor
   ```

   To write code in C#

   ```
   C#
   using System.Xml;
   using C1.Win.C1Editor;
   ```

3. Create a **Form_Load** event and add the following code there to create a new document and bind it to C1Editor:

   To write code in Visual Basic

   ```
   Visual Basic
   Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
       Dim myDoc As New XmlDocument()
       C1Editor1.Document = myDoc
   End Sub
   ```

   To write code in C#

   ```
   C#
   private void Form1_Load(object sender, EventArgs e)
   {
     XmlDocument myDoc = new XmlDocument();
     c1Editor1.Document = myDoc;
   }
   ```

In the next step you will add code to apply a cascading style sheet to the C1Editor content.

Step 3 of 4: Applying a Cascading Style Sheet

In this topic you will add the code that allows you to use a cascading style sheet (CSS) when the button is clicked.

1. In the Visual Studio **File** menu, select **New | File**. The **New File** dialog box appears.
2. Select **General** under *Categories* and choose **Style Sheet** under *Templates*.
3. Click **Open** and add this markup to the style sheet so it looks like the following:

   Markup

   ```
   body
   {
   font-family: Verdana;
          font-size: 10pt;
          line-height: normal;
          margin-bottom: 0pt;
   ```

```
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 0pt;
        color: Fuchsia;
}
h1 {
        font-family: Verdana;
        font-size: 20pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 8pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 10pt;
}
h2 {
        font-family: Verdana;
        font-size: 16pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 7pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 9pt;
        page-break-after: avoid;
}
h3 {
        font-family: Verdana;
        font-size: 16pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 7pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 9pt;
        page-break-after: avoid;
}
h4 {
        font-family: Verdana;
        font-size: 12pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 2pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 2pt;
        page-break-after: avoid;
}
.C1HBullet {
        font-family: Verdana;
        font-size: 10pt;
         font-style: italic;
        line-height: 14pt;
        margin-bottom: 0pt;
        margin-left: 18pt;
        margin-right: 0pt;
        margin-top: 5pt;
        text-indent: -18pt;
}
p {
        font-family: Verdana;
        font-size: 10pt;
```

```
        line-height: 14pt;
        margin-bottom: 0pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 5pt;
         text-indent: 18pt;
}
.C1SectionCollapsed {
        font-weight: bold;
}
```

4. Click the **Save** button and save the style sheet as *myCSS.css*, for example.
5. Right-click the project name in the Solution Explorer and select **Add | Existing Item**.
6. Choose the CSS and click **Add**.
7. Create a **Button1_Click** event and add the following code there so the cascading style sheet is applied when the button is clicked. The location may be different, depending on where you save your Visual Studio project.

To write code in Visual Basic

Visual Basic

```
Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    C1Editor1.LoadDesignCSS("C:\myCSS.css")

End Sub
```

To write code in C#

C#

```
private void button1_Click(object sender, EventArgs e)
        {
           c1Editor1.LoadDesignCSS(@"C:\myCSS.css");
        }
```

In the next step you will run the project and add text to the editor.

Step 4 of 4: Running the Project

Press **F5** to run the project and follow these steps:

1. Add some text to the editor.
2. Click the **Apply CSS** button. Notice the cascading style sheet is applied to the text.
3. In the C1EditorToolStripMain control, click the **Source view** button to view the code and then click the **Design view** or **Preview** button to switch editor modes.
4. Click the **Find** button to search for an item in the text. This opens the **Find and Replace** dialog box.
5. Click the **Save** button and enter a name for your document. You can open it again later by clicking the **Open file** button.

Congratulations! You have successfully completed the **Editor for WinForms Quick Start**.

C1Editor Design-Time Support

The following sections describe how to use **C1Editor**'s design-time environment to configure the C1Editor control.

Smart Tags

The **Editor for WinForms** controls include a smart tag (▶) in Visual Studio. A smart tag represents a shortcut tasks menu that provides the most commonly used properties in the controls.

To access the **Tasks** menu, click the smart tag in the upper-right corner of the control.

The **Tasks** menus is similar for all **Editor for WinForms** controls and it operates as follows:

- **About C1Editor**

  Displays the **About Editor** dialog box, which is helpful in finding the version number of the product and online resources such as how to purchase a license, how to contact ComponentOne, or how to view ComponentOne product forums.

Context Menus

The **Editor for WinForms** controls have additional commands available on their context menus that Visual Studio provides for all .NET and ASP.NET controls.

Right-click anywhere on one of the controls to display the context menu. The context menu commands operate as follows:

- **About C1Editor**

  Displays the **About Editor** dialog box, which is helpful in finding the version number of the product and online resources such as how to purchase a license, how to contact ComponentOne, or how to view ComponentOne product forums.

C1Editor Run-Time Elements

The following topics provide information regarding the run-time environment of the C1Editor control.

C1Editor Dialog Boxes

Within the C1Editor control are several dialog boxes that users can employ to edit Xhtml documents.

You can easily show a dialog box when a button is clicked, for example, simply by using the ShowDialog method and specifying the DialogType.

For example, to show the **PageSetup** dialog box when a button is clicked, you would use the following code:

To write code in C#

C#

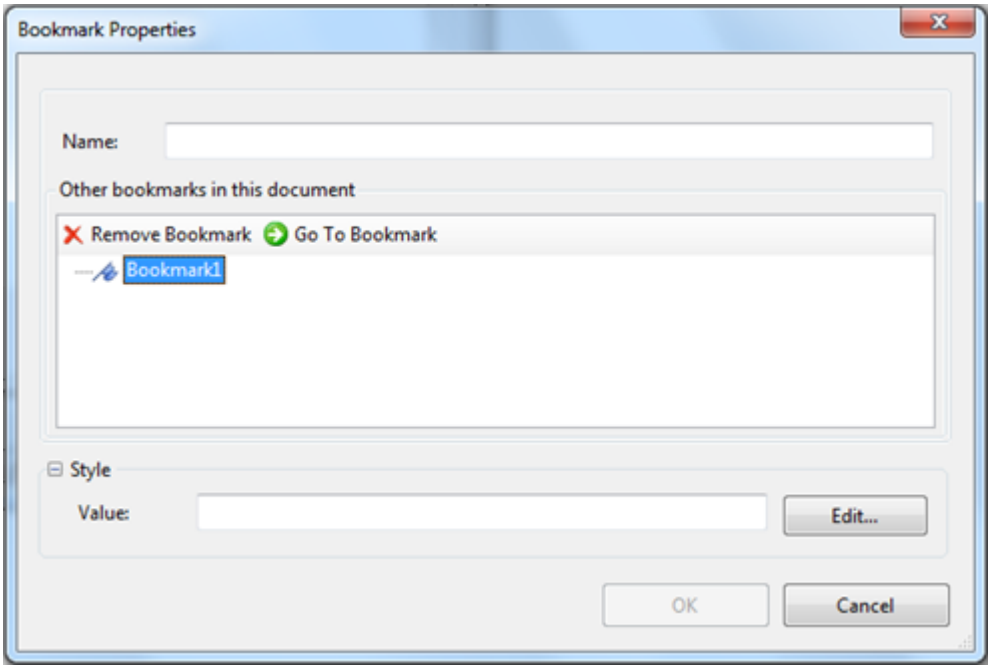
```
private void button7_Click(object sender, EventArgs e)
        {
            c1Editor1.ShowDialog(C1.Win.C1Editor.DialogType.PageSetup);

