WPF Recipes in C# 2008: A Problem-Solution Approach

ISBN-13 (pbk): 978-1-4302-1084-9

ISBN-13 (electronic): 978-1-4302-1083-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Todd Meister
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Sofia Marchant
Copy Editor: Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens and Octal Publishing, Inc.
Proofreader: April Eddy and Kim Burton
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Building and Debugging WPF Applications

WPF provides a great deal of powerful functionality that you can leverage to simplify and speed up the development and debugging processes of your applications. This includes functionality that would have required a great deal of effort in WinForms. From sharing resources across your application to creating custom properties that you can use in animations and bindings to narrowing down the debugging process of data bindings, there's something for everyone.

This chapter focuses on the basics of building a rich WPF application and some methods that you can use to help ease the debugging of data bindings. The recipes in this chapter describe how to:

- Create a standard WPF application (recipe 1-1)
- Handle an unhandled exception (recipe 1-2)
- Create and use dependency properties (recipes 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, and 1-9)
- Handle resources in an application (recipes 1-10, 1-11, and 1-12)
- Share properties throughout an application (recipe 1-13)
- Create a single-instance application (recipe 1-14)
- Manage multiple windows in an application (recipe 1-15)
- Debug data bindings (recipes 1-16 and 1-17)

1-1. Create a Standard WPF Application

Problem

You need to create a new, rich WPF desktop application.

Solution

Create a new project with a single `App.xaml` file, containing the main entry point for your application.

How It Works

In its simplest form, an application is defined by creating a `System.Windows.Application` object. When creating a new Windows Application project in Visual Studio, you are given the default definition of the `Application` object. The `Application` object provides useful functionality such as the following:

- A last chance to handle an unhandled exception
- Handling application-wide resources and properties
- Providing access to the windows contained in the application

The application definition needs a special `MSBuild` property to indicate that it contains the application's definition. This can be set using the Properties window of Microsoft Visual Studio, specifically, by setting the value of `Build Action` to `ApplicationDefinition`. If you attempt to compile a Windows Application project that doesn't have a file marked with a build action of `ApplicationDefinition`, you will receive an error stating that no main entry point was found in the application. One of the side effects of the `ApplicationDefinition` build action adds a definition of a `Main` method to your application's code-behind. This is the entry point for your application.

■ **Note** The `Application` class uses the Singleton pattern to ensure that only one instance of the `Application` object is created per `AppDomain`, because the `Application` object is shared throughout an `AppDomain`. For more information on the Singleton pattern, please refer to `http://en.wikipedia.org/wiki/Singleton_pattern`.

The Code

The following example details the default application structure for a simple Microsoft Windows application. The example comprises the following: the `App.xaml` file defines the markup for a `System.Windows.Application` object, with a build action of `ApplicationDefinition`; the `App.xaml.cs`, which contains the `Application` object's code-behind; the `Window1.xaml` file, which contains the markup for the application's main window; and `Window1.xaml.cs`, which contains the window's code-behind.

This is the code for `App.xaml`:

```
<Application x:Class="Recipe_01_01.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

This is the code for App.xaml.cs:

```
using System.Windows;

namespace Recipe_01_01
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
        }
    }
}
```

This is the code for Window1.xaml:

```
<Window
  x:Class="Recipe_01_01.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="300"
  Width="300">
  <Grid>

  </Grid>
</Window>
```

This is the code for Window1.xaml.cs:

```
using System.Windows;

namespace Recipe_01_01
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

1-2. Handle an Unhandled Exception

Problem

You need to handle any unexpected exceptions, allowing you to present the user with an informative dialog box and or log useful debug data.

Solution

Add an event handler to the `System.Windows.Application.DispatcherUnhandledException` event of your application. This will be invoked when an exception has not been handled in code; it allows you to handle the event, allowing the application to continue processing.

How It Works

The default exception handling in WPF will catch any unhandled exceptions that are thrown in the application's main UI thread and display a message to the user. Once the user handles the dialog box, the application shuts down. It is possible, though, to override this default behavior, which allows you to decide what action should be taken. This could be writing to some log file or handling the exception and allowing the application to continue.

