A WPF Numeric Entry Control with Popup Calculator

by Ben Watson

When WPF first shipped, there was a noticeable lack of certain controls we’ve become used to in Win32 and WinForms: Calendar, DateTimePicker, and NumericUpDown. WPF 4 adds [Calendar](http://msdn.microsoft.com/en-us/library/system.windows.controls.calendar(VS.100).aspx) and [DatePicker](http://msdn.microsoft.com/en-us/library/system.windows.controls.datepicker(VS.100).aspx), but not anything for numeric entry.

This article will detail how to create your own numeric entry control for integers. As a bonus, it will also have a popup calculator to allow you quickly manipulate the number. The first part will cover creating a basic numeric entry control with increment and decrement support, while the second part will extend this to add a calculator.

# Introduction to NumericEntryControl

For my solution I wanted something that behaved somewhat similarly to the WinForms [NumericUpdown](http://msdn.microsoft.com/en-us/library/729xt55s.aspx) control.

Here are some of the specifications I came up with:

1. Allows user to set Value, MaxValue, MinValue, Increment, and LargeIncrement
2. Text directly entered is limited to numbers
3. Pasted text is not intercepted, but when the control has lost focus it will be validated and reset to the previous value if necessary
4. Two buttons, for increment and decrement
5. Holding down the buttons with the mouse causes the number to increment continuously
6. Up and down increment and decrement by Interval
7. Page Up and Page Down increment and decrement by LargeInterval
8. This version only supports integers

# Creating the control

To begin, create a new WPF project and add a new User Control called NumericEntryControl. This will create a pair of .cs and .xaml files.

In the XAML file, change the <Grid> root element to be a <DockPanel>.

<UserControl

x:Class="NumericEntryDemo.NumericEntryControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

xmlns:my="clr-namespace:NumericEntryDemo" Width="200" Height="26">

<DockPanel>

</DockPanel>

</UserControl>

Before we add the controls, let’s add some properties to our user control to hold the values the controls will use. These are dependency properties in order to take advantage of all the WPF goodness like data binding and animation. Let’s also add standard .Net property wrappers.

 public partial class NumericEntryControl : UserControl

{

public static readonly DependencyProperty ValueProperty =

DependencyProperty.Register("Value",

typeof(Int32), typeof(NumericEntryControl),

new PropertyMetadata(0));

public static readonly DependencyProperty MaxValueProperty =

DependencyProperty.Register("MaxValue",

typeof(Int32), typeof(NumericEntryControl),

new PropertyMetadata(100));

public static readonly DependencyProperty MinValueProperty =

DependencyProperty.Register("MinValue",

typeof(Int32), typeof(NumericEntryControl),

new PropertyMetadata(0));

public static readonly DependencyProperty IncrementProperty =

DependencyProperty.Register("Increment",

typeof(Int32), typeof(NumericEntryControl),

new PropertyMetadata(1));

public static readonly DependencyProperty LargeIncrementProperty =

DependencyProperty.Register("LargeIncrement",

typeof(Int32), typeof(NumericEntryControl),

new PropertyMetadata(5));

public Int32 Value

{

get

{

return (Int32)GetValue(ValueProperty);

}

set

{

SetValue(ValueProperty, value);

}

}

public Int32 MaxValue

{

get

{

return (Int32)GetValue(MaxValueProperty);

}

set

{

SetValue(MaxValueProperty, value);

}

}

public Int32 MinValue

{

get

{

return (Int32)GetValue(MinValueProperty);

}

set

{

SetValue(MinValueProperty, value);

}

}

public Int32 Increment

{

get

{

return (Int32)GetValue(IncrementProperty);

}

set

{

SetValue(IncrementProperty, value);

}

}

public Int32 LargeIncrement

{

get

{

return (Int32)GetValue(LargeIncrementProperty);

}

set

{

SetValue(LargeIncrementProperty, value);

}

}

}

# Creating an incrementing TextBox

Add a TextBox inside the DockPanel and bind its text to the value we created in our control:

<DockPanel d:LayoutOverrides="Width">

<TextBox

x:Name="\_textbox"

Margin="2,0"

Text="{Binding Value,

Mode=TwoWay,

RelativeSource={RelativeSource FindAncestor,

AncestorLevel=1,

AncestorType={x:Type my:NumericEntryControl}},

UpdateSourceTrigger=PropertyChanged}"

HorizontalAlignment="Stretch"

HorizontalContentAlignment="Right"

VerticalContentAlignment="Center" />

</DockPanel>

This will create a TextBox that sizes itself with its parent (a feature I wanted, but is not strictly necessary) and  its text will be bound to the Value property of our UserControl.

