Walkthrough: Hosting a WPF Composite Control in Windows Forms

.NET Framework 4.5

Windows Presentation Foundation (WPF) provides a rich environment for creating applications. However, when you have a substantial investment in Windows Forms code, it can be more effective to extend your existing Windows Forms application with WPF rather than to rewrite it from scratch. A common scenario is when you want to embed one or more controls implemented with WPF within your Windows Forms application. For more information about customizing WPF controls, see Control Customization.

This walkthrough steps you through an application that hosts a WPF composite control to perform data-entry in a Windows Forms application. The composite control is packaged in a DLL. This general procedure can be extended to more complex applications and controls. This walkthrough is designed to be nearly identical in appearance and functionality to Walkthrough: Hosting a Windows Forms Composite Control in WPF. The primary difference is that the hosting scenario is reversed.

The walkthrough is divided into two sections. The first section briefly describes the implementation of the WPF composite control. The second section discusses in detail how to host the composite control in a Windows Forms application, receive events from the control, and access some of the control's properties.

Tasks illustrated in this walkthrough include:

- Implementing the WPF composite control.
- Implementing the Windows Forms host application.

For a complete code listing of the tasks illustrated in this walkthrough, see Hosting a WPF Composite Control in Windows Forms Sample.

Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2010.

Implementing the WPF Composite Control

The WPF composite control used in this example is a simple data-entry form that takes the user's name and

address. When the user clicks one of two buttons to indicate that the task is finished, the control raises a custom event to return that information to the host. The following illustration shows the rendered control.

WPF composite control

Creating the Project

To start the project:

1. Launch Microsoft Visual Studio, and open the **New Project** dialog box.
2. In Visual C# and the Windows category, select the **WPF User Control Library** template.
3. Name the new project **MyControls**.
4. For the location, specify a conveniently named top-level folder, such as **WindowsFormsHostingWpfControl**. Later, you will put the host application in this folder.
5. Click **OK** to create the project. The default project contains a single control named `UserControl1`.
6. In Solution Explorer, rename `UserControl1` to `MyControl1`.

Your project should have references to the following system DLLs. If any of these DLLs are not included by default, add them to your project.

- PresentationCore
- PresentationFramework
- System
- WindowsBase

Creating the User Interface

The user interface (UI) for the composite control is implemented with Extensible Application Markup

Language (XAML). The composite control UI consists of five TextBox elements. Each TextBox element has an associated TextBlock element that serves as a label. There are two Button elements at the bottom, **OK** and **Cancel**. When the user clicks either button, the control raises a custom event to return the information to the host.

Basic Layout

The various UI elements are contained in a Grid element. You can use Grid to arrange the contents of the composite control in much the same way you would use a **Table** element in HTML. WPF also has a Table element, but Grid is more lightweight and better suited for simple layout tasks.

The following XAML shows the basic layout. This XAML defines the overall structure of the control by specifying the number of columns and rows in the Grid element.

In MyControl1.xaml, replace the existing XAML with the following XAML.

XAML

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="MyControls.MyControl1"
      Background="#DCDCDC"
      Width="375"
      Height="250"
      Name="rootElement"
      Loaded="Init">

  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>


...


</Grid>
```

Adding TextBlock and TextBox Elements to the Grid

You place a UI element in the grid by setting the element's RowProperty and ColumnProperty attributes

You place a UI element in the grid by setting the element's RowProperty and ColumnProperty attributes to the appropriate row and column number. Remember that row and column numbering are zero-based. You can have an element span multiple columns by setting its ColumnSpanProperty attribute. For more information about Grid elements, see How to: Create a Grid Element.

The following XAML shows the composite control's TextBox and TextBlock elements with their RowProperty and ColumnProperty attributes, which are set to place the elements properly in the grid.

In MyControl1.xaml, add the following XAML within the Grid element.

XAML

