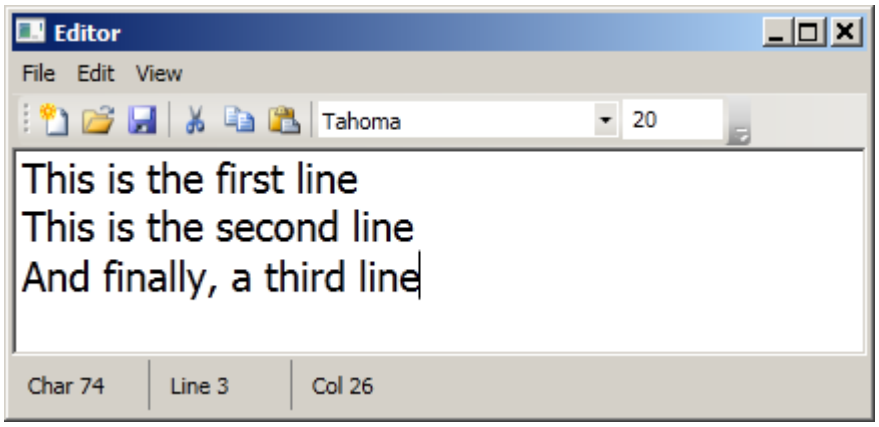
Lab 3

Toolbar and Status Bar for the Simple Editor

Introduction

In this lab you will enhance your simple editor by providing a toolbar and a status bar. You will provide an icon on a number of menu items, corresponding to the toolbar button images. You will also implement a checkable menu item to show or hide the status bar.



Suggested Time: 45 minutes

Root Directory: **OIC\WpfCs**

Directories:		
	Labs\Lab7\Editor	(do your work here)
	Chap06\Editor\Step5	(backup of starter code)
	Chap07\Editor\Step6	(answer)

Part 1. Implement a Toolbar

1. If you would like to continue with your own version of the Editor, delete the supplied starter files and replace them with your files from the Lab6B folder. Build and run your starter code.

2. Add the following to the XAML file, after the definition of the menu. Note that the various image files are in the directory **C:\OIC\Data\Grahpics**.

```
<ToolBarTray Name="tbTray"
             DockPanel.Dock="Top">
   <ToolBar>
     <Button Command="New">
       <Image Source="c:\OIC\Data\Graphics\new.png" />
     </Button>
```

```
    <Button Command="Open">
      <Image Source="c:\OIC\Data\Graphics\open.png" />
    </Button>
    <Button Command="Save">
      <Image Source="c:\OIC\Data\Graphics\save.png" />
    </Button>
    <Separator/>
    <Button Command="Cut">
      <Image Source="c:\OIC\Data\Graphics\cut.png" />
    </Button>
    <Button Command="Copy">
      <Image Source="c:\OIC\Data\Graphics\copy.png" />
    </Button>
    <Button Command="Paste">
      <Image Source="c:\OIC\Data\Graphics\paste.png" />
    </Button>
  </ToolBar>
</ToolBarTray>
```

3. Build and run. You'll hit an exception. If you run under the debugger, you'll see that the exception is thrown on the first statement of **EditorCanExecute()**. At this point, **txtData** has not yet been initialized. Why did a similar problem not occur with menus?

4. The toolbar is always visible, but the menus have to be pulled down by the user. Thus **EditorCanExecute()** won't be called before there is some interaction from the user.

5. Now let's fix the problem. Include in **EditorCanExecute()** a test for **txtData** being **null**.

```
private void EditorCanExecute(object sender, CanExecuteRoutedEventArgs
e)
{
   if (txtData == null)
      e.CanExecute = false;
   else if (txtData.Text != "")
      e.CanExecute = true;
   else
      e.CanExecute = false;
}
```

6. Build and run. You should now have a functional toolbar without having written any additional procedural code!

7. Try to see any difference in appearance of a disabled toolbar button from one that is not disabled.

8. The disabled buttons are not grayed out, but a subtle difference is that the enabled buttons are highlighted while the mouse hovers over them.

9. Add tool tips. For example, this XAML will add a tool tip for the New button.

```
  <Button Command="New"
```

```
          ToolTip="New">
    <Image Source="c:\OIC\Data\Graphics\new.png" />
  </Button>
```

10. Build and run. Now you can see an additional behavior difference with disabled buttons: the tool tip is not shown.

11. Enhance the menu items by providing an icon with image from the corresponding toolbar button. For example, this XAML will provide an icon for the New menu item.

```
<MenuItem Header="_New"
          Command="New">
 <MenuItem.Icon>
   <Image Source="c:\OIC\Data\Graphics\new.png"/>
 </MenuItem.Icon>
</MenuItem>
```