# Handling text input

It used to be that you pointed with a mouse and entered text with a keyboard. However, it is common now to enter text with a stylus, gestures, or some future method not invented yet. Thankfully, WPF supports generic text input handling so you don’t have to concern yourself with the specific hardware.

 public NumericEntryControl()

{

InitializeComponent();

\_textbox.PreviewTextInput +=

new TextCompositionEventHandler(\_textbox\_PreviewTextInput);

}

void \_textbox\_PreviewTextInput(object sender,

TextCompositionEventArgs e)

{

if (!IsNumericInput(e.Text))

{

e.Handled = true;

return;

}

}

private bool IsNumericInput(string text)

{

foreach (char c in text)

{

if (!char.IsDigit(c))

{

return false;

}

}

return true;

}

This prevents anything except numbers from being entered, whether via character recognition or keyboard. It does not, however, prevent the user from pasting non-numeric text into the box. We’ll handle that later.

# Validating Text input

It’s problematic to validate and correct user input as they are entering it. For example, if you set the MaxValue to 100, then every time you enter 1000, it jumps to 100, it can be jarring. It’s a similar situation with text pasted into the control. What the NumericUpDown control does is handle these sort of situations when the control loses focus.

To prepare for this, when the control gains focus, we need to save the last valid value so we have something to restore to.

When the control loses focus, we need to first verify that it is a number and if so, clip it to the bounds of our MinValue and MaxValue. If anything fails, set it back to the previous value.

public partial class NumericEntryControl : UserControl

{

private int \_previousValue = 0;

public NumericEntryControl()

{

InitializeComponent();

\_textbox.PreviewTextInput +=

new TextCompositionEventHandler(

\_textbox\_PreviewTextInput);

\_textbox.GotFocus +=

new RoutedEventHandler(\_textbox\_GotFocus);

\_textbox.LostFocus +=

new RoutedEventHandler(\_textbox\_LostFocus);

}

void \_textbox\_GotFocus(object sender, RoutedEventArgs e)

{

\_previousValue = Value;

}

void \_textbox\_LostFocus(object sender, RoutedEventArgs e)

{

int newValue = 0;

if (Int32.TryParse(\_textbox.Text, out newValue))

{

if (newValue > MaxValue)

{

newValue = MaxValue;

}

else if (newValue < MinValue)

{

newValue = MinValue;

}

}

else

{

newValue = \_previousValue;

}

\_textbox.Text = newValue.ToString();

}

}

Now the control will ensure there is a consistent state upon losing focus.

# Handle arrow keys

Just because WPF can handle text input from a variety of sources in a hardware-agnostic way doesn’t mean we should ignore the particular strengths of the keyboard. Specifically, we should handle the up and down arrows.

 public NumericEntryControl()

{

...

\_textbox.PreviewKeyDown +=

new KeyEventHandler(\_textbox\_PreviewKeyDown);

}

void \_textbox\_PreviewKeyDown(object sender, KeyEventArgs e)

{

switch (e.Key)

{

case Key.Up:

IncrementValue();

break;

case Key.Down:

DecrementValue();

break;

case Key.PageUp:

Value = Math.Min(Value + LargeIncrement, MaxValue);

break;

case Key.PageDown:

Value = Math.Max(Value - LargeIncrement, MinValue);

break;

default:

//do nothing

break;

}

}

private void IncrementValue()

{

Value = Math.Min(Value + Increment, MaxValue);

}

private void DecrementValue()

{

Value = Math.Max(Value - Increment, MinValue);

}

IncrementValue() and DecrementValue() are pulled out as their own method because they’re used later in the button-handling code as well.

The code so far is a perfectly usable textbox that accepts only numbers and can be incremented using the keyboard. Typically, however, we also need to support the mouse, and for that we need buttons (unless you want to do something exotic like programs such as Photoshop and Lightroom do, where text boxes have support for incrementing gestures—that’s another article, however).