```
<TextBlock Grid.Column="0"
      Grid.Row="0"
      Grid.ColumnSpan="4"
      Margin="10,5,10,0"
      HorizontalAlignment="Center"
      Style="{StaticResource titleText}">Simple WPF Control</TextBlock>

<TextBlock Grid.Column="0"
      Grid.Row="1"
      Style="{StaticResource inlineText}"
      Name="nameLabel">Name</TextBlock>
<TextBox Grid.Column="1"
      Grid.Row="1"
      Grid.ColumnSpan="3"
      Name="txtName"/>

<TextBlock Grid.Column="0"
      Grid.Row="2"
      Style="{StaticResource inlineText}"
      Name="addressLabel">Street Address</TextBlock>
<TextBox Grid.Column="1"
      Grid.Row="2"
      Grid.ColumnSpan="3"
      Name="txtAddress"/>

<TextBlock Grid.Column="0"
      Grid.Row="3"
      Style="{StaticResource inlineText}"
      Name="cityLabel">City</TextBlock>
<TextBox Grid.Column="1"
      Grid.Row="3"
      Width="100"
      Name="txtCity"/>

<TextBlock Grid.Column="2"
      Grid.Row="3"
      Style="{StaticResource inlineText}"
      Name="stateLabel">State</TextBlock>
<TextBox Grid.Column="3"
      Grid.Row="3"
      Width="50"
      Name="txtState"/>

<TextBlock Grid.Column="0"
      Grid.Row="4"
```

```
        Style="{StaticResource inlineText}"
        Name="zipLabel">Zip</TextBlock>
  <TextBox Grid.Column="1"
        Grid.Row="4"
        Width="100"
        Name="txtZip"/>
```

Styling the UI Elements

Many of the elements on the data-entry form have a similar appearance, which means that they have identical settings for several of their properties. Rather than setting each element's attributes separately, the previous XAML uses Style elements to define standard property settings for classes of elements. This approach reduces the complexity of the control and enables you to change the appearance of multiple elements through a single style attribute.

The Style elements are contained in the Grid element's Resources property, so they can be used by all elements in the control. If a style is named, you apply it to an element by adding a Style element set to the style's name. Styles that are not named become the default style for the element. For more information about WPF styles, see Styling and Templating.

The following XAML shows the Style elements for the composite control. To see how the styles are applied to elements, see the previous XAML. For example, the last TextBlock element has the `inlineText` style, and the last TextBox element uses the default style.

In MyControl1.xaml, add the following XAML just after the Grid start element.

XAML

```
<Grid.Resources>
  <Style x:Key="inlineText" TargetType="{x:Type TextBlock}">
    <Setter Property="Margin" Value="10,5,10,0"/>
    <Setter Property="FontWeight" Value="Normal"/>
    <Setter Property="FontSize" Value="12"/>
  </Style>
  <Style x:Key="titleText" TargetType="{x:Type TextBlock}">
    <Setter Property="DockPanel.Dock" Value="Top"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="Margin" Value="10,5,10,0"/>
  </Style>
  <Style TargetType="{x:Type Button}">
    <Setter Property="Margin" Value="10,5,10,0"/>
    <Setter Property="Width" Value="60"/>
  </Style>
  <Style TargetType="{x:Type TextBox}">
    <Setter Property="Margin" Value="10,5,10,0"/>
  </Style>
</Grid.Resources>
```

Adding the OK and Cancel Buttons

Adding the OK and Cancel Buttons

The final elements on the composite control are the **OK** and **Cancel**Button elements, which occupy the first two columns of the last row of the Grid. These elements use a common event handler, `ButtonClicked`, and the default Button style defined in the previous XAML.

In MyControl1.xaml, add the following XAML after the last TextBox element. The XAML part of the composite control is now complete.

XAML

