



Fundamentals

Estimated time for completion: 30 minutes

Overview:

All Silverlight developers need to be comfortable with dependency and attached properties, as they are fundamental to core features such as layout, data binding and animation. You can also use them for a number of other purposes, a common variant of which is to “inject” code into another type.

In this lab, you will create and use an attached property to do exactly this, injecting additional functionality into the standard TextBox control.

Goals:

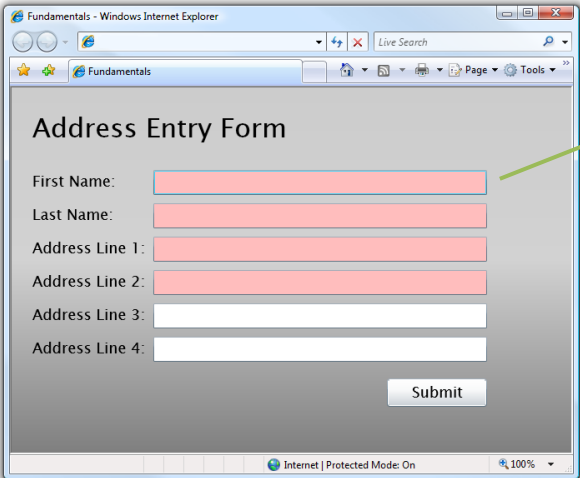
- Become comfortable with creating a dependency or attached property
- Use your own types within XAML

Introduction

Many applications highlight mandatory input fields in a different color (regardless of the fact that identifying items purely by color is generally a bad design choice). You will add this functionality to an application.

Introducing the application

1. The application that you will be enhancing is shown in the figure below.



Address Entry Form

First Name:

Last Name:

Address Line 1:

Address Line 2:

Address Line 3:

Address Line 4:

Submit

Mandatory fields

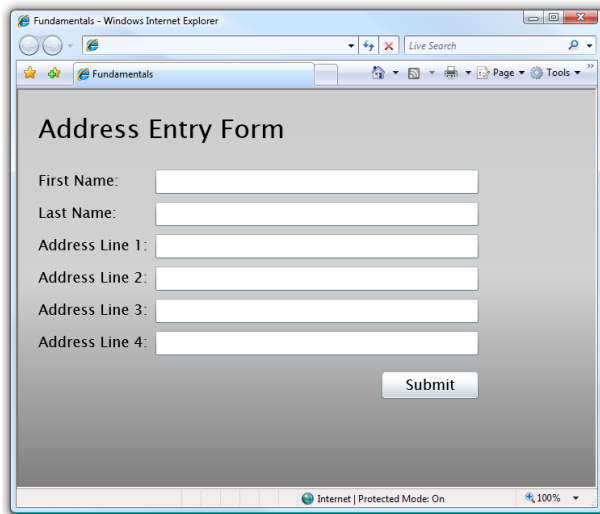
As you can see, it is just a simple name and address entry form, which consists of four mandatory entry fields and two optional fields. Note that the application is completely non-functional at this point: you will simply be working on adding the color highlighting effect shown above.

Enhancing the application

In this section of the lab, you will enhance the application so that it supports color highlighting of required fields that have yet to have their content set. You will do this using an attached property to inject code into the TextBox controls.

Opening the application

1. Using Visual Studio, open the Fundamentals solution from this [starter code folder](#).
2. Build and run the application. You will see the following application appear:



Note that none of the mandatory fields are highlighted in the delightful pink color requested by the users.

3. Close the application.

Creating the attached property

1. Add a new class to the Fundamentals application, naming it **TextBoxExtender**.
2. Add a new attached property to the **TextBoxExtender** class. The easiest way to do this is by using the **propa** snippet. Inside the **TextBoxExtender** class, type **propa** and then **press the tab key twice**.
3. Edit the inserted code so that the property's **type** is set to **bool**; its **name** is set to **IsRequired/IsRequiredProperty**; and the **ownerclass** is set to **TextBoxExtender**.