# A button you can hold down

Adding buttons to increment once per click is pretty easy, but we really want to be able to hold down the buttons and have the TextBox increment. Let’s start by adding the XAML for the buttons:

 <UserControl

x:Class="NumericEntryDemo.NumericEntryControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

xmlns:my="clr-namespace:NumericEntryDemo"

Width="200" Height="26"

>

<DockPanel d:LayoutOverrides="Width">

<Button x:Name="buttonDecrement"

DockPanel.Dock="Left"

Content="-"

Width="{Binding ActualHeight,

ElementName=buttonDecrement, Mode=Default}"

Height="{Binding ActualHeight,

ElementName=\_textbox, Mode=Default}"/>

<Button x:Name="buttonIncrement"

DockPanel.Dock="Right"

Content="+"

Width="{Binding ActualHeight,

ElementName=buttonDecrement, Mode=Default}"

Height="{Binding ActualHeight,

ElementName=\_textbox, Mode=Default}"/>

<TextBox

x:Name="\_textbox"

Margin="2,0"

Text="{Binding Value,

Mode=TwoWay,

RelativeSource={RelativeSource FindAncestor,

AncestorLevel=1,

AncestorType={x:Type my:NumericEntryControl}},

UpdateSourceTrigger=PropertyChanged}"

HorizontalAlignment="Stretch"

HorizontalContentAlignment="Right"

VerticalContentAlignment="Center" />

</DockPanel>

</UserControl>

Note that the button Width property is  bound to its own ActualHeight property, and the Height property is bound to the TextBox’s ActualHeight. This has the effect of keeping the buttons square, the same height as the TextBox. It’s an effect I wanted, but you can easily dispose of it. With these buttons, our control finally takes shape:

[image](http://www.philosophicalgeek.com/wp-content/uploads/2009/11/image9.png)

## How fast should we increment?

Before writing the code to do the incrementing as we hold the button down, it’s worth considering how fast the incrementing should occur. Ideally, we would want it to increment at about the same rate as if we were holding down the up key on the keyboard. Thankfully, the keyboard repeat rate is an operating system value that we can retrieve.

There are actually two values:

private static int \_delayRate =

System.Windows.SystemParameters.KeyboardDelay;

private static int \_repeatSpeed =

Math.Max(1, System.Windows.SystemParameters.KeyboardSpeed);

The delay rate is how long we should wait before starting the repetition, and is given in multiples of 250ms. Roughly speaking, humans can determine and control actions with lengths of time of about 200ms, so 250ms is a good value to start with. Any shorter and the repetition might start when it was not intended (say, if they just click the button instead of holding it down).

The keyboard speed is the number of times per second we should repeat—[sort of](http://msdn.microsoft.com/en-us/library/system.windows.systemparameters.keyboardspeed.aspx) (see http://msdn.microsoft.com/en-us/library/system.windows.systemparameters.keyboardspeed.aspx). The value can be 0, so because of the way I use it below I want to ensure it’s at least 1.

To allow us to hold the button down, we need to override the default mouse handling of a button which is to disable the standard LeftMouseButtonDown/Up messages. Instead, we need to handle the PreviewMouseLeftButtonDown message and its corresponding Up message.

When we handle the down message, we need to set a timer for the keyboard delay value. When we handle the timer’s tick, we need to increment (or decrement) the textbox value and change the timer’s interval to the repeat speed. This repeat speed is calculated merely by dividing 1000ms (1s) by the rate per second. There may be better ways, but this gets pretty close to the rate experienced by the keyboard on my computer. Finally, when the mouse button is released we need to stop the timer. We also do a final increment, which will cover the case where the user clicks instead of holds.

We also need to capture the mouse in case the user moves it off the button—otherwise the timer will just keep incrementing forever.

Here’s the code:

 private DispatcherTimer \_timer =

new DispatcherTimer();

private static int \_delayRate =

System.Windows.SystemParameters.KeyboardDelay;

private static int \_repeatSpeed =

Math.Max(1, System.Windows.SystemParameters.KeyboardSpeed);

private bool \_isIncrementing = false;

public NumericEntryControl()

{

...

buttonIncrement.PreviewMouseLeftButtonDown +=

new MouseButtonEventHandler(

buttonIncrement\_PreviewMouseLeftButtonDown);

buttonIncrement.PreviewMouseLeftButtonUp +=

new MouseButtonEventHandler(

buttonIncrement\_PreviewMouseLeftButtonUp);

buttonDecrement.PreviewMouseLeftButtonDown +=

new MouseButtonEventHandler(

buttonDecrement\_PreviewMouseLeftButtonDown);

buttonDecrement.PreviewMouseLeftButtonUp +=

new MouseButtonEventHandler(

buttonDecrement\_PreviewMouseLeftButtonUp);