        }
```

The following topics detail the dialog boxes that can be accessed through the C1Editor control. In some cases you may need to show your own customized versions of the dialog boxes. See Using a Custom Dialog Box for steps on how to do this.

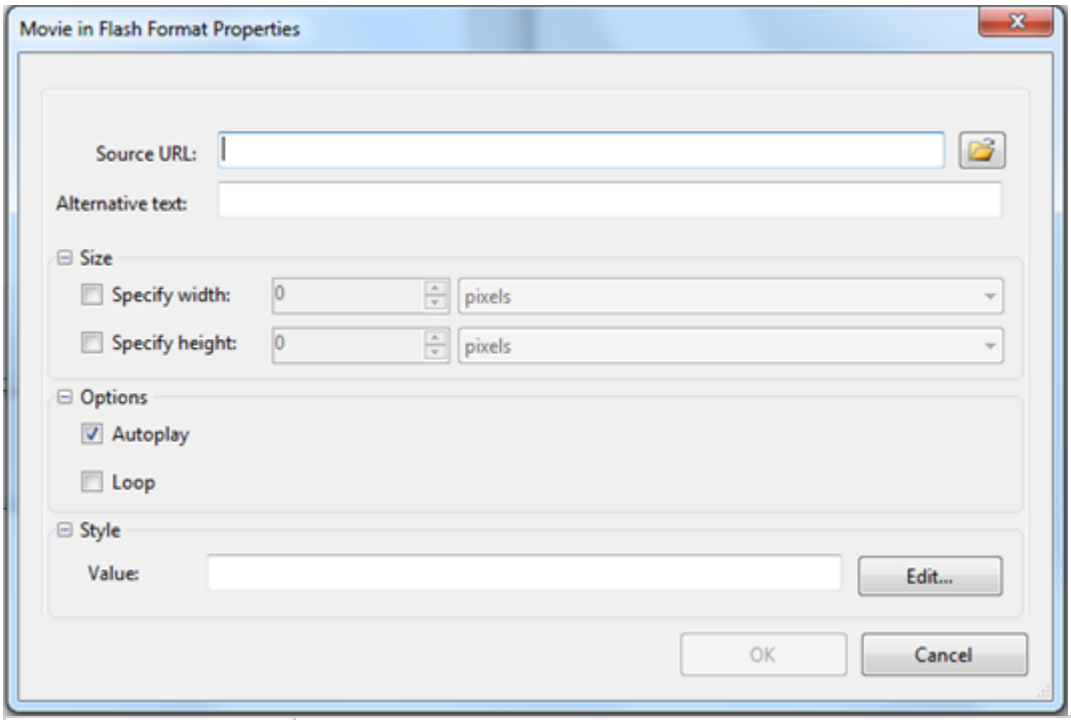
Bookmark Properties Dialog Box

In the **Bookmark Properties** dialog box, users can create a new bookmark by entering a name in the **Name** text box. They can also jump to or remove an existing bookmark.



Movie in Flash Format Properties Dialog Box

In the **Movie in Flash Format Properties** dialog box, users can specify the Flash movie to be inserted in the document as well as set the window size and play options.



Source URL	Click the **Browse** button to locate a Flash movie.
Alternative text	Enter the text that should appear in case the movie is unavailable.

Size

Specify width	Select this checkbox and enter a number to specify the movie window width. Choose pixels or percent.
Specify height	Select this checkbox and enter a number to specify the movie window height. Choose pixels or percent.

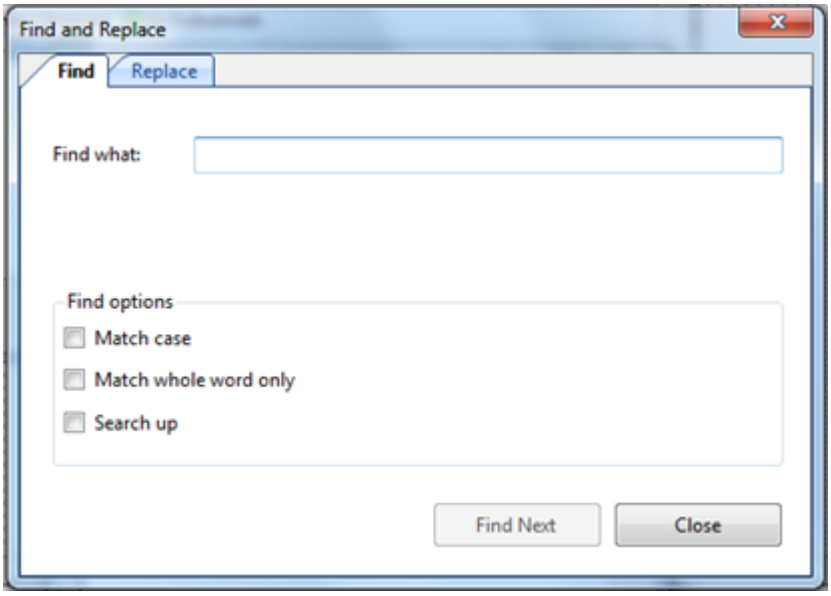
Options

Autoplay	Select this check box to automatically start the movie when the document is loaded.
Loop	Select this check box to continuously play the movie.

Find and Replace Dialog Box

The **Find** and **Replace** dialog boxes are the same dialog box with two tabs allowing you to find and/or replace text.

In the **Find and Replace** dialog box, users can find text and replace it with other text if they choose. To access the **Find and Replace** dialog box, click inside the C1Editor control and click CTRL+F.



To find specific text, enter the text in the **Find what** text box, select one of the check boxes under **Find options**, and click **Find Next**.

To replace specific text, enter the text to be replaced in the **Find what** text box, enter the text to replace it in the **Replace with** text box, and click **Replace** to replace certain instances or click **Replace All** to replace all instances.

Find Options

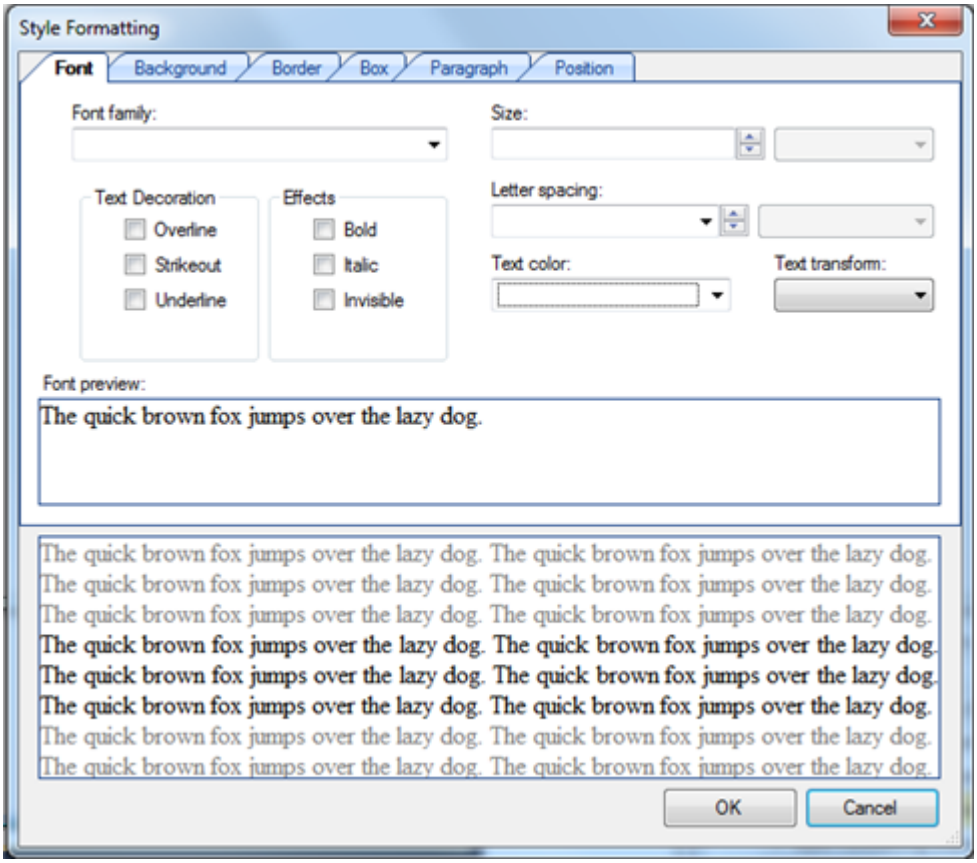
The **Find options** section of the **Find and Replace** dialog box allows you to specify certain search criteria as described in the following table:

Option	Description
Match case	Locates text that exactly matches the combination of uppercase and lowercase letters you type in the **Find what** box.
Match whole word only	Locates distinct occurrences of words, not groups of characters inside words.
Search up	Searches for text specified in the **Find what** box from the bottom of the control upwards.

Style Formatting Dialog Box

The **Style Formatting** dialog box can be accessed by right-clicking text in the editor and selecting **Formatting** or by clicking the **Edit** button in the Style section of the following dialog boxes: **Bookmark**, **Movie in Flash Format Properties**, **Hyperlink**, **Picture**, **Table**, **Row Properties**, **Column Properties**, and **Cell Properties**.

The **Style Formatting** dialog box consists of six tabs: **Font**, **Background**, **Border**, **Box, Paragraph, and Position**. Note that not all tabs are available for each of the dialog boxes.



Font Tab

On the **Font** tab, users can specify the following:

Font family	Determines the font used.
Size	Determines the size of the text.
Text Decoration	Gives you the option to overline, strikeout, or underline the text.
Effects	Gives you the option to bold, italicize, or make the text invisible.
Letter spacing	Determines the amount of space, in pixels, to place between letters.
Text color	Determines the color of the text.
Text transform	Allows you to change the text to capitalize the first letter of each word, make all letters lowercase, or make all letters uppercase.

Background Tab

Background color	Determines the color to place behind the text.

Image

File name	Click the **Browse** button to locate an image to place in the editor.

Position Y	Determines the horizontal position of the background image relative to the text.
Repeat	Determines whether the image is repeated vertically, horizontally, both or not at all.
Position X	Determines the vertical position of the image relative to the text: top, center, or bottom.

Border Tab

Border style	Determines the style for the top, bottom, right and left border. Options include: None, Dashed, Dotted, Double, Groove, Hidden, Inset, Outset, Ridge, and Solid.
Border width	Sets a Thin, Medium, or Thick border for the top, bottom, right, and left borders.
Border color	Determines the color of the border for top, bottom, right, and left borders.

Box Tab

Padding	Determines the padding space around the text, in pixels, from the top, right, left, and bottom.
Margins	Determines the margin space, in pixels, from the top, right, left, and bottom.

Paragraph Tab

Text align	Aligns the selected text Left, Right, Center, or Justify.
Word spacing	Determines the amount of space, in pixels, between each word.
Line height	Determines the height of the selected line(s), in pixels.
Text indent	Determines the number of pixels to indent the first line.

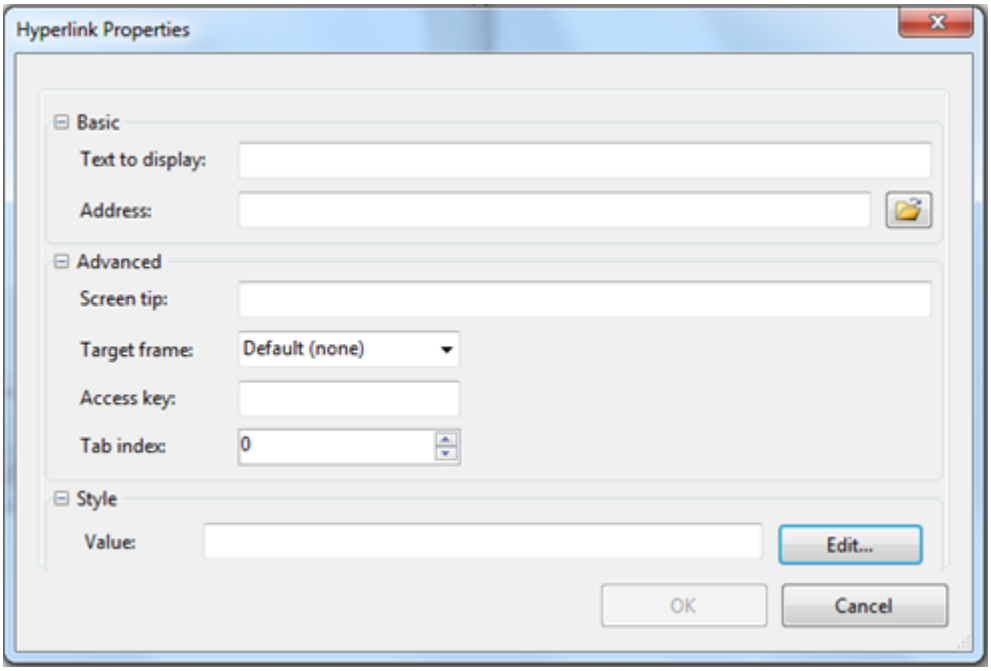
Position Tab

Position	Determines the position of the table.
Z index	Sets the stack order of elements.