```
<Button Grid.Row="5"
        Grid.Column="0"
        Name="btnOK"
        Click="ButtonClicked">OK</Button>
<Button Grid.Row="5"
        Grid.Column="1"
        Name="btnCancel"
        Click="ButtonClicked">Cancel</Button>
```

Implementing the Code-Behind File

The code-behind file, MyControl1.xaml.cs, implements three essential tasks:

1. Handles the event that occurs when the user clicks one of the buttons.
2. Retrieves the data from the TextBox elements, and packages it in a custom event argument object.
3. Raises the custom `OnButtonClick` event, which notifies the host that the user is finished and passes the data back to the host.

The control also exposes a number of color and font properties that enable you to change the appearance. Unlike the WindowsFormsHost class, which is used to host a Windows Forms control, the ElementHost class exposes the control's Background property only. To maintain the similarity between this code example and the example discussed in Walkthrough: Hosting a Windows Forms Composite Control in WPF, the control exposes the remaining properties directly.

The Basic Structure of the Code-Behind File

The code-behind file consists of a single namespace, `MyControls`, which will contain two classes, `MyControl1` and `MyControlEventArgs`.

```
namespace MyControls
{
  public partial class MyControl1 : Grid
  {
```

```
    //...
  }
  public class MyControlEventArgs : EventArgs
  {
    //...
  }
}
```

The first class, `MyControl1`, is a partial class containing the code that implements the functionality of the UI defined in MyControl1.xaml. When MyControl1.xaml is parsed, the XAML is converted to the same partial class, and the two partial classes are merged to form the compiled control. For this reason, the class name in the code-behind file must match the class name assigned to MyControl1.xaml, and it must inherit from the root element of the control. The second class, `MyControlEventArgs`, is an event arguments class that is used to send the data back to the host.

Open MyControl1.xaml.cs. Change the existing class declaration so that it has the following name and inherits from Grid.

C#

```
public partial class MyControl1 : Grid
```

Initializing the Control

The following code implements several basic tasks:

- Declares a private event, `OnButtonClick`, and its associated delegate, `MyControlEventHandler`.
- Creates several private global variables that store the user's data. This data is exposed through corresponding properties.
- Implements a handler, `Init`, for the control's Loaded event. This handler initializes the global variables by assigning them the values defined in MyControl1.xaml. To do this, it uses the Name assigned to a typical TextBlock element, `nameLabel`, to access that element's property settings.

Delete the existing constructor and add the following code to your `MyControl1` class.

C#

```
public delegate void MyControlEventHandler(object sender, MyControlEventArgs ar
gs);
public event MyControlEventHandler OnButtonClick;
private FontWeight _fontWeight;
private double _fontSize;
private FontFamily _fontFamily;
private FontStyle _fontStyle;
private SolidColorBrush _foreground;
private SolidColorBrush _background;
```

```
private void Init(object sender, EventArgs e)
{
    //They all have the same style, so use nameLabel to set initial values.
    _fontWeight = nameLabel.FontWeight;
    _fontSize = nameLabel.FontSize;
    _fontFamily = nameLabel.FontFamily;
    _fontStyle = nameLabel.FontStyle;
    _foreground = (SolidColorBrush)nameLabel.Foreground;
    _background = (SolidColorBrush)rootElement.Background;
}
```

Handling the Buttons' Click Events

The user indicates that the data-entry task is finished by clicking either the **OK** button or the **Cancel** button. Both buttons use the same Click event handler, `ButtonClicked`. Both buttons have a name, `btnOK` or `btnCancel`, that enables the handler to determine which button was clicked by examining the value of the *sender* argument. The handler does the following:

- Creates a `MyControlEventArgs` object that contains the data from the TextBox elements.
- If the user clicked the **Cancel** button, sets the `MyControlEventArgs` object's `IsOK` property to **false**.
- Raises the `OnButtonClick` event to indicate to the host that the user is finished, and passes back the collected data.

Add the following code to your `MyControl1` class, after the `Init` method.

C#