\_timer.Tick += new EventHandler(\_timer\_Tick);

}

void buttonIncrement\_PreviewMouseLeftButtonDown(

object sender, MouseButtonEventArgs e)

{

buttonIncrement.CaptureMouse();

\_timer.Interval =

TimeSpan.FromMilliseconds(\_delayRate \* 250);

\_timer.Start();

\_isIncrementing = true;

}

void buttonIncrement\_PreviewMouseLeftButtonUp(

object sender, MouseButtonEventArgs e)

{

\_timer.Stop();

buttonIncrement.ReleaseMouseCapture();

IncrementValue();

}

void buttonDecrement\_PreviewMouseLeftButtonDown(

object sender, MouseButtonEventArgs e)

{

buttonDecrement.CaptureMouse();

\_timer.Interval =

TimeSpan.FromMilliseconds(\_delayRate \* 250);

\_timer.Start();

\_isIncrementing = false;

}

void buttonDecrement\_PreviewMouseLeftButtonUp(

object sender, MouseButtonEventArgs e)

{

\_timer.Stop();

buttonDecrement.ReleaseMouseCapture();

DecrementValue();

}

void \_timer\_Tick(object sender, EventArgs e)

{

if (\_isIncrementing)

{

IncrementValue();

}

else

{

DecrementValue();

}

\_timer.Interval =

TimeSpan.FromMilliseconds(1000.0 / \_repeatSpeed);

}

And voila! A NumericEntryControl that’s basic and easy-to-use for both keyboard and mouse.

**Further improvements**

This isn’t the last word in numeric entry controls, by any means. There are many ways to accomplish it, and this is one that worked well for me. There are a number of further enhancements you could do (and perhaps should do):

* More validation
* Ensure that MaxValue >= MinValue
* Set focus to the TextBox when the control gains focus (maybe)
* Define strokes to increment and decrement with a stylus
* The user can change the keyboard repeat rate through Control Panel. This control could be modified to listen for updates to this value.

# Adding a Popup Calculator

Now that we have a basic numeric entry control, let's extend it by adding the power of a keyboard-driven calculator popup window.

If you’ve used Quicken or the now-sadly-defunct Microsoft Money, you’ve seen a feature that pops up a calculator window whenever you hit +, –, \*, or / in a number field.

Here are some further requirements for such a control:

* It should be activated when the user types +, –, \*, or /
* It should be dismissed if the user hits Escape, Enter, or clicks anywhere outside the calculator (which is considered equivalent to Enter, not escape)
* Both mouse and keyboard input should be handled
* This version only handles integers
* It should handle Backspace key to erase previously-input character
* It should be visually compact
* It should allow the user to string computations together. A + B – C / D \* E

# Extend the Numeric Entry Control

We first need to come up with a way to trigger the calculator in our numeric entry control. This is going to simply be if the user hits any math-operation character, such as +, –, \*, or /.

First, we’ll need to define the allowed operations and a way to associate keyboard keys with math operations:

enum MathOperation

{

None = 0,

Add,

Subtract,

Multiply,

Divide

};

 Now, let’s subclass NumericEntryControl with a new class that adds some additional keyboard handling.

class NumericEntryWithCalcControl : NumericEntryControl

{

static Dictionary<Key, MathOperation> \_keytoOp =

new Dictionary<Key, MathOperation>();

static NumericEntryWithCalcControl()

{

//associate keyboard keys with a math operation

\_keytoOp[Key.Add] = MathOperation.Add;

\_keytoOp[Key.Subtract] = MathOperation.Subtract;

\_keytoOp[Key.Multiply] = MathOperation.Multiply;

\_keytoOp[Key.Divide] = MathOperation.Divide;

#if DEBUG

//aid in debugging on keyboards with

//limited layouts (like laptops)

\_keytoOp[Key.A] = MathOperation.Add;

\_keytoOp[Key.S] = MathOperation.Subtract;

\_keytoOp[Key.M] = MathOperation.Multiply;

\_keytoOp[Key.D] = MathOperation.Divide;

#endif

}

}

With this bookkeeping down, we can intercept the keystrokes and show our calculator window (which we’ll define later) in the right spot. We also should handle when the window closes.