Width	Determines the width of cells.
Height	Determines the height of cells.

Hyperlink Properties Dialog Box

In the **Hyperlink Properties** dialog box, users can enter a URL, specify the text to display, add a tooltip, specify the

frame where the Web page will open, create an access key, and enter a tab index.



Basic

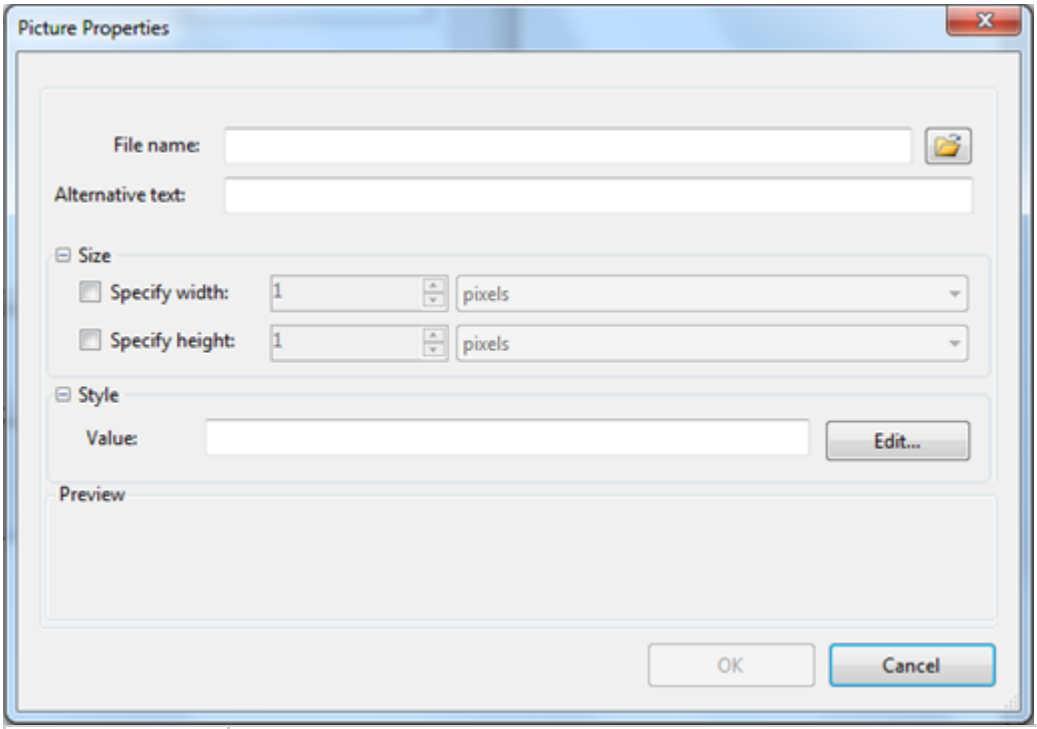
Text to display	Enter the text that will appear as a hyperlink in the document.
Address	Enter the URL address for the hyperlink.

Advanced

Screen tip	Enter the tooltip text that will appear when a user hovers over the hyperlink.
Target frame	Click the drop-down arrow to select a frame where the web page will appear when the hyperlink is clicked.
Access key	Enter a key that can be pressed with the ALT key to jump to a specific control on the page without using the mouse.
Tab index	Enter a number to define the tab order for the hyperlink.

Picture Properties Dialog Box

Users can insert and preview a picture using the **Picture Properties** dialog box.



File name	Click the **Browse** button to locate a picture to insert.
Alternative text	Enter the text that should appear in case the image is unavailable.

Size

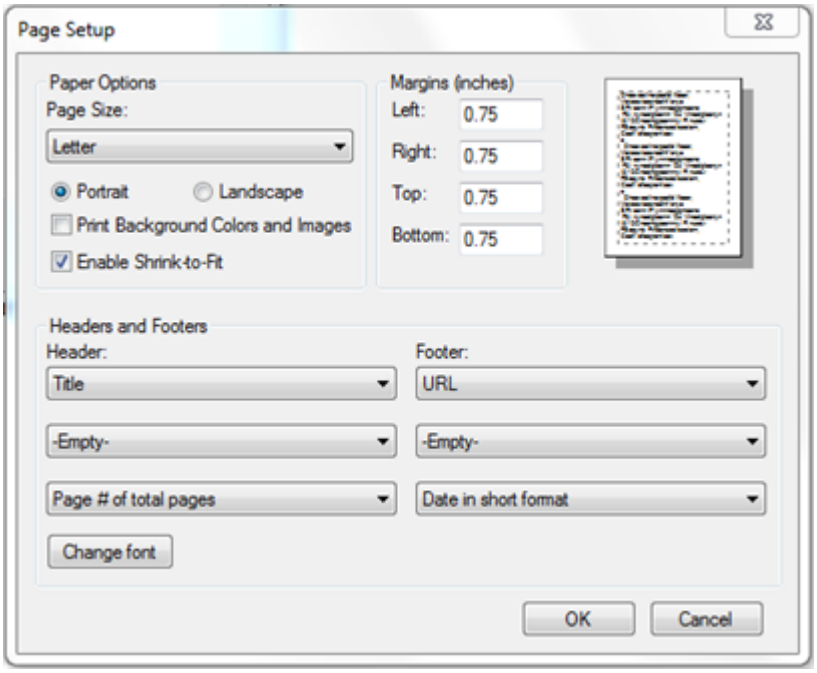
Specify width	Select this checkbox and enter a number to specify the image width. Choose pixels or percent.
Specify height	Select this checkbox and enter a number to specify the image height. Choose pixels or percent.

Preview

A preview of the image appears in this area.

Page Setup Dialog Box

In the **Page Setup** dialog box, users can specify the page size and orientation, set page margins, and create headers and footers.



Paper Options

Page Size	Click the drop-down arrow and select a paper size.
Portrait or **Landscape**	Select **Portrait** or **Landscape** to determine the page orientation.
Print Background Colors and Images	Select this checkbox to print background colors and pictures.
Enable Shrink-to-Fit	Select this checkbox to scale the content to fit on the page.

Margins

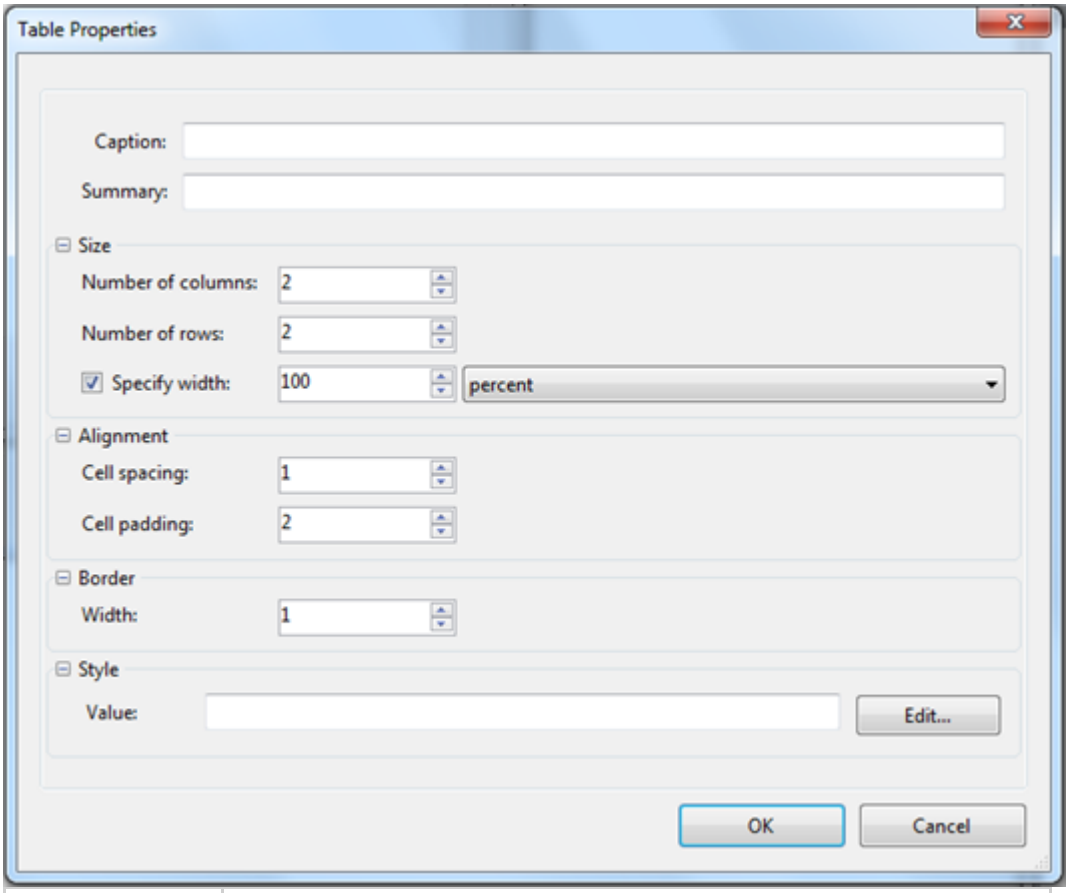
Specify the page margins in the **Left**, **Right**, **Top**, and **Bottom** boxes.

Headers and Footers

Click the drop-down arrows under **Header** or **Footer** to select the header or footer to be inserted, or create your own custom header or footer. Click the **Change Font** button to change the header or footer font, style, and size.

Table Properties Dialog Box

In the **Table Properties** dialog box, users create a new table, specify the number of columns and rows, set up cell spacing and padding, create a border, and add a caption for the table. Right-click the table to access additional properties for the row, column, or cell.



Caption	Enter the caption to appear above the table.
Summary	Enter a description for the table.

Size

Number of columns	Enter the number of table columns.
Number of rows	Enter the number of table rows.
Specify width	Select this checkbox to specify the width of the table, either in percent or pixels.

Alignment

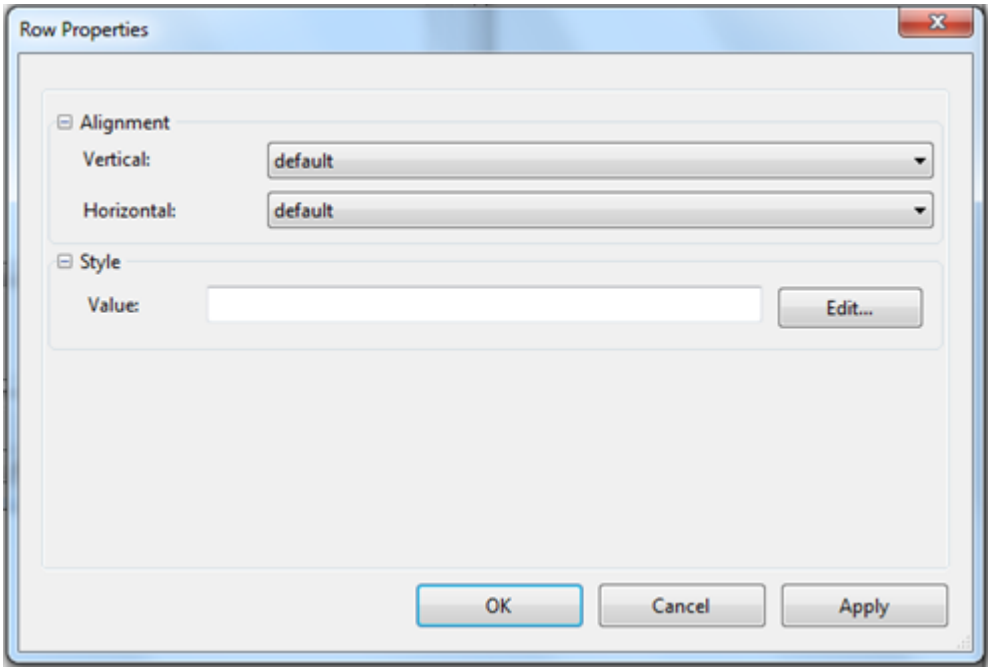
Cell spacing	Enter a number to determine spacing between cells.
Cell padding	Enter a number to determine the spacing that appears around cell content.

Border

Width	Enter the width of the table border.

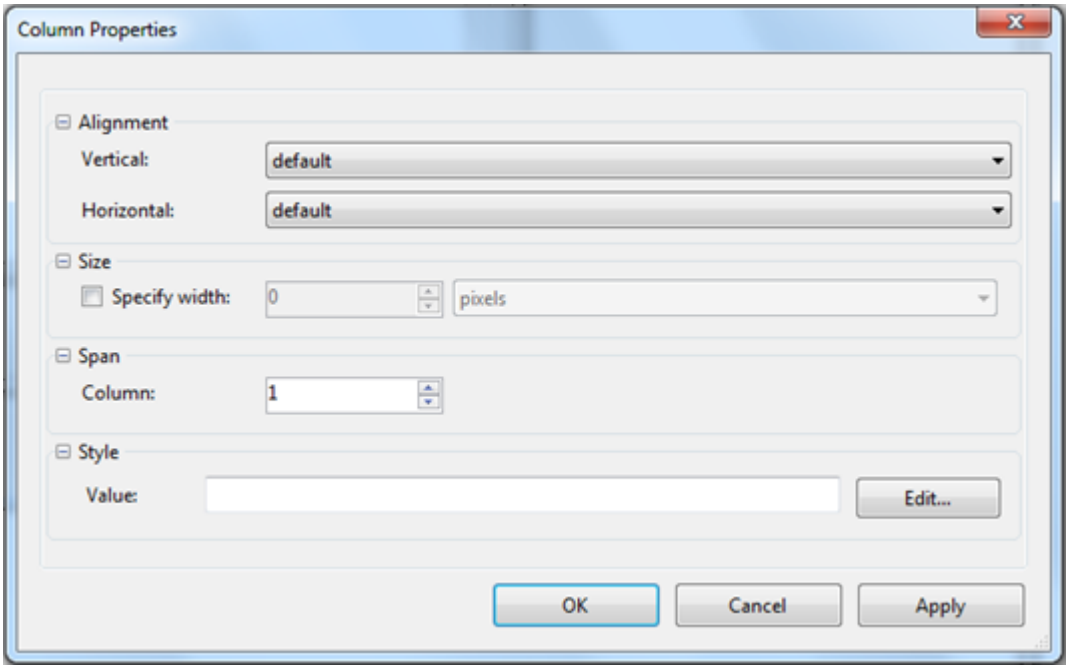
Row Properties

Right-click a table and select **Row Properties** to access this dialog box. Click the drop-down arrows next to **Vertical** or **Horizontal** to specify the alignment.



Column Properties

Right-click a table and select **Column Properties** to access this dialog box.



Alignment

Vertical	Click the drop-down arrows to specify the vertical alignment: top, middle, or bottom.
Horizontal	Click the drop-down arrows next to **Horizontal** to specify the alignment: left, center, or right.

Size

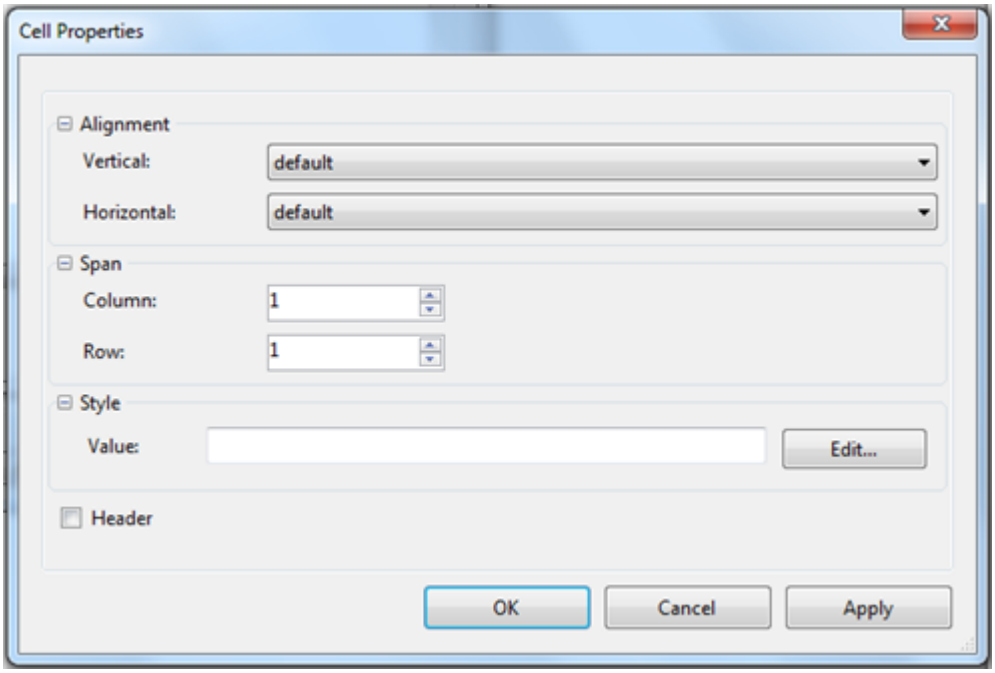
Specify width	Select this checkbox and enter a number in pixels or percent for the width of the column.

Span

Column	Enter a number to determine how many columns the property changes should affect.

Cell Properties

Right-click a table and select **Cell Properties** to access this dialog box.



Alignment

Vertical	Click the drop-down arrows to specify the vertical alignment: top, middle, or bottom.
Horizontal	Click the drop-down arrows next to **Horizontal** to specify the alignment: left, center, or right.

Span

Column	Enter a number to determine how many columns the property changes should affect.
Row	Enter a number to determine how many rows the property changes should affect.

Header

Select this checkbox format the text in a cell as a header, or centered and bold.

Using a Custom Dialog Box

There may be instances where you want to use your own custom version of the built-in C1Editor dialog boxes. This can easily be done with the CustomDialogs property.

First implement the custom dialog box and make sure it supports the appropriate interface, **IFindReplaceDialog** for a custom **Find and Replace** dialog box, for example.

In this example, we'll assume you have created three custom dialog boxes: **BookmarkDialog**, **FindReplaceDialog**, and **FormattingDialog**.

Add the following code to your project to set the CustomDialogs property.

To write code in C#

C#