Your code should look like this:

```
public class TextBoxExtender
{
    public static bool GetIsRequired(DependencyObject obj)
    {
        return (bool)obj.GetValue(IsRequiredProperty);
    }

    public static void SetIsRequired(DependencyObject obj,
                                     bool value)
    {
        obj.SetValue(IsRequiredProperty, value);
    }

    public static readonly DependencyProperty IsRequiredProperty =
        DependencyProperty.RegisterAttached("IsRequired",
        typeof(bool),
        typeof(TextBoxExtender),
        new UIPropertyMetadata(0));
}
```

4. Unfortunately, the propa snippet generates code targeted at WPF, not Silverlight. Therefore, the last parameter to the **RegisterAttached** method is incorrect.

Replace the **new UIPropertyMetadata(0)** statement with the following code:

```
new PropertyMetadata( false, OnIsRequiredChanged )
```

Your call to the **RegisterAttached** method should now look like this:

```
public static readonly DependencyProperty IsRequiredProperty =
    DependencyProperty.RegisterAttached("IsRequired",
    typeof(bool),
    typeof(TextBoxExtender),
    new PropertyMetadata(false, OnIsRequiredChanged));
```

At this point, you might be wondering about two things: why is the property a Boolean, and what is `OnIsRequiredChanged`?

The reason you're using a Boolean for the property is simply that it makes sense for the behavior of the code that we want to inject: namely, is this field required or not.

The `OnIsRequiredChanged` is the magical mechanism that lets you inject code into another type. Specifically, it is the callback method that will be called whenever the value of the `IsRequiredProperty` is changed on a target `DependencyObject`.

5. Implement the **`OnIsRequiredChanged`** method in the `TextBoxExtender` class. The method should return `void` and take two parameters: a `DependencyObject` and a `DependencyPropertyChangedEventArgs`.

Your method should look like this:

```
private static void OnIsRequiredChanged(DependencyObject dobj,
                                         DependencyPropertyChangedEventArgs e)
{
}
```

6. This method is called whenever the value of `IsRequired` is changed on the target `DependencyObject`. You can read the value of the dependency property from the **`NewValue`** property of the `e` parameter.

Write code now inside this method to check if the `DependencyObject` passed in is a `TextBox`. If it is, and the value of `NewValue` is **`true`**, subscribe to the `TextBox`'s **`TextChanged`** event (name the handler method **`TargetTextChanged`**); if the value is **`false`**, then unsubscribe.

Flip the page to look at how this code might look.

```

private static void OnIsRequiredChanged(DependencyObject dobj,
                                       DependencyPropertyChangedEventArgs e)
{
    TextBox target = dobj as TextBox;
    if (target == null) return;

    bool connecting = (bool)e.NewValue;

    if (connecting)
    {
        target.TextChanged += TargetTextChanged;
    }
    else
    {
        target.TextChanged -= TargetTextChanged;
    }
}

private static void TargetTextChanged(object sender,
                                     TextChangedEventArgs e)
{
}

```

7. By now, you should be able to see how the code injection works. When you attach the property onto a `TextBox`, the `OnIsRequiredChanged` callback will fire, and you will subscribe to the `TextChanged` event.

Therefore, now add code to the `TargetTextChanged` method to set the **Background** property of the `TextBox` to an appropriately colored brush. The best way to do this is to declare two static fields in the class, as shown below, and simply toggle between them based on the length of text in the `TextBox`.

```

private static void TargetTextChanged(object sender,
                                     TextChangedEventArgs e)
{
    TextBox target = sender as TextBox;
    target.Background = (target.Text.Length == 0) ?
                        errorBrush : normalBrush;
}

private static SolidColorBrush normalBrush =
    new SolidColorBrush(Colors.White);
private static SolidColorBrush errorBrush =
    new SolidColorBrush(Color.FromArgb(255, 255, 189, 189));