```
private void ButtonClicked(object sender, RoutedEventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(true,
                                                        txtName.Text,
                                                        txtAddress.Text,
                                                        txtCity.Text,
                                                        txtState.Text,
                                                        txtZip.Text);
    if (sender == btnCancel)
    {
        retvals.IsOK = false;
    }
    if (OnButtonClick != null)
        OnButtonClick(this, retvals);
}
```

Creating Properties

The remainder of the class simply exposes properties that correspond to the global variables discussed previously. When a property changes, the set accessor modifies the appearance of the control by changing the corresponding element properties and updating the underlying global variables.

Add the following code to your `MyControl1` class.

C#

```
public FontWeight MyControl_FontWeight
{
    get { return _fontWeight; }
    set
    {
        _fontWeight = value;
        nameLabel.FontWeight = value;
        addressLabel.FontWeight = value;
        cityLabel.FontWeight = value;
        stateLabel.FontWeight = value;
        zipLabel.FontWeight = value;
    }
}
public double MyControl_FontSize
{
    get { return _fontSize; }
    set
    {
        _fontSize = value;
        nameLabel.FontSize = value;
        addressLabel.FontSize = value;
        cityLabel.FontSize = value;
        stateLabel.FontSize = value;
        zipLabel.FontSize = value;
    }
}
public FontStyle MyControl_FontStyle
{
    get { return _fontStyle; }
    set
    {
        _fontStyle = value;
        nameLabel.FontStyle = value;
        addressLabel.FontStyle = value;
        cityLabel.FontStyle = value;
        stateLabel.FontStyle = value;
        zipLabel.FontStyle = value;
    }
}
public FontFamily MyControl_FontFamily
{
    get { return _fontFamily; }
    set
    {
        _fontFamily = value;
        nameLabel.FontFamily = value;
        addressLabel.FontFamily = value;
        cityLabel.FontFamily = value;
```

```
            stateLabel.FontFamily = value;
            zipLabel.FontFamily = value;
        }
    }

    public SolidColorBrush MyControl_Background
    {
        get { return _background; }
        set
        {
            _background = value;
            rootElement.Background = value;
        }
    }
    public SolidColorBrush MyControl_Foreground
    {
        get { return _foreground; }
        set
        {
            _foreground = value;
            nameLabel.Foreground = value;
            addressLabel.Foreground = value;
            cityLabel.Foreground = value;
            stateLabel.Foreground = value;
            zipLabel.Foreground = value;
        }
    }
```

Sending the Data Back to the Host

The final component in the file is the `MyControlEventArgs` class, which is used to send the collected data back to the host.

Add the following code to your `MyControls` namespace. The implementation is straightforward, and is not discussed further.

C#

```
    public class MyControlEventArgs : EventArgs
    {
        private string _Name;
        private string _StreetAddress;
        private string _City;
        private string _State;
        private string _Zip;
        private bool _IsOK;

        public MyControlEventArgs(bool result,
                                  string name,
                                  string address,
                                  string city,
                                  string state,
                                  string zip)
        {
```

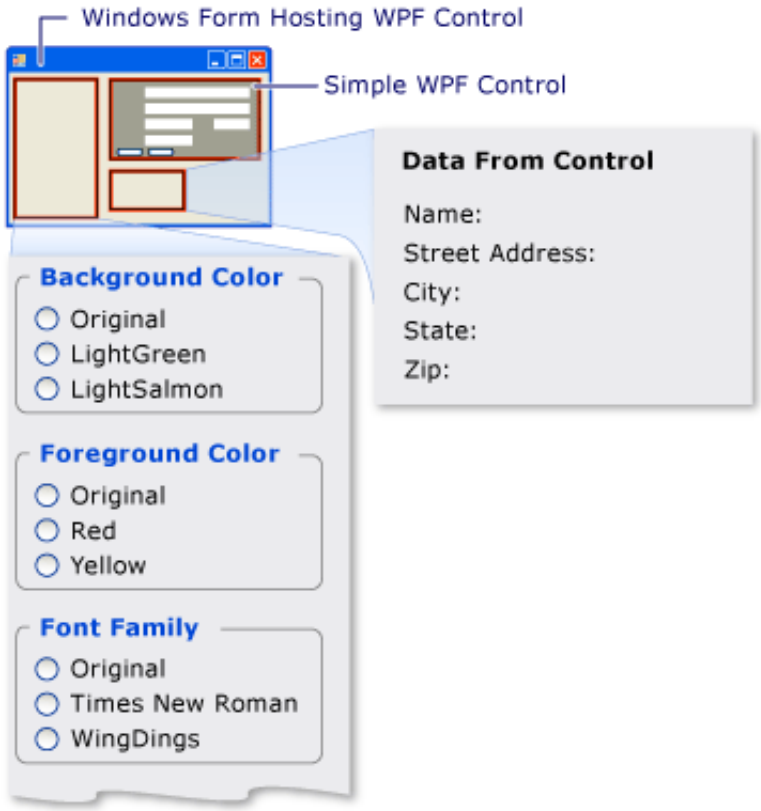
```
        {
            _IsOK = result;
            _Name = name;
            _StreetAddress = address;
            _City = city;
            _State = state;
            _Zip = zip;
        }