private CalcWindow \_calc = null;

protected override void OnKeyDown(

System.Windows.Input.KeyEventArgs e)

{

MathOperation op = MathOperation.None;

if (\_keytoOp.TryGetValue(e.Key, out op))

{

e.Handled = true;

ProcessOperation(op);

}

else

{

e.Handled = false;

base.OnKeyDown(e);

}

}

private void ProcessOperation(MathOperation mathOperation)

{

//show the calc dialog and prime with Value and Op

if (mathOperation != MathOperation.None)

{

\_calc = new CalcWindow(Value, mathOperation);

\_calc.Closed += new EventHandler(calc\_Closed);

Point point = this.PointToScreen(new Point(0, 0));

//adjust location to correct for DPI settings

PresentationSource source =

PresentationSource.FromVisual(this);

double dpiX = 96.0 \*

source.CompositionTarget.TransformToDevice.M11;

double dpiY = 96.0 \*

source.CompositionTarget.TransformToDevice.M22;

\_calc.Left = point.X \* 96.0 / dpiX;

\_calc.Top = point.Y \* 96.0 / dpiY;

\_calc.Show();

}

}

void calc\_Closed(object sender, EventArgs e)

{

if (\_calc != null)

{

this.Value = \_calc.CurrentValue;

\_calc.Closed -= new EventHandler(calc\_Closed);

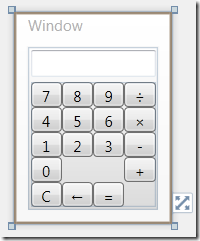
\_calc = null;

}

}

Notice in ProcessOperation that the location of the calculator window is set to the upper left corner of the NumericEntryControl and it takes into account the current DPI. This is important if you're using anything other than the standard 96 DPI.

## Create the Calculator

[[](file:///C:\Users\Ben\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles1827B0E7\calcwindow4.png)](file:///C:\Users\Ben\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles1827B0E7\calcwindow4.png)The actual calculator is a WPF window with no border (or a thin one), a textbox, and buttons representing a standard calculator layout.

The screenshot to the right shows the design layout of this window. The C button clears the input and sets the current value to 0. The left arrow erases the last character you input.

Since this window is designed to popup over other controls, I wanted to keep it as compact as possible.

The XAML code looks like this:

<Window x:Class="NumericEntryDemo.CalcWindow"

xmlns=

"http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:my="clr-namespace:NumericEntryDemo"

Title="" WindowStyle="None"

SizeToContent="WidthAndHeight"

WindowStartupLocation="Manual"

ResizeMode="NoResize"

BorderThickness="0">

<DockPanel>

<TextBox Name="\_textbox"

DockPanel.Dock="Top"

Margin="2"

IsReadOnly="True"

Text="{Binding

RelativeSource={RelativeSource FindAncestor,

AncestorType=my:CalcWindow, AncestorLevel=1},

Path=CurrentValue}"

TextAlignment="Right" />

<UniformGrid Columns="4" Rows="5"

Width="100" Height="100" Margin="2">

<Button Name="\_button7" Content="7"

Click="digitButton\_Click"/>

<Button Name="\_button8" Content="8"

Click="digitButton\_Click"/>

<Button Name="\_button9" Content="9"

Click="digitButton\_Click"/>

<Button Name="\_buttonDivide" Content="÷"

Click="opButton\_Click"/>

<Button Name="\_button4" Content="4"

Click="digitButton\_Click"/>

<Button Name="\_button5" Content="5"

Click="digitButton\_Click"/>

<Button Name="\_button6" Content="6"

Click="digitButton\_Click"/>

<Button Name="\_buttonMultiply" Content="×"

Click="opButton\_Click"/>

<Button Name="\_button1" Content="1"

Click="digitButton\_Click"/>

<Button Name="\_button2" Content="2"

Click="digitButton\_Click"/>

<Button Name="\_button3" Content="3"

Click="digitButton\_Click"/>

<Button Name="\_buttonSubtract" Content="-"

Click="opButton\_Click"/>

<Button Name="\_button0" Content="0"

Click="digitButton\_Click"/>

<!-- empty space -->

<Rectangle/>

<Rectangle/>

<Button Name="\_buttonAdd" Content="+"

Click="opButton\_Click"/>

<Button Name="\_buttonClear" Content="C"

Click="clearButton\_click"/>

<Button Name="\_buttonErase" Content="←"

Click="backButton\_Click"/>

<Button Name="\_buttonEquals" Content="="

Click="equalsButton\_Click"/>

</UniformGrid>

</DockPanel>

<Window.Background>

<LinearGradientBrush EndPoint="0.5,1"

StartPoint="0.5,0">

<GradientStop Color="White" Offset="0" />

<GradientStop Color="#FFDBDBDB" Offset="1" />

</LinearGradientBrush>

</Window.Background>

</Window>

All of the click handlers and binding targets will be filled out in the code-behind.