```
private void InitCustomDialogs()
        {
            editor.CustomDialogs.BookmarkDialog = new BookmarkEditorForm();
            editor.CustomDialogs.FindReplaceDialog = new FindReplaceForm();
            editor.CustomDialogs.FormattingDialog = new FormattingForm();
        }
```

Then you can use the ShowDialog method to open each new dialog box. In this example, we have a toolStrip with three buttons that, when clicked, open the custom dialog boxes.

To write code in C#

C#

```
private void toolStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
        {
            // opens the Bookmark dialog box
            if (e.ClickedItem == buttonBookmark)
               editor.ShowDialog(C1.Win.Editor.DialogType.Bookmark);
            // opens the Find dialog box
            else if (e.ClickedItem == buttonFind)
               editor.ShowDialog(C1.Win.Editor.DialogType.Find);
            // opens the Formatting dialog box
            else if (e.ClickedItem == buttonFormatting)
               editor.ShowDialog(C1.Win.Editor.DialogType.Format);
```

```
        }
```

For a detailed example on creating and using custom dialog boxes, see the **Custom Dialogs** sample installed with the product.

Keyboard Shortcuts

The **Editor for WinForms** controls allow users to complete several functions through the use of keyboard shortcuts when the KeyboardShortcutsEnabled property is set to **True** (default). The following table details the functions that can be accessed through keyboard shortcuts:

Function	Keyboard Shortcut
Bold	Ctrl+B
Italicize	Ctrl+I
Underline	Ctrl+U
Undo	Ctrl+Z
Redo	Ctrl+Y
Select All	Ctrl+A
Copy	Ctrl+C, Ctrl+Shift+Insert
Cut	Ctrl+X, Ctrl+Delete
Paste	CTRL+V, Ctrl+Shift+Insert
Clear Formatting	Ctrl+Space
Find	Ctrl+F
Replace	Ctrl+H
Print	Ctrl+P
Save	Ctrl+S
New	Ctrl+N
Open	Ctrl+O

Creating an XHTML Editor in Code

The following topics introduce the C1Editor control by walking you through the creation of a simple XHTML editor with custom toolbars and built-in style sheets. The toolbars show how you can implement several common editing tasks using code. They are meant to be used for documentation purposes, not as a replacement to the toolbars included with the C1Editor which are much more extensive and can be used without any programming.

The following topics are demonstrated in the **C1EditorQuickStart** sample included with the product. Please refer to the pre-installed product samples through the following path:

Documents\ComponentOne Samples\WinForms

The ToolStripBase class

The **ToolStripBase** class is an abstract class that provides common functionality to all C1Editor toolstrips used in our project.

Specifically, the base class provides an Editor property that connects the toolstrip to an editor and connects handlers to the DocumentChanged and SelectionChanged events. It also provides an AddButton method that makes it easy to populate the toolstrips.

The **ToolStripBase** class is extremely simple. For details, please refer to the source code.

The ToolStripMain class

The **ToolStripMain** class derives from **ToolStripBase** and provides the usual file, clipboard, editing, and spell-checking commands. It implements a subset of the C1EditorToolstripMain class that ships with the C1Editor.

This is what the **ToolStripMain** control looks like:

File Commands

The **ToolStripMain** class implements the usual *New*, *Open*, *Save* and *Save As* file commands.

The **New** command is implemented as follows:

To write code in C#

C#

```
bool NewDocument()
{
    if (OKToDiscardChanges())
    {
        Editor.Document = new System.Xml.XmlDocument();
        _fileName = null;
        SetDirty(false);
        return true;
    }
    return false;
}
```

The method calls a helper **OkToDiscardChanges** method that checks whether the loaded document has any changes. If it does, the method prompts the user to verify that it is OK to discard the changes before creating a new document.

To actually create the new document, the method sets the **C1Editor**'s Document property to a new instance of an **XmlDocument** object. The **XmlDocument** represents the content of the editor, and you may use the **XmlDocument** object model to modify the document.

This is one of the main strengths of the C1Editor control. It allows you to create and modify document content using the powerful and familiar **XmlDocument** object model. This will be demonstrated in more detail in later sections.

The **Open** command is implemented as follows:

To write code in C#

C#

```
bool LoadDocument()
{
    if (OKToDiscardChanges())
    {
        using (OpenFileDialog dlg = new OpenFileDialog())
        {
            // get file name
            dlg.Filter = Properties.Resources.FileFilter;
            dlg.DefaultExt = Properties.Resources.DefaultExt;
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                try
                {
                   // load document
                    Editor.LoadXml(dlg.FileName);
                    _fileName = dlg.FileName;
                    SetDirty(false);
                    return true;
                }
                catch (Exception x)
                {
                    MessageBox.Show(x.Message, "Error",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
                    return false;
                }
            }
        }
    }

    // canceled...
    return false;
}
```

As before, the method starts by calling the **OKToDiscardChanges** method. It then uses an **OpenFileDialog** to allow the user to pick the document he wants to load. Finally, the method calls the **C1Editor**'s LoadXml method to load the document into the editor.

The **Save** method is implemented in a similar way:

To write code in C#

C#