        public string MyName
        {
            get { return _Name; }
            set { _Name = value; }
        }
        public string MyStreetAddress
        {
            get { return _StreetAddress; }
            set { _StreetAddress = value; }
        }
        public string MyCity
        {
            get { return _City; }
            set { _City = value; }
        }
        public string MyState
        {
            get { return _State; }
            set { _State = value; }
        }
        public string MyZip
        {
            get { return _Zip; }
            set { _Zip = value; }
        }
        public bool IsOK
        {
            get { return _IsOK; }
            set { _IsOK = value; }
        }
    }
```

Build the solution. The build will produce a DLL named MyControls.dll.

Implementing the Windows Forms Host Application

The Windows Forms host application uses an ElementHost object to host the WPF composite control. The application handles the `OnButtonClick` event to receive the data from the composite control. The application also has a set of option buttons that you can use to modify the control's appearance. The following illustration shows the application.

WPF composite control hosted in a Windows Forms application



Creating the Project

To start the project:

1. Launch Visual Studio, and open the **New Project** dialog box.
2. In Visual C# and the Windows category, select the **Windows Forms Application** template.
3. Name the new project **WFHost**.
4. For the location, specify the same top-level folder that contains the MyControls project.
5. Click **OK** to create the project.

You also need to add references to the DLL that contains `MyControl1` and other assemblies.

1. Right-click the project name in Solution Explorer, and select **Add Reference**.
2. Click the **Browse** tab, and browse to the folder that contains MyControls.dll. For this walkthrough, this folder is MyControls\bin\Debug.
3. Select MyControls.dll, and then click **OK**.
4. Add references to the following assemblies.
    - PresentationCore
    - PresentationFramework

- System.Xaml
- WindowsBase
- WindowsFormsIntegration

Implementing the User Interface for the Application

The UI for the Windows Form application contains several controls to interact with the WPF composite control.

1. Open Form1 in the Windows Form Designer.
2. Enlarge the form to accommodate the controls.
3. In the upper-right corner of the form, add a System.Windows.Forms.Panel control to hold the WPF composite control.
4. Add the following System.Windows.Forms.GroupBox controls to the form.

Name	Text
groupBox1	Background Color
groupBox2	Foreground Color
groupBox3	Font Size
groupBox4	Font Family
groupBox5	Font Style
groupBox6	Font Weight
groupBox7	Data from control

5. Add the following System.Windows.Forms.RadioButton controls to the System.Windows.Forms.GroupBox controls.

GroupBox	Name	Text
groupBox1	radioBackgroundOriginal	Original
groupBox1	radioBackgroundLightGreen	LightGreen
groupBox1	radioBackgroundLightSalmon	LightSalmon

groupBox2	radioForegroundOriginal	Original
groupBox2	radioForegroundRed	Red
groupBox2	radioForegroundYellow	Yellow
groupBox3	radioSizeOriginal	Original
groupBox3	radioSizeTen	10
groupBox3	radioSizeTwelve	12
groupBox4	radioFamilyOriginal	Original
groupBox4	radioFamilyTimes	Times New Roman
groupBox4	radioFamilyWingDings	WingDings
groupBox5	radioStyleOriginal	Normal
groupBox5	radioStyleItalic	Italic
groupBox6	radioWeightOriginal	Original
groupBox6	radioWeightBold	Bold

6. Add the following System.Windows.Forms.Label controls to the last System.Windows.Forms.GroupBox. These controls display the data returned by the WPF composite control.

GroupBox	**Name**	**Text**
groupBox7	lblName	Name:
groupBox7	lblAddress	Street Address:
groupBox7	lblCity	City:
groupBox7	lblState	State:
groupBox7	lblZip	Zip:

Initializing the Form

You generally implement the hosting code in the form's Load event handler. The following code shows the

Load event handler, a handler for the WPF composite control's Loaded event, and declarations for several global variables that are used later.

In the Windows Forms Designer, double-click the form to create a Load event handler. At the top of Form1.cs, add the following **using** statements.

C#