To start with, we need a few housekeeping variables:

* The original value upon starting the calculator, just in case after all the calculations the user just hits Escape
* A way to track whether we’ve canceled the whole thing
* The current operation
* The current input, in both string and number form
* The previously calculated value, allowing you to string together calculations like “123 + 456 / 2 \* 29”
* Map buttons and keystrokes to numbers, math operations, and the other commands.

With that, we can see the beginning of our class:

public partial class CalcWindow : Window

{

private int \_initialValue = 0;

private bool \_canceled = false;

private MathOperation \_currentOp = MathOperation.None;

//value of current input

public static readonly DependencyProperty

CurrentValueProperty = DependencyProperty.Register(

"CurrentValue", typeof(int), typeof(CalcWindow),

new PropertyMetadata(0));

//current input in string form

public static readonly DependencyProperty

CurrentInputProperty = DependencyProperty.Register(

"CurrentInput", typeof(string), typeof(CalcWindow),

new PropertyMetadata(""));

public static readonly DependencyProperty

PreviousValueProperty = DependencyProperty.Register(

"PreviousValue", typeof(int), typeof(CalcWindow),

new PropertyMetadata(0));

private static readonly Dictionary<Key, int> \_keyDigits =

new Dictionary<Key, int>();

private static readonly Dictionary<Key, MathOperation> \_keyOps =

new Dictionary<Key, MathOperation>();

private readonly Dictionary<Button, int> \_buttonDigits =

new Dictionary<Button, int>();

private readonly Dictionary<Button, MathOperation> \_buttonOps =

new Dictionary<Button, MathOperation>();

static CalcWindow()

{

\_keyDigits[Key.D0] = 0;

\_keyDigits[Key.D1] = 1;

\_keyDigits[Key.D2] = 2;

\_keyDigits[Key.D3] = 3;

\_keyDigits[Key.D4] = 4;

\_keyDigits[Key.D5] = 5;

\_keyDigits[Key.D6] = 6;

\_keyDigits[Key.D7] = 7;

\_keyDigits[Key.D8] = 8;

\_keyDigits[Key.D9] = 9;

\_keyDigits[Key.NumPad0] = 0;

\_keyDigits[Key.NumPad1] = 1;

\_keyDigits[Key.NumPad2] = 2;

\_keyDigits[Key.NumPad3] = 3;

\_keyDigits[Key.NumPad4] = 4;

\_keyDigits[Key.NumPad5] = 5;

\_keyDigits[Key.NumPad6] = 6;

\_keyDigits[Key.NumPad7] = 7;

\_keyDigits[Key.NumPad8] = 8;

\_keyDigits[Key.NumPad9] = 9;

\_keyOps[Key.Add] = MathOperation.Add;

\_keyOps[Key.Subtract] = MathOperation.Subtract;

\_keyOps[Key.Multiply] = MathOperation.Multiply;

\_keyOps[Key.Divide] = MathOperation.Divide;

#if DEBUG

\_keyOps[Key.A] = MathOperation.Add;

\_keyOps[Key.S] = MathOperation.Subtract;

\_keyOps[Key.M] = MathOperation.Multiply;

\_keyOps[Key.D] = MathOperation.Divide;

#endif

}

#region Property Wrappers

public int CurrentValue

{

get

{

return (Int32)GetValue(CurrentValueProperty);

}

private set

{

SetValue(CurrentValueProperty, value);

}

}

public string CurrentInput

{

get

{

return GetValue(CurrentInputProperty) as string;

}

private set

{

SetValue(CurrentInputProperty, value);

}

}

public int PreviousValue

{

get

{

return (Int32)GetValue(PreviousValueProperty);

}

private set

{

SetValue(PreviousValueProperty, value);

}

}

#endregion

public CalcWindow(int initialValue, MathOperation operation)

