Inherited Text Control

Source	http://www.dotnetspider.com/kb/Article1107.aspx

This article will try to explain the following concepts in writing custom controls.

1. **Usage of Overriden Events.**
2. **Trapping Control Keys.**
3. **Validation of Text entry.**
4. **DefaultValue Attribute Usage.**
5. **Paste Event trapping.**
6. **Designtime - Runtime Comparison.**
7. **Windows Messages.**

Prerequisites: This article assumes user is familiar with OO Concepts, C# and Windows Forms programming.

First of all we have a TextBox control in System.Windows.Forms namespace which already has useful functionality. Here we override some of its methods / events to get the custom behavior for the Textbox. First I'll override a common event which will be raised on every keypress.

```
//
// This method captures each character and if the input is the backspace it
will just move
// the cursor to the left, this way we have changed the way backspace works.
//
protected override void OnKeyPress(KeyPressEventArgs e)
{
if ( ( Keys ) e.KeyChar == Keys.Back )
{
if ( this.SelectionStart != 0 )
{
this.SelectionStart = this.SelectionStart - 1;
e.Handled = true;
}
}
}
```

With this overridden event we can trap the key presses but what if we want to capture the

special keys. Here we need to know two things. First one is the windows messages and second the special function PreprocessMessage. This method Preprocesses input messages within the message loop before they are dispatched. The Message structure wraps messages that Windows sends. You can use this structure to wrap a message and assign it to the window procedure to be dispatched. You can also use this structure to get information about a message the system sends to your application or controls. Here is how we can use these two things to capture Delete key.

```
//
// This method captures each message and if the input is delete it will clear
// the text inside the textbox.
//
public override bool PreProcessMessage(ref Message msg)
{
Keys keyData = ( ( Keys ) ( int ) msg.WParam ) ModifierKeys;
Keys keyCode = ( ( Keys ) ( int ) msg.WParam );
if ( keyData == Keys.Delete )
{
this.Clear();
}
return false;
}
```

Now I can trap any key pressed by the user even the control keys, but what if I want to capture the right click and paste event. For handling such events we have to override another special function WndProc, with the help of this method we can easily handle the editing events like cut, copy, and paste events. All messages are sent to the WndProc method after getting filtered through the PreProcessMessage method. We also have to access the clipboard data, this is achieved by using IDataObject and Clipboard facilities available. Here is how we can achieve a custom paste operation. //

```
// This method captures each message to this control and if the message is
paste action
// it will call a custom pasting method.
//
protected override void WndProc(ref Message m)
{
IDataObject clipData = Clipboard.GetDataObject();
bool ctrlDown = false;
bool shiftDown = false;
bool handledMessage = false; // used to decide if call to base WndProc
needed.
switch ( m.Msg )
{
case 0x0100 :
{
if((int)m.WParam == 0x0011)
{
```

```
ctrlDown = true;
break;
}
if((int)m.WParam == 0x0010)
{
shiftDown = true;
break;
}
if((ctrlDown == true) && ((int)m.WParam == 0x0056))
{
handledMessage = PerformPaste((string)clipData.GetData(DataFormats.Text));
break;
}
if((shiftDown == true) && ((int)m.WParam == 0x002D))
{
handledMessage = PerformPaste((string)clipData.GetData(DataFormats.Text));
break;
}
}
break;
case 0x0101 :
{
if((int)m.WParam == 0x0011)
{
ctrlDown = false;
break;
}
if((int)m.WParam == 0x0010)
{
shiftDown = false;
break;
}
}
break;
case 0x0302 :
{
handledMessage = PerformPaste((string)clipData.GetData(DataFormats.Text));
}
break;
}
if(handledMessage == true)
{
m.Result = new IntPtr( 0 );
}
else
{
base.WndProc(ref m);
}
}

private bool PerformPaste( string text )
{
if ( this.Text == string.Empty )
{
this.Text = text;
return true;
```

```
}
else
{
return false;
}
}
```

Now lets have a property which we can use to define the way our textbox behaves. We will define an enum Validation like this which will contain all the validation types we need.

```
//
// Validation enum defining lower case and upper case members.
//
public enum Validation
{
UpperCase,
LowerCase
};
```

We will use this enum to set what kind of input the textbox accepts. In OnkeyPress override we will add this code to validate the user input to the textbox.

```
//
// This code will allow the upper / lower case letters based on the
validation type
// set in the design time.
//
if ( validationType == Validation.LowerCase )
{
if ( ( Keys )e.KeyChar <= Keys.Z && ( Keys )e.KeyChar >= Keys.A )
{
e.Handled = true;
}
}
else
{
if ( ( Keys )e.KeyChar <= ( Keys ) 'z' && ( Keys )e.KeyChar >= ( Keys ) 'a' )
{
e.Handled = true;
}
}
```

We will define a public property ValidationType to give the ability to set it design time using DefaultValue attribute to specify design time behavior of the property.

```
//
// Public property to expose the validation type in design time.
//
[DefaultValue(Validation.UpperCase), Description("This property specifies how
to validate."), Category("Design")]
public Validation ValidationType
{
get
{
return validationType;
}
set
{
validationType = value;
MessageBox.Show( validationType.ToString() );
}
}
```

When you build this and add this to the toolbox of your visual studio and add this control to a form the properties window will look like this. This screenshot explains how the DefaultValue attribute works.

~~~ End of Article ~~~