```
using System;
using System.Windows;
using System.Windows.Navigation;
using System.Windows.Controls;
using System.Windows.Media;

namespace MyControls
{
    public partial class MyControl1 : Grid
    {
        public delegate void MyControlEventHandler(object sender, MyControlEventA
rgs args);
        public event MyControlEventHandler OnButtonClick;
        private FontWeight _fontWeight;
        private double _fontSize;
        private FontFamily _fontFamily;
        private FontStyle _fontStyle;
        private SolidColorBrush _foreground;
        private SolidColorBrush _background;

        private void Init(object sender, EventArgs e)
        {
            //They all have the same style, so use nameLabel to set initial value
s.
            _fontWeight = nameLabel.FontWeight;
            _fontSize = nameLabel.FontSize;
            _fontFamily = nameLabel.FontFamily;
            _fontStyle = nameLabel.FontStyle;
            _foreground = (SolidColorBrush)nameLabel.Foreground;
            _background = (SolidColorBrush)rootElement.Background;
        }

        private void ButtonClicked(object sender, RoutedEventArgs e)
        {
            MyControlEventArgs retvals = new MyControlEventArgs(true,
                                                                txtName.Text,
                                                                txtAddress.Text,
                                                                txtCity.Text,
                                                                txtState.Text,
                                                                txtZip.Text);
            if (sender == btnCancel)
            {
                retvals.IsOK = false;
            }
            if (OnButtonClick != null)
                OnButtonClick(this, retvals);
        }

        public FontWeight MyControl_FontWeight
```

```
        {
            get { return _fontWeight; }
            set
            {
                _fontWeight = value;
                nameLabel.FontWeight = value;
                addressLabel.FontWeight = value;
                cityLabel.FontWeight = value;
                stateLabel.FontWeight = value;
                zipLabel.FontWeight = value;
            }
        }
        public double MyControl_FontSize
        {
            get { return _fontSize; }
            set
            {
                _fontSize = value;
                nameLabel.FontSize = value;
                addressLabel.FontSize = value;
                cityLabel.FontSize = value;
                stateLabel.FontSize = value;
                zipLabel.FontSize = value;
            }
        }
        public FontStyle MyControl_FontStyle
        {
            get { return _fontStyle; }
            set
            {
                _fontStyle = value;
                nameLabel.FontStyle = value;
                addressLabel.FontStyle = value;
                cityLabel.FontStyle = value;
                stateLabel.FontStyle = value;
                zipLabel.FontStyle = value;
            }
        }
        public FontFamily MyControl_FontFamily
        {
            get { return _fontFamily; }
            set
            {
                _fontFamily = value;
                nameLabel.FontFamily = value;
                addressLabel.FontFamily = value;
                cityLabel.FontFamily = value;
                stateLabel.FontFamily = value;
                zipLabel.FontFamily = value;
            }
        }

        public SolidColorBrush MyControl_Background
        {
            get { return _background; }
            set
            {
                _background = value;
```