12. Add a combo box to the toolbar for specifying the font family. Also add a text box for specifying the font size. There should be a separator between the previous buttons and these new items on the toolbar.

```
  <Separator/>
  <ComboBox Name="cmbFontFamily"
            Width="150"
            SelectionChanged="cmbFont_SelectionChanged">
  </ComboBox>
  <TextBox Name="txtFontSize"
           Width="50"
           SelectionChanged="cmbFont_SelectionChanged">
  </TextBox>
```

13. Provide code to initialize the combo box with the system font families and the text box with the current font size. You can use the same code that you had in connection with the font dialog box.

```
public Editor()
{
   InitializeComponent();
   foreach (FontFamily fam in Fonts.SystemFontFamilies)
      cmbFontFamily.Items.Add(fam.Source);
   txtFontSize.Text = txtData.FontSize.ToString();
   cmbFontFamily.Text = txtData.FontFamily.Source;
}
```

14. Implement the common handler for the **SelectionChanged** event of the combobox.

```
void cmbFont_SelectionChanged(object sender, RoutedEventArgs args)
{
   txtData.FontSize = Convert.ToDouble(txtFontSize.Text);
   txtData.FontFamily =
      new FontFamily(cmbFontFamily.SelectedItem.ToString());
}
```

15. Build and run. You get a crash, because the handler is called before **txtData** has been initialized.

16. Provide a **bool** data member **init**, set to false initially and to true after **InitializeComponent()** has been called.

```
private bool init = false;

public Editor()
{
   InitializeComponent();
   foreach (FontFamily fam in Fonts.SystemFontFamilies)
      cmbFontFamily.Items.Add(fam.Source);
   txtFontSize.Text = txtData.FontSize.ToString();
   cmbFontFamily.Text = txtData.FontFamily.Source;
   init = true;
}
```

17. In the **SelectionChanged** event handler, test that **init** is true.

```
if (init)
{
   txtData.FontSize = Convert.ToDouble(txtFontSize.Text);
   txtData.FontFamily =
      new FontFamily(cmbFontFamily.SelectedItem.ToString());
}
```

18. Build and run. You should now be able to change the font using the controls on the toolbar. Exercise all the functionality of your little editor.

Part 2. Implement a Status Bar

1. In the XAML file add a StatusBar docked to the bottom.

```
<StatusBar DockPanel.Dock="Bottom"
           Name="statEditor">
</StatusBar>
```

2. Add three label controls to the status bar with content “Char”, “Line” and “Col”. There should be a separator between them.

```
<Label Name="lblChar" Width="60">
  Char
</Label>
<Separator/>
<Label Name="lblLine" Width="60">
  Line
</Label>
<Separator/>
<Label Name="lblCol" Width="60">
  Col
</Label>
```

3. Provide a handler for the **SelectionChanged** event of the **txtData** control.

```
<TextBox Name="txtData"
          TextWrapping="Wrap"
          AcceptsReturn="True"
          VerticalScrollBarVisibility="Auto"
          SelectionChanged="txtData_SelectionChanged">
 </TextBox>
```

4. Implement the handler in the code-behind file. “Char” represents a zero-based index of the current character position of the insertion point, which can be found from the **SelectionStart** data member. “Line” represents the line number, starting from 1. This can be found via the **GetLineIndexFromCharacterIndex()** method. “Col” represents the column position starting from 1 in the current line. This can be found from the **GetCharacterIndexFromLineIndex()** method.

```
private void txtData_SelectionChanged(object sender, RoutedEventArgs
args)
{
   int iChar = txtData.SelectionStart;
   lblChar.Content = string.Format("Char {0}", iChar);
   int iLine = txtData.GetLineIndexFromCharacterIndex(iChar);
   lblLine.Content = string.Format("Line {0}", iLine + 1);
   int iCol = iChar - txtData.GetCharacterIndexFromLineIndex(iLine);
   lblCol.Content = string.Format("Col {0}", iCol + 1);
}
```

5. Build and run. Observe how the information in the status bar changes as you type or move about the insertion point.

6. As a final touch, add a new “StatusBar” menu item to the View menu. This is checkable and indicates whether the status bar is visible or not. It is initialized to be checked. There is a common handler for the **Checked** and **Unchecked** events.

```
<MenuItem Header="_StatusBar"
   IsCheckable="True"
   IsChecked="True"
   Checked="OnStatusChecked"
   Unchecked="OnStatusChecked">
</MenuItem>
```

7. Implement **OnStatusChecked()** in the code behind file. Be sure to test that **init** is true.

```
private void OnStatusChecked(object sender, RoutedEventArgs args)
{
   if (init)
   {
      MenuItem item = sender as MenuItem;
      if (item.IsChecked)
         statEditor.Visibility = Visibility.Visible;
      else
         statEditor.Visibility = Visibility.Collapsed;
   }
}
```

8. Build and run. Exercise the various features of your little editor, which should now be fully functional.