```
bool SaveDocument()
{
    // no name? go get one...
    if (string.IsNullOrEmpty(_fileName))
    {
        return SaveDocumentAs();
    }

    // got the name, save the file
    try
    {
        Editor.SaveXml(_fileName);
        SetDirty(false);
        return true;
    }
    catch (Exception x)
    {
        MessageBox.Show(x.Message, "Error",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
        return false;
    }
}
```

The method starts by checking whether the current document already has a file name associated with it. If it does not, then it calls the **SaveDocumentAs** method, which prompts the user for a file name and then returns to the **SaveDocument** method.

Once the document has a file name, the method uses the **C1Editor**'s SaveXml method to save the current document to the file.

Clipboard Commands

The **ToolStripMain** class implements the usual *Cut*, *Copy*, and *Paste* clipboard commands.

All clipboard commands are deferred to the C1Editor itself:

To write code in C#

C#

```
void Cut_Click(object sender, EventArgs e)
{
    Editor.Cut();
}
void Copy_Click(object sender, EventArgs e)
{
    Editor.Copy();
}
void Paste_Click(object sender, EventArgs e)
```

```
{
    Editor.PasteAsText(); //.Paste();
}
```

The C1Editor clipboard commands are simple and easy to use, like the ones in the regular **TextBox** and **RichTextBox** controls.

Notice that the **Paste** command is implemented with a call to the PasteAsText command. This ensures that the text being pasted is not interpreted as XHTML.

Editing Commands

The **ToolStripMain** class implements the usual *Undo*, *Redo*, *Find*, and *Replace* commands, which are also deferred to the C1Editor control:

To write code in C#

C#

```
void Undo_Click(object sender, EventArgs e)
{
    Editor.Undo();
}
void Redo_Click(object sender, EventArgs e)
{
    Editor.Redo();
}
void Find_Click(object sender, EventArgs e)
{
    Editor.ShowDialog(C1.Win.C1Editor.DialogType.Find);
}
void Replace_Click(object sender, EventArgs e)
{
    Editor.ShowDialog(C1.Win.C1Editor.DialogType.Replace);
}
```

The find and replace commands are implemented with calls to the **C1Editor**'s ShowDialog commands, which also supports several other commands defined by the DialogType enumeration and not used in this sample:

- **NewTable:** Inserts a new table.
- **Image:** Inserts or edits an image.
- **FlashMovie:** Inserts or edits a Flash movie.
- **Hyperlink:** Inserts or edits a hyperlink.
- **Bookmark:** Inserts or edits a bookmark.
- **Find:** Finds text in the document.
- **Replace:** Finds and replaces text in the document.
- **Format:** Applies formatting to the current selection.
- **PageSetup:** Shows a page setup dialog.

Word Count command

Finally, the **ToolStripMain** class implements a word count command similar to the one in some versions of Microsoft Word. This method requires extracting the text in the document and analyzing it to count characters, words,

sentences, and paragraphs.

Here is the implementation:

To write code in C#

C#

```
void WordCount_Click(object sender, EventArgs e)
{
    int chars = 0;
    int nonSpaceChars = 0;
    int words = 0;
    int paragraphs = 0;
    int sentences = 0;

    // calculate statistics
    string text = Editor.Text;
    int length = text.Length;
    for (int i = 0; i < length; i++)
    {
        char c = text[i];

        // count chars
        chars++;

        // count non-space chars
        if (!char.IsWhiteSpace(c))
        {
            nonSpaceChars++;
        }

        // count paragraphs
        if (c == '\n' || i == length - 1)
        {
            if (i == length - 1 || text[i + 1] != '\n')
            {
                paragraphs++;
            }
        }

        // count sentences
        if (c == '.' || c == '!' || c == '?' || i == length - 1)
        {
            if (i == length - 1 || char.IsWhiteSpace(text, i + 1))
            {
                sentences++;
            }
        }

        // count words
        if (char.IsLetterOrDigit(c))
        {
```

```
            if (i == length - 1 || !char.IsLetterOrDigit(text, i + 1))
            {
                words++;
            }
        }
    }

    // show statistics
    string msg = string.Format(
        "Words: {0:n0}\r\n" +
        "Characters: {1:n0}\r\n" +
        "Non-Space Characters: {2:n0}\r\n" +
        "Sentences: {3:n0}\r\n" +
        "Paragraphs: {4:n0}\r\n" +
        "Average Word Length: {5:n1}\r\n" +
        "Average Sentence Length: {6:n1}\r\n" +
        "Average Paragraph Length: {7:n1}\r\n",
        words, chars, nonSpaceChars, sentences, paragraphs,
        words > 0 ? nonSpaceChars / (float)words : 0f,
        sentences > 0 ? nonSpaceChars / (float)sentences : 0f,
        paragraphs > 0 ? nonSpaceChars / (float)paragraphs : 0f);
    MessageBox.Show(msg, "Word Count");
}
```

The method starts by declaring variables to hold the statistics it will calculate. It then retrieves the document contents as plain text using the Text property, and scans that string counting characters, words, sentences, and paragraphs. Finally, it displays a message box with the statistics.

This sample shows that although the C1Editor supports the rich and powerful **XmlDocument** object model, you can easily bypass that and get directly to the actual text content when that is convenient. The Text property is similar to the **Text** property in the **RichTextBox** control.

Spell-Checking Commands

The C1Editor implements spell-checking with a **SpellChecker** property that allows you to connect it to a **C1SpellChecker** component.

The **C1SpellChecker** is a separate, stand-alone component that can be used to spell-check many types of controls including **TextBox**, **RichTextBox**, **WebBrowser**, and of course the C1Editor. The **C1SpellChecker** includes spelling dictionaries for over 20 languages, and it includes a dictionary editor that allows you to create your own custom dictionaries.

The C1Editor supports as-you-type spell checking (with red wavy lines under misspelled words and suggestions in context menus) as well as a dialog-based spell checking that highlights each misspelled word and allows the user to correct each one from the spell dialog.

The **ToolStripMain** class exposes as-you-type spell-checking as follows:

To write code in C#

C#

```
void ShowSpellingErrors_Click(object sender, EventArgs e)
{
    if (SpellChecker != null)
```

```
    {
        bool show = !_btnShowErrors.Checked;
        _btnShowErrors.Checked = show;
        SpellChecker.SetActiveSpellChecking(
            Editor,
            Editor.GetActiveXInstance(),
            show);
    }
}
```

The core of the method is a call to the **C1SpellChecker**'s **SetActiveSpellChecking** method, which turns the as-you-type spell checking on or off for the specified control.

The **ToolStripMain** class exposes modal spell-checking as follows:

To write code in C#

C#

```
void Spell_Click(object sender, EventArgs e)
{
    if (SpellChecker != null)
    {
        SpellChecker.CheckControl(
            Editor,
            Editor.GetActiveXInstance());
    }
}
```

The core of the method is a call to the **C1SpellChecker**'s **CheckControl** method, which performs the dialog-based spell-checking for the specified control.

Enabling and Disabling Commands

Many of the commands described above may or may not be available depending on whether a document is currently loaded, the selection state, or the clipboard content. If a command is not available, it should be disabled in the tool strip.

The **ToolStripMain** class handles this by overriding the **UpdateState** method as follows:

To write code in C#

C#

```
public override void UpdateState()
{
    Enabled = Editor != null;
    _btnCopy.Enabled = _btnCut.Enabled = Editor.CanCopy;
    _btnPaste.Enabled = Editor.CanPasteAsText; // CanPaste
    _btnUndo.Enabled = Editor.CanUndo();
    _btnRedo.Enabled = Editor.CanRedo();
    _btnSpell.Enabled = _btnShowErrors.Enabled = SpellChecker != null;
}
```

The implementation is self-explanatory. The C1Editor provides properties and methods that allow the toolstrip to

update itself very easily.

The ToolStripStyles class

The **ToolStripStyles** class derives from **ToolStripBase** and provides commands for selecting cascading style sheets, showing and applying styles to the selection, the usual bold, italic and underline buttons, and a reset style button that removes any custom styles applied to the selection.

This is what the **ToolStripStyles** control looks like:

Cascading Style Sheets

The first command in the **ToolStripStyles** class allows users to apply different cascading style sheets to the current document. This illustrates the simple and easy way in which content and formatting can be separated in XHTML and how the C1Editor supports this feature.

The code that implements this feature starts by defining a dictionary with a few cascading style sheet names and definitions:

To write code in C#

C#

```
ToolStripComboBox AddCssCombo()
{
    // define css dictionary
    _css = new Dictionary<string, string>();
    _css["CodeProject"] = @"
        body { font-family:Verdana; font-size:10pt; }
        pre { background-color:#ffd29b; }";
    _css["ComponentOne"] = @"
        body { font-family:Georgia; font-size:11pt; }
        code { background-color:#f8f8f8; }
        pre { background-color:#f8f8f8; }";
    _css["Microsoft"] = @"
        h1 { font-family:Arial; font-size:16pt; border-top:solid 2px black; }
        h2 { font-family:Arial; font-size:14pt; border-top:solid 1px black; }
        h3 { font-family:Arial; font-size:12pt; border-top:solid 1px grey; }
        body { font-family:Times New Roman; font-size:9pt; }
        code { background-color:#f8f8f8; font-size:9pt; }
        pre { background-color:#f8f8f8; font-size:9pt; }";
    _css["Blues"] = @"
        body { font-family:Book Antiqua; font-size:12pt; background-color:#e8e8f8; }
        code { background-color:#000000; color:#ffffff; }
        pre { background-color:#000000; color:#ffffff; }";

    // add combo to expose our style sheets
    ToolStripComboBox cmb = new ToolStripComboBox();
    cmb.AutoSize = false;
    cmb.Width = 120;
```

```
    cmb.DropDownStyle = ComboBoxStyle.DropDownList;
    foreach (string name in _css.Keys)
    {
        cmb.Items.Add(name);
    }