```
                _background = value;
                rootElement.Background = value;
            }
        }
        public SolidColorBrush MyControl_Foreground
        {
            get { return _foreground; }
            set
            {
                _foreground = value;
                nameLabel.Foreground = value;
                addressLabel.Foreground = value;
                cityLabel.Foreground = value;
                stateLabel.Foreground = value;
                zipLabel.Foreground = value;
            }
        }
    }

    public class MyControlEventArgs : EventArgs
    {
        private string _Name;
        private string _StreetAddress;
        private string _City;
        private string _State;
        private string _Zip;
        private bool _IsOK;

        public MyControlEventArgs(bool result,
                                  string name,
                                  string address,
                                  string city,
                                  string state,
                                  string zip)
        {
            _IsOK = result;
            _Name = name;
            _StreetAddress = address;
            _City = city;
            _State = state;
            _Zip = zip;
        }

        public string MyName
        {
            get { return _Name; }
            set { _Name = value; }
        }
        public string MyStreetAddress
        {
            get { return _StreetAddress; }
            set { _StreetAddress = value; }
        }
        public string MyCity
        {
            get { return _City; }
            set { _City = value; }
        }
```

```
        public string MyState
        {
            get { return _State; }
            set { _State = value; }
        }
        public string MyZip
        {
            get { return _Zip; }
            set { _Zip = value; }
        }
        public bool IsOK
        {
            get { return _IsOK; }
            set { _IsOK = value; }
        }
    }
}
```

C#

```
using System.Windows;
using System.Windows.Forms.Integration;
using System.Windows.Media;
```

Replace the contents of the existing `Form1` class with the following code.

C#

```
private ElementHost ctrlHost;
private MyControls.MyControl1 wpfAddressCtrl;
System.Windows.FontWeight initFontWeight;
double initFontSize;
System.Windows.FontStyle initFontStyle;
System.Windows.Media.SolidColorBrush initBackBrush;
System.Windows.Media.SolidColorBrush initForeBrush;
System.Windows.Media.FontFamily initFontFamily;

public Form1()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
    ctrlHost = new ElementHost();
    ctrlHost.Dock = DockStyle.Fill;
    panel1.Controls.Add(ctrlHost);
    wpfAddressCtrl = new MyControls.MyControl1();
    wpfAddressCtrl.InitializeComponent();
    ctrlHost.Child = wpfAddressCtrl;

    wpfAddressCtrl.OnButtonClick +=
        new MyControls.MyControl1.MyControlEventHandler(
        avAddressCtrl_OnButtonClick);
    wpfAddressCtrl.Loaded += new RoutedEventHandler(
```

```
            avAddressCtrl_Loaded);
    }

    void avAddressCtrl_Loaded(object sender, EventArgs e)
    {
        initBackBrush = (SolidColorBrush)wpfAddressCtrl.MyControl_Background;
        initForeBrush = wpfAddressCtrl.MyControl_Foreground;
        initFontFamily = wpfAddressCtrl.MyControl_FontFamily;
        initFontSize = wpfAddressCtrl.MyControl_FontSize;
        initFontWeight = wpfAddressCtrl.MyControl_FontWeight;
        initFontStyle = wpfAddressCtrl.MyControl_FontStyle;
    }
```

The `Form1_Load` method in the preceding code shows the general procedure for hosting a WPF control:

1. Create a new ElementHost object.
2. Set the control's Dock property to DockStyle.Fill.
3. Add the ElementHost control to the Panel control's Controls collection.
4. Create an instance of the WPF control.
5. Host the composite control on the form by assigning the control to the ElementHost control's Child property.

The remaining two lines in the `Form1_Load` method attach handlers to two control events:

- `OnButtonClick` is a custom event that is fired by the composite control when the user clicks the **OK** or **Cancel** button. You handle the event to get the user's response and to collect any data that the user specified.
- Loaded is a standard event that is raised by a WPF control when it is fully loaded. The event is used here because the example needs to initialize several global variables using properties from the control. At the time of the form's Load event, the control is not fully loaded and those values are still set to **null**. You need to wait until the control's Loaded event occurs before you can access those properties.

The Loaded event handler is shown in the preceding code. The `OnButtonClick` handler is discussed in the next section.

Handling OnButtonClick

The `OnButtonClick` event occurs when the user clicks the **OK** or **Cancel** button.

The event handler checks the event argument's `IsOK` field to determine which button was clicked. The `lbl`*data* variables correspond to the Label controls that were discussed earlier. If the user clicks the **OK** button, the data from the control's TextBox controls is assigned to the corresponding Label control. If the user clicks **Cancel**, the Text values are set to the default strings.

user clicks **Cancel**, the Text values are set to the default strings.

Add the following button click event handler code to the `Form1` class.

C#