{

\_initialValue = initialValue;

\_currentOp = operation;

CurrentValue = \_initialValue;

PreviousValue = CurrentValue;

InitializeComponent();

InitializeButtons();

}

private void InitializeButtons()

{

\_buttonDigits[\_button0] = 0;

\_buttonDigits[\_button1] = 1;

\_buttonDigits[\_button2] = 2;

\_buttonDigits[\_button3] = 3;

\_buttonDigits[\_button4] = 4;

\_buttonDigits[\_button5] = 5;

\_buttonDigits[\_button6] = 6;

\_buttonDigits[\_button7] = 7;

\_buttonDigits[\_button8] = 8;

\_buttonDigits[\_button9] = 9;

\_buttonOps[\_buttonAdd] = MathOperation.Add;

\_buttonOps[\_buttonSubtract] = MathOperation.Subtract;

\_buttonOps[\_buttonMultiply] = MathOperation.Multiply;

\_buttonOps[\_buttonDivide] = MathOperation.Divide;

}

}

All that’s left is to the actual work of handling input. Since input can be from both keyboard and button clicks, let’s first define methods to handle the general operations, and we can hook up the input later.

## Generic Input Handling

First, we’ll look at the generic input handlers. These are called by the mouse and keyboard handlers and they return true so they can notify the caller of whether they actually handled the operation.

The “C” button on the calculator will not only clear the current input, but set the current value to 0 (I’m not going to bother implementing a “CE” button that clears just the current input, but it would be easy to add).

 private bool HandleClear()

{

CurrentInput = "";

CurrentValue = 0;

return true;

}

When the user hits escape, you want to close the window, after setting the value back to the initial one.

private bool HandleEscape()

{

\_canceled = true;

CurrentValue = \_initialValue;

Close();

return true;

}

We need the \_canceled flag because, as we’ll see below, when the window is deactivated, under normal circumstances we want to complete the current operation, but in the canceled case we want to do nothing.

When a user hits Enter or presses the Equals button, the current operation should be completed and the window closed. In this control, since we also want the current operation to be completed when the window is deactivated, hitting Equals will just cause the window to close and we'll handle the calculation in the deactivation code.

 private bool HandleEquals()

{

//The final operation will be

//handled when the window closes

Close();

return true;

}

This calculator provides a back button to erase the previously-entered digit. If the input becomes empty, then the current value is set to 0.

 private bool HandleBack()

{

if (!string.IsNullOrEmpty(CurrentInput))

{

CurrentInput = CurrentInput.Substring(0, CurrentInput.Length - 1);

//this is guaranteed to succeed because we validate when we add digits

if (!string.IsNullOrEmpty(CurrentInput))

{

CurrentValue = Int32.Parse(CurrentInput);

}

else

{

CurrentValue = 0;

}

}

return true;

}

When accepting new input, our button and keyboard handlers will ensure that the user can only enter digits, but we also need to make sure we handle integer overflow. An easy way to do this is to treat the new digit as a character and append it to the current input and then reparse it.

 private bool HandleDigit(int digit)

{

string newInput = CurrentInput + digit.ToString();

int temp = 0;

//make sure the full input is a number

if (Int32.TryParse(newInput, out temp))

{

CurrentInput = newInput;

CurrentValue = temp;

return true;

}

return false;

}

Finally, when the user selects an operation we need to do one of two things. If there is no current input, then we can just change the current operation to the new one. This lets you change your mind, say if you hit '+', but really wanted '-'.

On the other hand, if the user has already entered a number, then you know you’re finishing up the current expression and the user wants to calculate the results and start a new expression. As an example,  consider the expression A + B. A is the initial value of the NumericEntryControl, the + is what triggered the calculator to show up, and B is the number the user entered after the calculator was visible. When the next operation is input, A + B should be resolved, placed into A, the current operation set, and the user can now enter the next B, and so on.

private bool HandleOperation(MathOperation op)

{

if (CurrentInput.Length > 0)

{

ProcessCurrentOperation();

}

\_currentOp = op;

return true;

}

With that, we’ve seen all of the generic input handling. Now let's move on to the actual math.

**Doing the Math**

The ProcessCurrentOperation method is the heart of this control as it updates all the values according to the user’s input. Here too we need to consider what happens if the user’s calculations cause an integer overflow. To detect this we need to use a combination of the **checked** keyword and some exception handling. To simplify things, if this happens, the whole calculation is canceled and the window closed. We also handle divide by zero errors by just setting the new value to 0.

