Input Events

Estimated time for completion: 45 minutes

Overview:

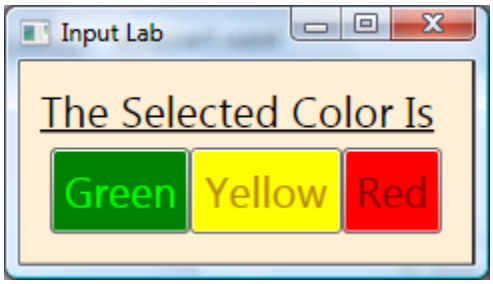
In this lab you will use **events** and **commands** to handle user input.

Goals:

- Understand routed input events
- Handle events for child elements
- Learn how to use `Commands` to manage user input

Part 1 – Listening to events

In this part, you'll create a new application. In the main window, a list of buttons will display different colors. When a button is clicked, the color of the button is applied to a Label. The finished application should look like:



1. Create a new WPF application using Visual Studio and build the above user interface. If you need help, follow the step-by-step instructions below:
    a. Set the `FontSize` to "14pt" on the Window and the `SizeToContent` property to "WidthAndHeight" to size it automatically. In the above screen shot, the Background is set to "PapayaWhip" for fun.
    b. Replace the root layout panel `Grid` with a `StackPanel` and the `Margin` to "10" to offset it a bit from the window parent.
    c. In the `StackPanel`, add a `TextBlock` with the `TextDecorations` set to "Underline" – give it a name to access it from code behind.

d. Add a second horizontal `StackPanel` and inside it, add three `Buttons` for **Green**, **Yellow** and **Red**. Set the `Background` and `Foreground` properties to get the colors desired. You can also set some Padding to get it to look like the above screen shot.

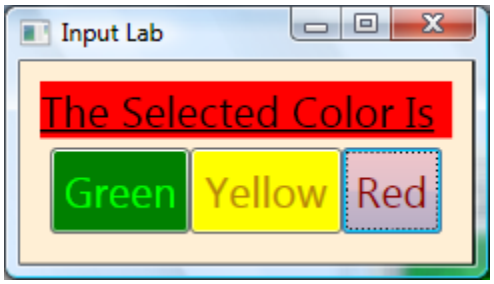
2. Set the `Click` event on each button to a new method you'll create shortly named `OnButtonClick`.
3. Go to the code file for the window. Add a method with the following signature:

```
void OnButtonClick(object sender, RoutedEventArgs args)
```

4. Inside this method, cast the sender to a `Button` and set the background of the `TextBlock` to the background of the sender.

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
   Button buttonSender = (Button)sender;
   TextBlock1.Background = buttonSender.Background;
}
```

5. Compile and run. Verify that the background of the `TextBlock` changes when you click on any button.



Note: you may notice that the buttons change from their color to a blue-ish tone that fades in and out while the button has focus. This is the Aero theme applying an animation to the focused button – you can't override this without replacing the control template (which we will do later in the class). If you'd like to stop that effect, check out the lab solution's App.xaml file – it merges in the Classic theme (Windows 2000 look and feel) which does not have this effect so the buttons stay the color you have set them!

Part 2 – Listening to child events

1. As you probably noticed, setting the `Click` event on each individual button is pretty tedious. In this part, you'll have the parent stack panel listen to the events coming from the children.
2. Remove the `Click` events from the button. Add a `Click` event to the `StackPanel`. Because the `Click` event is defined by `Button` instead of `StackPanel`, use the full name.

3. Compile and test.
4. Verify that you get an exception. This is because the sender is the stack panel rather than the button and we are attempting to cast it directly in the code!
5. Use the `Source` of the event rather than the `Sender`.

```
void OnButtonClick(object sender, RoutedEventArgs args)
 {
     Button buttonSender = (Button)args.Source;
     TextBox1.Background = buttonSender.Background;
 }
```

6. Compile and test. Verify that it works as expected. Try placing the event handler on the Window – does it still work there?

Solution

The after project contains an implementation of these two sections for your examination. The solution file is work\after\Input\InputLabAfter.sln

---

Part 3 – Using Commands

In the previous section, we linked the event directly to the handler. While this works just fine, it is not as flexible as it could be. In this part, you'll define a command will bind to each button and a handler. The resulting application will look identical to the above parts but have fewer ties between the code behind and XAML parts:

Steps:

1. Start with the project from Part 2, or use the solution file provided above.
2. Create a new static class called `MyCommands`.
3. Define a static field of type `RoutedUICommand` on the class called `SetBackground`.
    a. Create and assign a new `RoutedUICommand` to the static field either as a field initializer or in a static constructor.
    b. You will need to include the `System.Windows.Input` namespace for this.