```
void avAddressCtrl_OnButtonClick(
    object sender,
    MyControls.MyControl1.MyControlEventArgs args)
{
    if (args.IsOK)
    {
        lblAddress.Text = "Street Address: " + args.MyStreetAddress;
        lblCity.Text = "City: " + args.MyCity;
        lblName.Text = "Name: " + args.MyName;
        lblState.Text = "State: " + args.MyState;
        lblZip.Text = "Zip: " + args.MyZip;
    }
    else
    {
        lblAddress.Text = "Street Address: ";
        lblCity.Text = "City: ";
        lblName.Text = "Name: ";
        lblState.Text = "State: ";
        lblZip.Text = "Zip: ";
    }
}
```

Build and run the application. Add some text in the WPF composite control and then click **OK**. The text appears in the labels. At this point, code has not been added to handle the radio buttons.

Modifying the Appearance of the Control

The RadioButton controls on the form will enable the user to change the WPF composite control's foreground and background colors as well as several font properties. The background color is exposed by the ElementHost object. The remaining properties are exposed as custom properties of the control.

Double-click each RadioButton control on the form to create CheckedChanged event handlers. Replace the CheckedChanged event handlers with the following code.

C#

```
private void radioBackgroundOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = initBackBrush;
}

private void radioBackgroundLightGreen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = new SolidColorBrush(Colors.LightGreen);
}

private void radioBackgroundLightSalmon_CheckedChanged(object sender, EventArgs e
```

```
)
{
    wpfAddressCtrl.MyControl_Background = new SolidColorBrush(Colors.LightSalmon)
;
}

private void radioForegroundOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = initForeBrush;
}

private void radioForegroundRed_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new System.Windows.Media.SolidColorBrus
h(Colors.Red);
}

private void radioForegroundYellow_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new System.Windows.Media.SolidColorBrus
h(Colors.Yellow);
}

private void radioFamilyOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = initFontFamily;
}

private void radioFamilyTimes_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new System.Windows.Media.FontFamily("Ti
mes New Roman");
}

private void radioFamilyWingDings_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new System.Windows.Media.FontFamily("Wi
ngDings");
}

private void radioSizeOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = initFontSize;
}

private void radioSizeTen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 10;
}

private void radioSizeTwelve_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 12;
}

private void radioStyleOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = initFontStyle;
```

```
}

private void radioStyleItalic_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = System.Windows.FontStyles.Italic;
}

private void radioWeightOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontWeight = initFontWeight;
}

private void radioWeightBold_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontWeight = FontWeights.Bold;
}
```

Build and run the application. Click the different radio buttons to see the effect on the WPF composite control.

See Also

Tasks

Walkthrough: Hosting a 3-D WPF Composite Control in Windows Forms

Reference

ElementHost
WindowsFormsHost

Concepts

Walkthrough: Hosting a Windows Forms Composite Control in WPF

Other Resources

WPF Designer