(The checked keyword tells .NET to issue exceptions when integer math overflows. The default behavior depends on compilation and environment settings, but the unchecked behavior is usually what happens).

private void ProcessCurrentOperation()

{

if (string.IsNullOrEmpty(CurrentInput))

{

return;

}

int newValue = 0;

checked

{

try

{

switch (\_currentOp)

{

case MathOperation.Add:

newValue = PreviousValue + CurrentValue;

break;

case MathOperation.Subtract:

newValue = PreviousValue - CurrentValue;

break;

case MathOperation.Multiply:

newValue = PreviousValue \* CurrentValue;

break;

case MathOperation.Divide:

if (CurrentValue != 0)

{

newValue = PreviousValue / CurrentValue;

}

else

{

newValue = CurrentValue;

}

break;

default:

Debug.Assert(false, "Invalid operation. Should never happen!");

break;

}

CurrentInput = "";

CurrentValue = newValue;

PreviousValue = newValue;

}

catch (OverflowException)

{

//will cause the window to close, so we better cancel everything first

\_canceled = true;

CurrentValue = \_initialValue;

MessageBox.Show("The result overflowed, calculation canceled.");

}

}

}

## Deactivation

As we said before, deaction should cause the current operation to complete, unless the user has canceled.

 protected override void OnDeactivated(EventArgs e)

{

if (!\_canceled)

{

ProcessCurrentOperation();

if (IsVisible)

{

//window is still visible if we've

//clicked away from the window

//before finishing calculation

Close();

}

}

base.OnDeactivated(e);

}

The only thing that remains is hooking up mouse and keyboard input to our existing methods.

## Mouse Handling

These are already referenced in the XAML, and all they do is forward to the  generic handlers.

void opButton\_Click(object sender, RoutedEventArgs e)

{

MathOperation op = \_buttonOps[sender as Button];

e.Handled = HandleOperation(op);

}

void digitButton\_Click(object sender, RoutedEventArgs e)

{

int digit = \_buttonDigits[sender as Button];

e.Handled = HandleDigit(digit);

}

private void clearButton\_click(object sender, RoutedEventArgs e)

{

e.Handled = HandleClear();

}

private void backButton\_Click(object sender, RoutedEventArgs e)

{

e.Handled = HandleBack();

}

private void equalsButton\_Click(object sender, RoutedEventArgs e)

{

e.Handled = HandleEquals();

}

## Keyboard Handling

To handle the keyboard, we need to preempt the normal message flow by looking at the preview messages (which flow from the top of the UI hierarchy down, rather than the normal messages which bubble from the bottom to the top). If we don’t do this, then when we hit keys like Enter, they go to the button with the current focus, when instead we want it to always cause the Equals functionality.

 protected override void OnPreviewKeyDown(KeyEventArgs e)

{

/\*take over all keyboard input

\* this is important because if a button has focus

\* and we hit enter then the button will be clicked

\* \*/

int digit = 0;

MathOperation op = MathOperation.None;

if (\_keyDigits.TryGetValue(e.Key, out digit))

{

e.Handled = HandleDigit(digit);

}

else if (\_keyOps.TryGetValue(e.Key, out op))

{

e.Handled = HandleOperation(op);

}

else

{

switch (e.Key)

{

case Key.Enter:

e.Handled = HandleEquals();

break;

case Key.Escape:

e.Handled = HandleEscape();

break;

case Key.Back:

e.Handled = HandleBack();

break;

case Key.C:

e.Handled = HandleClear();

break;

default:

e.Handled = false;

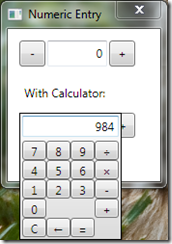
break;

}

}

base.OnPreviewKeyDown(e);

}

[[](file:///C:\Users\Ben\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles1827B0E7\calc_demo%5b4%5d.png)](file:///C:\Users\Ben\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles1827B0E7\calc_demo%5b4%5d.png)There you have it, a working pop-up calculator, usable from any other control you can subclass. As is often the case, there are a lot more features you could add to this basic implementation:

* Undo
* Floating-point support
* Button highlights with corresponding key presses
* More advanced display, showing current operator, previous values, etc.
* More robust handling of overflow