```
public static class MyCommands
{
   public static RoutedUICommand SetBackground = new RoutedUICommand(
             "Set Background", "SetBackground", typeof(MyCommands));
 }
```

4. Remove the button click handler for the stack panel from your original solution.
5. Remember that the `Command` property is used on some controls to execute a specific command when the control is triggered. Our goal is to change the buttons to invoke a command instead of directly tying the logic to a handler.
6. For each of the buttons, set the `Command` property to invoke the command you just created.
    a. You may have to map an XML namespace for this. The below solution assumes you have mapped the namespace to the XAML namespace "me".

```
<StackPanel Orientation="Horizontal">
   <Button Padding="5" Foreground="Lime" Background="Green" Content="Green"
           Command="me:MyCommands.SetBackground" />
   <Button Padding="5" Foreground="DarkGoldenrod" Background="Yellow"
        Content="Yellow" Command="me:MyCommands.SetBackground" />
   <Button Padding="5" Foreground="DarkRed" Background ="Red" Content="Red"
           Command="me:MyCommands.SetBackground" />
</StackPanel>
```

Note: you can also use `{x:Static ns:Type.Command}` but it's not necessary here because the `Command` property has a type converter which assumes a static already. Feel free to fully type it out to try it if you like.

7. Compile and test your changes.
8. Verify that the buttons are disabled. This is because WPF does not know how to execute the command or even whether the command is executable.
9. Create a new `StackPanel.CommandBindings` section and add a new `CommandBinding` to it.
10. Set the `Command` property to the command you just defined.
11. Set the `Executed` and `CanExecute` attributes to new methods - `OnExecute` and `OnCanExecute`. Visual Studio will automatically create the code-behind handlers for you if you hit TAB, or right-click and select "Navigate to Event Handler".

```
<StackPanel.CommandBindings>
  <CommandBinding Command="me:MyCommands.SetBackground"
                  Executed="OnExecute" CanExecute="OnCanExecute" />
</StackPanel.CommandBindings>
```

12. In the `OnCanExecute` handler in the code behind, specify that the command is always executable by setting `e.CanExecute` to true.
    a. The method should conform to the `CanExecuteRoutedEventHandler` signature – Visual Studio should have created this for you, or you can type it in as follows:

```
void OnCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
   e.CanExecute = true;
}
```

13. Leave the `OnExecute` method empty for now by removing the exception that Visual Studio placed into it.
    a. The method should conform to the `ExecutedRoutedEventHandler` signature.
14. Compile and run. Verify that the buttons are now enabled.
15. When the command is executed, we want to specify the color to set. You'll use command parameters for this.
16. On each button, set the `CommandParameter` property to a `SolidColorBrush` with the appropriate color.
    a. **Note**: you can also use `{x:Static Brushes.xxx}`

```
<Button Padding="5" Foreground="Lime" Background="Green" Content="Green"
        Command="me:MyCommands.SetBackground"
        CommandParameter="{x:Static Brushes.Green}" />
<Button Padding="5" Foreground="DarkGoldenrod" Background="Yellow"
        Content="Yellow" Command="me:MyCommands.SetBackground">
   <Button.CommandParameter>
      <SolidColorBrush Color="Yellow" />
   </Button.CommandParameter>
</Button>
<Button Padding="5" Foreground="DarkRed" Background ="Red" Content="Red"
        Command="me:MyCommands.SetBackground"
        CommandParameter="{x:Static Brushes.Red}" />
```

17. Finally, implement `OnExecute` by getting the brush from the parameter and setting it as the background color of the text box.

```
void OnExecute(object sender, ExecutedRoutedEventArgs args)
{
     Brush b = (Brush)args.Parameter;
     TextBlock1.Background = b;
}
```

18. Compile and run. Verify that clicking the buttons sets the background color.

Solution

The **after** folder contains solutions for both sections which you can examine if you are having trouble. There are solutions supplied for both Visual Studio 2008 SP1 and Visual Studio 2010.