    Items.Add(cmb);
    return cmb;
}
```

This code executes only once, when the tool strip is initialized. It creates a dictionary with four style sheets and adds their names to the combo box on the toolstrip.

When one of the items in the combo box is selected, the following event handler applies the selected style sheet to the C1Editor:

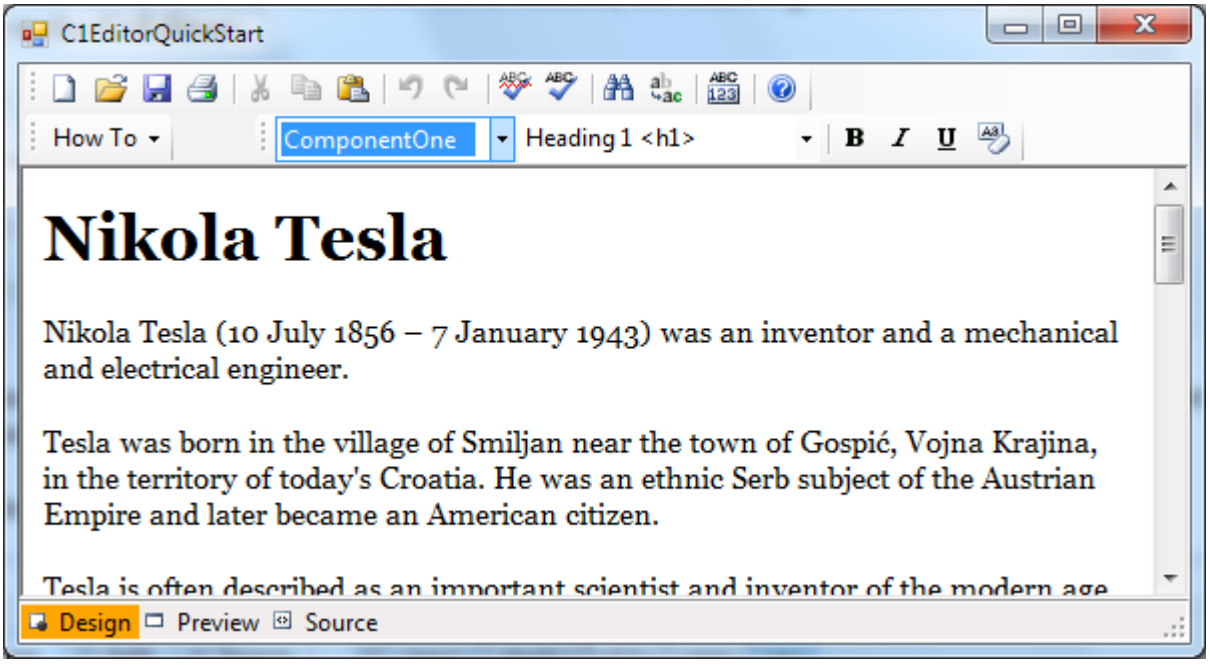
To write code in C#

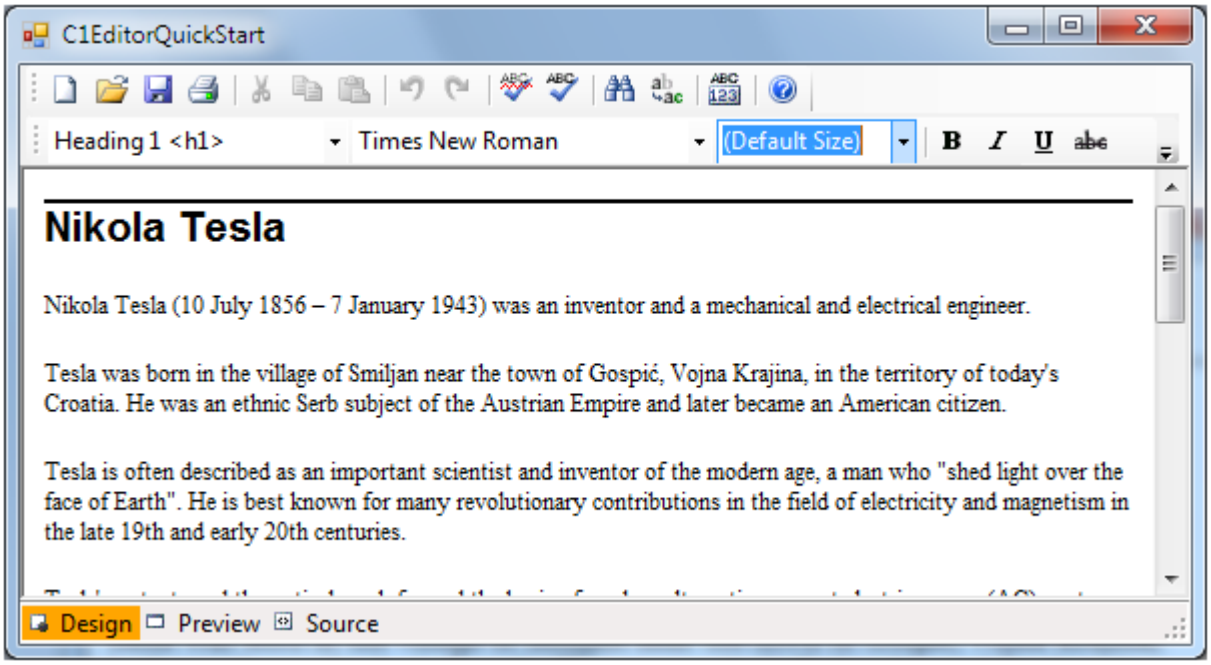
C#

```
void _cmbCss_SelectedIndexChanged(object sender, EventArgs e)
{
    var css = _css[_cmbCss.Text];
    using (var s = new MemoryStream(Encoding.UTF8.GetBytes(css)))
    {
        Editor.LoadDesignCSS(s);
    }
}
```

The code retrieves the style sheet from the dictionary, then creates a stream from the style sheet string, and calls the LoadDesignCSS method to apply the style sheet to the document.

The images below show the same document with different style sheets applied to it:





The ability to separate appearance from content by using style sheets is extremely important. Using this mechanism, you can update the appearance of entire document libraries by changing a single style sheet file. You can also create different style sheets for use when posting documents on web pages or print versions for example.

Showing and Applying Styles

The second combo box in the toolstrip allows users to apply styles to the current selection. It is also implemented in two parts. First, the combo box is populated with the standard XHTML paragraph styles:

To write code in C#

C#

```
ToolStripComboBox AddStyleCombo()
{
    ToolStripComboBox cmb = new ToolStripComboBox();
    cmb.AutoSize = false;
    cmb.Width = 150;

    cmb.Items.Add("(None)");
    cmb.Items.Add("Paragraph <p>");
    for (int i = 1; i < 7; i++)
    {
        cmb.Items.Add(string.Format("Heading {0} <h{0}>", i));
    }
    cmb.Items.Add("Unordered List <ul>");
    cmb.Items.Add("Ordered List <ol>");
    cmb.Items.Add("Pre-formatted <pre>");

    Items.Add(cmb);
    return cmb;
}
Next, three event handlers are used to apply the selected style when the user
```

```
interacts with the combo box:
void _cmbStyle_LostFocus(object sender, EventArgs e)
{
    ValidateAndApplyStyle();
}
void _cmbStyle_SelectedIndexChanged(object sender, EventArgs e)
{
    ValidateAndApplyStyle();
}
void _cmbStyle_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        ValidateAndApplyStyle();
    }
}
```

Next, three event handlers are used to apply the selected style when the user interacts with the combo box:

To write code in C#

C#

```
void ValidateAndApplyStyle()
{
    string style = _cmbStyle.Text;
    int start = style.IndexOf('<');
    int end = style.IndexOf('>');
    if (end > -1 && end > start)
    {
        var tag = style.Substring(start + 1, end - start - 1);
        Editor.Selection.ApplyTag(tag);
    }
}
```

The method starts by retrieving the tag from the combo box, and then applies the style to the current selection using the **Selection** object's **ApplyTag** method.

In addition to the **ApplyTag** method, the **Selection** object also provides **RemoveTag** and **IsTagApplied** methods for managing the tags applied to the selection.

Note that although these tags affect the appearance of the document, they are really related to the structure of the document. The actual appearance of a **<p>** or **<h1>** tag for example is defined by the style sheet that is currently applied to the document.

Bold, Italic, Underline, Clear Formatting

The bold, italic, underline, and clear formatting commands are very easy to implement, they simply delegate the command to corresponding methods in the C1Editor:

To write code in C#

C#

```
void Bold_Click(object sender, EventArgs e)
```

```
{
    _btnBold.Checked = SelectionFontBold = !SelectionFontBold;
}
void Italic_Click(object sender, EventArgs e)
{
    _btnItalic.Checked = SelectionFontItalic = !SelectionFontItalic;
}
void Underline_Click(object sender, EventArgs e)
{
    _btnUnderline.Checked = SelectionFontUnderline = !SelectionFontUnderline;
}
void ClearFormatting_Click(object sender, EventArgs e)
{
    Selection s = Editor.Selection;
    s.ClearFormatting();
}
```

The first three commands use helper properties defined by the **ToolStripStyles** class: **SelectionFontBold**, **SelectionFontItalic**, and **SelectionFontUnderline**. These helper properties are implemented as follows:

To write code in C#

C#

```
private bool SelectionFontBold
{
    get
    {
        return Editor.Mode == EditorMode.Design
            ? Editor.Selection.IsTagApplied("strong")
            : false;
    }
    set
    {
        if (value)
        {
            Editor.Selection.ApplyTag("strong");
        }
        else
        {
            Editor.Selection.RemoveTag("strong");
        }
    }
}
```

The implementation uses the **C1Editor**'s Selection property, which returns a **Selection** object that represents the current selection. The **Selection** object has methods to apply, remove, and check whether formatting tags are applied to the selection.

The same logic used above is used to implement the **SelectionFontItalic** and **SelectionFontUnderline** helper properties.

Enabling and Disabling Commands

The commands in the **ToolStripStyles** class need to be updated when the selection changes, so the bold button is checked when the selected text is bold for example.

As before, this task is performed by the **UpdateState** method:

To write code in C#

C#

```
public override void UpdateState()
{
    if (_selectionChanged)
    {
        _selectionChanged = false;
        ShowStyles();
    }
}
```

To avoid updating the toolstrip commands too frequently, we use a *selectionChanged* flag to keep track of changes in the selection. If there were any changes since the last time the method was invoked, then the **ShowStyles** method is called. Here is the implementation:

To write code in C#

C#

```
void ShowStyles()
{
    // show inline styles
    Selection s = Editor.Selection;
    if (s != null)
    {
        _btnBold.Checked = SelectionFontBold;
        _btnItalic.Checked = SelectionFontItalic;
        _btnUnderline.Checked = SelectionFontUnderline;
    }

     // find selected style
    XmlNode node = GetSelectedNode();
    if (node == null)
    {
        _cmbStyle.SelectedIndex = 0;
    }
    else
    {
        bool found = false;
        while (node != null && !found)
        {
            string style = string.Format("<{0}>", node.Name);
            foreach (string item in _cmbStyle.Items)
            {
                if (item.IndexOf(style) > -1)
```

```
                {
                    _cmbStyle.Text = item;
                    found = true;
                    break;
                }
            }
            node = node.ParentNode;
        }
    }
}
```

The method starts by updating the state of the bold, italic, and underline buttons using the helper properties described earlier. Then it calls the **GetSelectedNode** to retrieve the **XmlNode** that represents the current selection and looks for the matching node type in the *cmbStyle* combo box. If a match is not found, parent nodes are scanned until a match is found or until we reach the top level node in the document.

The **GetSelectedNode** method is implemented as follows:

To write code in C#

C#