```

8. The final refinement is to make sure that `TargetTextChanged` is called from your `OnIsRequiredChanged` method, just to make sure that the `TextBox`'s `Background` is set to the correct color initially, as shown below:

```
private static void OnIsRequiredChanged(DependencyObject dobj,
                                       DependencyPropertyChangedEventArgs e)
{
    TextBox target = dobj as TextBox;
    if (target == null) return;

    bool connecting = (bool)e.NewValue;

    if (connecting)
    {
        target.TextChanged += TargetTextChanged;
    }
    else
    {
        target.TextChanged -= TargetTextChanged;
    }

    TargetTextChanged(target, null);
}
```

9. You have completed the coding of the `TextBoxExtender`, so build your application and fix any compiler errors that might have crept in.

Applying the attached property

1. You will now apply the attached property to the four TextBoxes on the page, so open up MainPage.xaml.
2. Before you can apply the property, you must bring the `TextBoxExtender` into scope. To do this, add a new `xmlns` attribute to the `UserControl` element, setting the prefix to `my:` and the namespace to `clr-namespace:Fundamentals`.

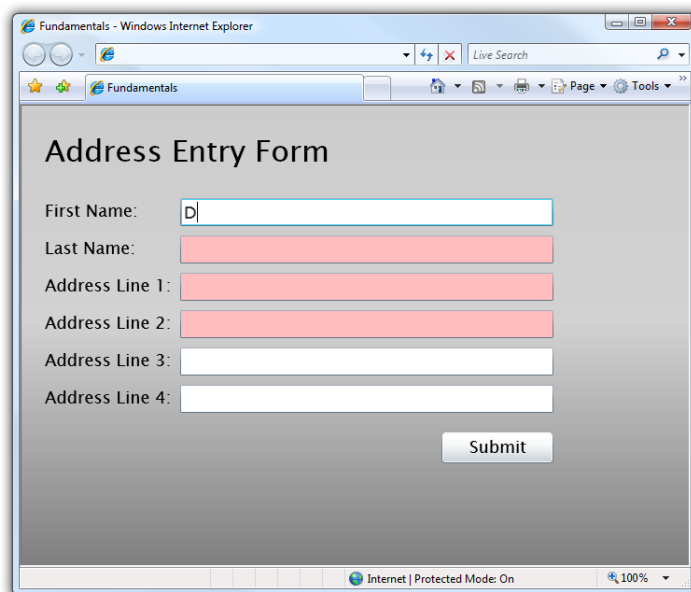
Your code should look like this:

```
<UserControl x:Class="Fundamentals.MainPage"
    ... other namespaces ...
    xmlns:my="clr-namespace:Fundamentals">
```

3. Now that your `TextBoxExtender` is in scope, add the `IsRequired` attached property to the four `TextBox` elements `txtFirstName`, `txtLastName`, `txtLine1` and `txtLine2`. You need to set the value for each to `true`. An example is shown below:

```
<TextBox my:TextBoxExtender.IsRequired="true" ... />
```

4. Build and run the application. The top four `TextBox` elements should be an attractive pink color. Start typing your first name into the top `TextBox`. Its background should instantly go white as soon as the first letter appears (and go pink again if you delete the content), as shown below:



5. Play with the application until you're happy how it works.

6. There are a few things to think about with the code that you've written. For example, do you think that it's a good idea that the color is hardcoded inside the extender? Should you embed validation logic in the UI or the underlying business component?

Feel free to discuss these points and anything else about this technique with your instructor.

7. When you're finished playing, close down the application.

Review

In this lab, you injected code into some `TextBox` controls by using an attached property. The property's change callback mechanism provided you with an opportunity to connect, or disconnect, event handlers. This simplified the process of handling those events without having to do large amounts of work. It also enabled you to encapsulate the functionality into a re-usable class, without forcing you to inherit from `TextBox`.

This common technique is widely used in both Silverlight and WPF, and is one of the many benefits of the dependency /attached property mechanism. However, it should be noted that the new Behavior functionality offered within Expression Blend, which builds on the concept that you've worked on here, makes this even easier for developers and designers to work together.

Solutions

[Fundamentals](#)