```
XmlNode GetSelectedNode()
{
     // return node if start and end nodes are the same
    Selection selRange = Editor.Selection;
    if (selRange != null)
    {
        XmlNode startNode = selRange.Start.Node;
        XmlNode endNode = selRange.End.Node;
        return object.Equals(startNode, endNode)
            ? startNode
            : null;
    }
    return null;
}
```

The method starts by retrieving the Selection property, then checking that its start and end nodes are the same. If the selection spans multiple nodes, the method returns null which indicates there is no single style representing the selection.

Performing Other Common Tasks

The sections above described how to perform common tasks using the **C1Editor**'s object model, focusing on the types of task that would be implemented in toolbars.

This section focuses on other types of task that are commonly needed when implementing methods that process documents programmatically. Specifically, we will show you how to select and replace text in the document using the **C1Editor**'s **Selection** object and using the **XmlDocument** object of the current document. Both tasks can be performed using either approach, which can be useful in different situations.

Selecting a Paragraph using the Select Method

The Select method in the C1Editor is equivalent to the familiar **Select** method in the **TextBox** and **RichTextBox** controls. You pass the index of the first character in the selection and the selection length, and the current selection is updated.

The code below shows how you could use the Select method to select the sixth paragraph in the current document:

To write code in C#

C#

```
void selectParagraph_Click(object sender, EventArgs e)
{
    // get text (notice new line handling)
    string txt = this.c1Editor1.Text;
    txt = txt.Replace("\r\n", "\n");

    // find 6th paragraph
    int start = IndexOf(txt, '\n', 6) + 1;
    int len = IndexOf(txt, '\n', 7) - start;

    // select the paragraph
    c1Editor1.Select(start, len);
}
```

The code starts by retrieving the Text property and replacing *cr/lf* combinations with simple *lf* characters. This is necessary to get the selection indices to match the content of the document. Next, the code uses the **IndexOf** helper method to find the position of the sixth paragraph. Finally, it selects the paragraph using the Select method.

The implementation of the **IndexOf** method is as follows:

To write code in C#

C#

```
int IndexOf(string text, char chr, int count)
{
    for (int index = 0; index < text.Length; index++)
    {
        if (text[index] == chr)
        {
            count--;
            if (count <= 0)
            {
                return index;
            }
        }
    }
    return text.Length;
}
```

Selecting a Paragraph using the XmlDocument

To perform the same task using the **XmlDocument** object model, you would use this code:

To write code in C#

C#

```
void selectXmlNode_Click(object sender, EventArgs e)
{
    // find 6th node in document
    XmlDocument doc = c1Editor1.Document;
    XmlNodeList nodes = SelectNodes(doc, "/html/body/p");
    if (nodes.Count > 5)
    {
        XmlNode node = nodes[5];
        C1TextRange range = c1Editor1.CreateRange();
        range.MoveTo(node);
        range.Select();
    }
}
```

The code is considerably simpler. It retrieves the current document from the C1Editor, and then uses the **SelectNodes** helper method to get a list of all paragraph tags. It then creates a C1TextRange object, moves this new range to the node that represents the sixth paragraph in the document, and selects the range using the **C1TextRange**'s Select method.

The **SelectNodes** method is a handy utility that takes care of XML namespaces by automatically creating and using an **XmlNamespaceManager** (this is standard **XmlDocument** logic, not directly related to the C1Editor):

To write code in C#

C#

```
XmlNodeList SelectNodes(XmlDocument doc, string xpath)
{
    if (doc.DocumentElement.Attributes["xmlns"] != null)
    {
        // add namespace manager
        string xmlns = doc.DocumentElement.Attributes["xmlns"].Value;
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(doc.NameTable);
        nsmgr.AddNamespace("x", xmlns);
        xpath = xpath.Replace("/", "/x:");
        return doc.SelectNodes(xpath, nsmgr);
    }
    else
    {
        return doc.SelectNodes(xpath);
    }
}
```

Search and Replace with the Selection object

Another common task for a text editor is search/replace functionality. The C1Editor has built-in commands for find and replace (implemented above using the ShowDialog command), but in some cases you may want to implement this programmatically.

The code below shows how you can perform search and replace tasks using the **Selection** object:

To write code in C#

C#

```
void replaceUsingSelection_Click(object sender, EventArgs e)
{
    // strings to search and replace
    string search = "user";
    string replace = "customer";

    // do the work
    int count = 0;
    for (int start = 0; ; start += search.Length)
    {
        // get text (notice new line handling)
        string txt = this.c1Editor1.Text;
        txt = txt.Replace("\r\n", "\n");

        // find text, break when done
        start = txt.IndexOf(search, start, StringComparison.OrdinalIgnoreCase);
        if (start < 0)
        {
            break;
        }

        // select match
        this.c1Editor1.Select(start, search.Length);

        // replace text and make it bold
        this.c1Editor1.Selection.Text = replace;
        this.c1Editor1.Selection.ApplyTag("strong");
    }

    // done
    MessageBox.Show(string.Format("Done, {0} instances found.", count));
}
```

The code implements a loop that gets the editor's content as plain text and looks for a "search" string using the standard string.**IndexOf** method. When a match is found, the code selects the string using the Select method and then replaces the text by modifying the **Text** property of the **Selection** object. As a bonus, the code also makes the replacement bold by applying the "strong" tag to the replacement text.

A good degree of compatibility with the familiar object model of the **TextBox** and **RichTextBox** classes is an important feature of the C1Editor.

Search and Replace with the XmlDocument

To perform the same search/replace task with the **XmlDocument** object model, you would use the following code:

To write code in C#

C#

```
void replaceUsingXmlDocument(object sender, EventArgs e)
{
```

```
    XmlDocument doc = this.c1Editor1.Document;
    XmlNodeList nodes = SelectNodes(doc, "/html/body");
    if (nodes.Count > 0)
    {
        ReplaceNodeText(nodes[0], "user", "customer");
    }
}
```

The code uses the **SelectNodes** method described earlier to select a single node that represents the document's body. It then calls the helper method **ReplaceNodeText** given below:

To write code in C#

C#

```
void ReplaceNodeText(XmlNode node, string search, string replace)
{
    foreach (XmlNode child in node.ChildNodes)
    {
        ReplaceNodeText(child, search, replace);
    }
    if (node.NodeType == XmlNodeType.Text)
    {
        node.InnerText = node.InnerText.Replace(search, replace);
    }
}
```

ReplaceNodeText then takes an **XmlNode** as a parameter and scans all the node's children by calling itself recursively until it reaches a text node. At that point, it uses the regular **string.Replace** method to modify the content of the node.

The changes made to the **XmlDocument** are automatically reflected in the C1Editor. Exposing the power and flexibility of the **XmlDocument** object model for processing XHTML documents is one of the main advantages of the C1Editor control.

Editor for WinForms Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos which may make use of other development tools included with the ComponentOne Studio.

Please refer to the pre-installed product samples through the following path:

Documents\ComponentOne Samples\WinForms

The following tables provide a short description for each sample.

Visual Basic Samples

Sample	Description
C1WordsX	This sample shows how you can use the C1Editor control to build a rich text editor. The editor contains a Ribbon control and shows how you can implement formatting, search and replace, tables, images, spell-checking, PDF export, and other advanced text editing features using the C1Editor control.
CustomDialogs	This sample demonstrates how to create custom dialogs for: - inserting/editing hyperlinks, bookmarks, picture, objects and tables; - implementing find/replace commands; - implementing basic text formatting. The C1Editor control contains built-in dialogs for all these tasks, but on some occasions you may want to implement your own in order to customize their appearance or behavior.
CustomTags	This sample shows how you can define and use your own custom tags.
EditModes	This sample shows how the Mode property works. The application contains a form with a C1Editor control. You can set the Mode property of the C1Editor using the buttons at the bottom of the form.
PrintTemplate	This sample demonstrates how you can customize the print and print preview features.
ToolStrips	This sample shows how you can use toolstrips from the C1.Win.Editor.ToolStrips namespace.
SyntaxHighlight	This sample shows how to implement a syntax-highlighting editor using a C1Editor.
UserCSS	This sample shows how you can customize the appearance of the document using custom CSS files.
C1EditorQuickStart	This sample shows how to implement a basic text editor application using the C1Editor control.
ContentEditable	This sample shows how you can keep parts of the document from being edited.
HtmlEvents	Demonstrates abilities of the new event HtmlEvent.

C# Samples

Sample	Description
C1WordsX	This sample shows how you can use the C1Editor control to build a rich text editor. The editor contains a Ribbon control and shows how you can implement formatting, search and replace, tables, images, spell-checking, PDF export, and other advanced text editing features using the C1Editor control.
CustomDialogs	This sample demonstrates how to create custom dialogs for: - inserting/editing hyperlinks, bookmarks, picture, objects and tables; - implementing find/replace commands; -

Sample	Description
	implementing basic text formatting. The C1Editor control contains built-in dialogs for all these tasks, but on some occasions you may want to implement your own in order to customize their appearance or behavior.
CustomTags	This sample shows how you can define and use your own custom tags.
EditModes	This sample shows how the Mode property works. The application contains a form with a C1Editor control. You can set the Mode property of the C1Editor using the buttons at the bottom of the form.
PrintTemplate	This sample demonstrates how you can customize the print and print preview features.
ToolStrips	This sample shows how you can use toolstrips from the C1.Win.Editor.ToolStrips namespace.
SyntaxHighlight	This sample shows how to implement a syntax-highlighting editor using a C1Editor.
UserCSS	This sample shows how you can customize the appearance of the document using custom CSS files.
C1EditorQuickStart	This sample shows how to implement a basic text editor application using the C1Editor control. Supports Ctrl+P, Ctrl+N, and Ctrl+O shortcuts' custom handling.
ContentEditable	This sample shows how you can keep parts of the document from being edited.
HtmlEvents	Demonstrates abilities of the new event HtmlEvent.

Editor for WinForms Task-Based Help

The task-based help section assumes that you are familiar with programming in the Visual Studio environment and have a general understanding of the C1Editor control.

Each topic provides a solution for specific tasks using the C1Editor control. By following the steps outlined in each topic, you will be able to create projects using a variety of C1Editor features.

Each task-based help topic also assumes that you have created a new .NET project and added a C1Editor control to the form.

Changing the C1Editor Mode

The C1Editor control features three editor modes: **Design**, **Source**, and **Preview**. You can determine which of these views users will see initially by setting the Mode property.

1. In Visual Studio, click the **View** menu and select **Code** to switch to **Source** view, if necessary.
2. Add the following statement to your project.

   To write code in Visual Basic

   ```
   Visual Basic
   Imports C1.Win.C1Editor
   ```

   To write code in C#

   ```
   C#
   using C1.Win.C1Editor;
   ```

3. Add the following code to the **Page_Load** event to set the Mode property.

   To write code in Visual Basic

   ```
   Visual Basic
   C1Editor1.Mode = EditorMode.Source
   ```

   To write code in C#

   ```
   C#
   c1Editor1.Mode = EditorMode.Source;
   ```

   Note: Please note this sample changes the editor mode to Source. You can also set this property to Design or Preview.

4. Press **F5** to build the project and observe C1Editor opens in **Source** view.

Binding C1Editor to a Document

You can bind the C1Editor control to a document specified in the Document property. If the document is edited within the C1Editor, the underlying XmlDocument syncs to match it. If the XmlDocument changes in code, these changes are visible in the C1Editor control at run time.

1. In Visual Studio, click the **View** menu and select **Code** to switch to **Source** view, if necessary.
2. Add the following statements to your project.

To write code in Visual Basic

```
Visual Basic
Imports C1.Win.C1Editor
Imports System.Xml
```

To write code in C#

```
C#
using C1.Win.C1Editor;
using System.Xml;
```

3. Add the following code to the **Page_Load** event to set the Document property.

To write code in Visual Basic

```
Visual Basic
Dim myDoc as new XmlDocument()
C1Editor1.Document = myDoc
```

To write code in C#

```
C#
XmlDocument myDoc = new XmlDocument();
c1Editor1.Document = myDoc;
```

Loading an XHTML Document from a File

You can load an XHTML document into the C1Editor using the LoadXml method.

1. In the Visual Studio Solution Explorer, right-click the project name, select **New Folder**, and name it **Xhtml**.
2. Place the XHTML document you would like to load into this folder. For this example, we will use a document named "galaxy.htm".
3. Click the **View** menu and select **Code** to switch to **Source** view, if necessary.
4. Add the following statements to your project.

To write code in Visual Basic

```
Visual Basic
Imports C1.Win.C1Editor
Imports System.IO
```

To write code in C#

```
C#
using C1.Win.C1Editor;
using System.IO;
```

5. Add the following code to the **Page_Load** event. This code uses the LoadXml method to load your XHTML document into the C1Editor. You will need to update the file path to the location of the XHTML document on your machine.

   To write code in Visual Basic

   ```
   Visual Basic
   C1Editor1.LoadXml("C:\galaxy.htm")
   ```

   To write code in C#

   ```
   C#
   c1Editor1.LoadXml(@"C:\galaxy.htm");
   ```

6. Run the project and the C1Editor will show your XHTML document.

Linking a ToolStrip to C1Editor

You can link one of the four built-in ToolStrips to C1Editor by setting the Editor property.

Setting the Editor property using the Properties Window

To set the Editor property in the Visual Studio Properties window, follow these steps:

1. Right-click your **EditorToolStrip** and select **Properties**.
2. In the Visual Studio Properties window, click the drop-down arrow next to the Editor property and select your C1Editor.

Note: If you expand the Editor property node, you can set other properties for the C1Editor here.

Setting the Editor property Programmatically

Add the following code to your form, in the **Form_Load** event, for example:

To write code in Visual Basic

```
Visual Basic
C1EditorToolStripMain1.Editor = C1Editor1
```

To write code in C#

```
C#
c1EditorToolStripMain1.Editor = c1Editor1;
```

Creating a Custom ToolStrip

You can implement your own toolstrip to use with C1Editor by using the C1EditorToolStripBase as a base class and adding buttons.

The following code is an example of a toolbar with three buttons: Cut, Copy, and Paste.

To write code in C#

C#

```
using C1.Win.C1Editor;
using C1.Win.C1Editor.ToolStrips;

namespace C1EditorCustomToolStrip
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
          public enum CommandButton
    {
        Cut,
        Copy,
        Paste
    }
         public class MyToolStrip : C1EditorToolStripBase
        {
            protected override void OnInitialize()
            {
                AddButton(C1.Win.C1Editor.ToolStrips.CommandButton.Cut);
                AddButton(C1.Win.C1Editor.ToolStrips.CommandButton.Copy);
                AddButton(C1.Win.C1Editor.ToolStrips.CommandButton.Paste);
            }
        }
   public class C1EditorToolStripButton : ToolStripButton
        {
            public C1Editor Editor { get; set; }
            public CommandButton Command { get; set; }
        }
         private void Form1_Load(object sender, EventArgs e)
        {
            // Using MyToolStrip in an application:

            MyToolStrip toolStrip = new MyToolStrip();
            this.Controls.Add(toolStrip);
            toolStrip.Editor = c1Editor1;
        }
    }
}
```

Selecting Characters in the C1Editor

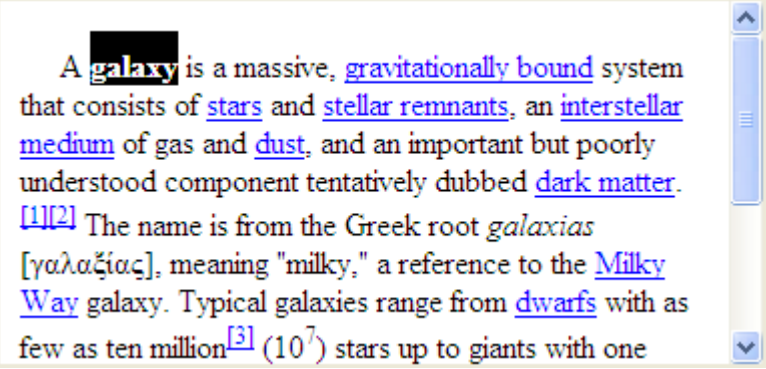
You can use the SelectionStart property to specify the first character to select in the range. Use the SelectionLength property to specify how many characters should be selected.

To write code in C#

C#

```
private void button1_Click(object sender, EventArgs e)
        {
            C1Editor1.SelectionStart = 2;
            C1Editor1.SelectionLength = 6;
        }
```

For example, the following code starts the selection after the second character in the C1Editor and continues until six characters are selected:



Using a Cascading Style Sheet with C1Editor

1. Create a .NET project and add a C1Editor and **Button** control to the form.
2. In the Solution Explorer, double-click **Resources.resx**, click **Add Resource**, and select **Add New String**.
3. Enter **sEditorText** as the **Name** of the resource.
4. In the **Value** column, enter some content for the .xml document. You can copy this sample content and enter it if you prefer.

   Sample content

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE html SYSTEM
"C:\Users\jjj\AppData\Local\Temp\tmpB1BB.tmp"[]><html
xmlns="http://www.w3.org/1999/xhtml"><head><title>Famous
Pittsburgers</title><meta http-equiv="Content-Type"
content="text/html; charset=utf-8" /></head><body><h1>Famous
Pittsburghers</h1><p class="BodyText">Many famous and successful
people were born and raised in Pittsburgh; although they found
fame in other cities, they called Pittsburgh home.</p><p
class="C1SectionCollapsed">Actors/Comedians</p><ul><li><p
class="C1HBullet">Gene Kelly (Dancer and star of "Singin' in the
Rain")</p></li><li><p class="C1HBullet">Michael Keaton (Star of
"Batman," "Mr. Mom," and many other movies)</p></li><li><p
class="C1HBullet">Dennis Miller ("Saturday Night Live" cast
member and TV/radio talk show host) </p></li></ul><p
class="C1SectionCollapsed">TV Legend</p><ul><li><p
class="C1HBullet">Fred Rogers (Producer and host of PBS's "Mister
Rogers' Neighborhood")</p></li></ul><p
class="C1SectionCollapsed">Artists</p><ul><li><p
class="C1HBullet">Mary Cassatt (Impressionist
painter)</p></li><li><p class="C1HBullet">Andy Warhol (Pop
artist; Pittsburgh is the home of the Andy Warhol Museum)
</p></li></ul><p class="C1SectionCollapsed">Sports
Legends</p><ul><li><p class="C1HBullet">Dan Marino (Quarterback
of the Miami Dolphins and University of Pittsburgh
Panthers)</p></li><li><p class="C1HBullet">Joe Montana (Super
Bowl champion quarterback of the San Francisco
49ers)</p></li><li><p class="C1HBullet">Joe Namath (Outspoken New
York Jets quarterback; victorious in Super Bowl
```

```
III)</p></li><li><p class="C1HBullet">Johnny Unitas (Quarterback
of the Baltimore Colts for 17 years)</p></li><li><p
class="C1HBullet">Arnold Palmer (Golf legend and winner of more
than 60 PGA events, including the Masters and the U.S.
Open)</p></li></ul><p
class="C1SectionCollapsed">Singers</p><ul><li><p
class="C1HBullet">Perry Como (Recorded over 100 hit
singles)</p></li><li><p class="C1HBullet">Bobby Vinton ("Roses
are Red" and "Blue Velvet" hit #1 on the Billboard
charts)</p></li></ul><p
class="C1SectionCollapsed">Writers</p><ul><li><p
class="C1HBullet">Rachel Carson (Author of "Silent Spring," as
well as a pioneering
environmentalist)</p></li></ul></body></html>
```

5. Go back to your form and double-click it.
6. In the **Form_Load** event, enter code to load the .xml document so it looks like the following:

   To write code in C#

   C#

```
private void Form1_Load(object sender, EventArgs e)
        {
            c1Editor1.LoadXml(Resources.sEditorText, null);
        }
```

Load a CSS

1. Create a CSS and add it to your project by right-clicking the project name in the Solution Explorer, select **Add Existing Item**, choosing the CSS, and clicking **Add**. You can use this sample CSS if you don't already have one.

   Sample CSS

```
html {

        font-family: Verdana;

        font-size: 10pt;

        line-height: normal;

        margin-bottom: 0pt;

        margin-left: 0pt;

        margin-right: 0pt;

        margin-top: 0pt;

}



h1 {

        font-family: Verdana;

        font-size: 20pt;
```

```
        font-weight: bold;
        line-height: normal;
        margin-bottom: 8pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 10pt;
}

h2 {
        font-family: Verdana;
        font-size: 16pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 7pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 9pt;
        page-break-after: avoid;
}

h3 {
        font-family: Verdana;
        font-size: 16pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 7pt;
        margin-left: 0pt;
```

```
        margin-right: 0pt;
        margin-top: 9pt;
        page-break-after: avoid;
}

h4 {
        font-family: Verdana;
        font-size: 12pt;
        font-weight: bold;
        line-height: normal;
        margin-bottom: 2pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 2pt;
        page-break-after: avoid;
}

.C1HBullet {
        font-family: Verdana;
        font-size: 10pt;
         font-style: italic;
        line-height: 14pt;
        margin-bottom: 0pt;
        margin-left: 18pt;
        margin-right: 0pt;
        margin-top: 5pt;
        text-indent: -18pt;
```

```
}


p {
        font-family: Verdana;
        font-size: 10pt;
        line-height: 14pt;
        margin-bottom: 0pt;
        margin-left: 0pt;
        margin-right: 0pt;
        margin-top: 5pt;
         text-indent: 18pt;
}


.C1SectionCollapsed {
        font-weight: bold;
}
```

2. Add a **button1_Click** event and use the LoadDesignCSS method:

   To write code in C#

   C#

```
privatevoid button1_Click(object sender, EventArgs e)
        {
            c1Editor1.LoadDesignCSS(@"C:\CSS1.css");
        }
```

Run the Project

1. Press F5 to run the project. Notice the document is loaded in the C1Editor.
2. Click the button. The document is formatted with the CSS.

For a detailed example of using custom cascading style sheets, see the **UserCSS** sample that comes with this product.