To allow an application to provide its own unhandled exception behavior, you need to add a `System.Windows.Threading.DispatcherUnhandledExceptionEventHandler` to the `DispatcherUnhandledException` event on the current application. The handler is passed a `System.Windows.Threading.DispatcherUnhandledExceptionEventArgs` object, which contains a reference to the exception that was unhandled and a flag to indicate whether the exception has been handled. If the exception is marked as being handled, the default WPF exception handling will not kick in. Instead, the operation that was running is halted, but the application will continue running, unless otherwise instructed.

Exceptions raised on threads other than the main UI thread will not be rethrown on the UI thread by default; thus, `DispatcherUnhandledException` does not get raised. If this behavior is required, it will need to be implemented by handling the exception on the owning thread, dispatching it to the UI thread and then rethrowing the exception from the UI thread.

■ **Note** When using the `DispatcherUnhandledException` event to catch unhandled exceptions, you may still find your IDE breaking on an exception and informing you that it is unhandled. This is to be expected if you have your IDE configured to break on unhandled exceptions. Continue the program's execution, and you will see the exception being handled by your custom code.

The Code

The following code demonstrates how to handle the `Application.DispatcherUnhandledException` event. The following markup defines the content of the `App.xaml` file, or whatever name you have given to the file in your project with a build action of `ApplicationDefinition`.

```
<Application
  x:Class="Recipe_01_02.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml"
  DispatcherUnhandledException="App_DispatcherUnhandledException"
/>
```

The following code block defines the code for the code-behind of the previous markup and contains the declaration for `App_DispatcherUnhandledException`:

```
using System;
using System.Windows;
using System.Windows.Threading;

namespace Recipe_01_02
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        private void App_DispatcherUnhandledException (object sender,
                                    DispatcherUnhandledExceptionEventArgs e)
        {
            string msg =
                string.Format("An unhandled exception has occurred.{0}{0}{1}",
                              Environment.NewLine,
                              e.Exception);

            MessageBox.Show(msg, "Recipe_01_02");

            //Handling this event will result in the application
            //remaining alive. This is useful if you are able to
            //recover from the exception.
            e.Handled = true;
        }
    }
}
```

The next code block gives the markup used to define the application's main window. The window contains three `System.Windows.Controls.Button` controls, which demonstrate the behavior of the default WPF exception handling and how it can be overridden.

```
<Window
  x:Class="Recipe_01_02.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
```

```
  Height="135"
  Width="300">
  <StackPanel>
    <Button
      x:Name="btnThrowHandledException"
      Click="btnThrowHandledException_Click"
      Content="Throw Handled Exception"
      Margin="10,10,10,5"
    />

    <Button
      x:Name="btnThrowUnhandledException"
      Click="btnThrowUnhandledException_Click"
      Content="Throw Unhandled Exception"
      Margin="10,5,10,5"
    />

    <Button
      x:Name="btnThrowUnhandledExceptionFromThread"
      Click="btnThrowUnhandledExceptionFromThread_Click"
      Content="Throw Unhandled Exception From a New Thread"
      Margin="10,5,10,10"
    />
  </StackPanel>
</Window>
```

The final code block defines the code-behind for the window defined earlier. It contains the three `Button.Click` event handlers that execute the examples. The first button throws a new `System.NotImplementedException`, which is caught using a `try...catch` block and doesn't progress any further. The second button throws a new `NotImplementedException` that is not handled in code and invokes `DispatcherUnhandledException`, which is handled by `App`. The third button throws a new `NotImplementedException` from a `System.ComponentModel.BackgroundWorker`, illustrating that the exception does not invoke `DispatcherUnhandledException`.

```
using System;
using System.Windows;
using System.ComponentModel;

namespace Recipe_01_02
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
```

```
            InitializeComponent();
        }

        private void btnThrowHandledException_Click(object sender,
                                                    RoutedEventArgs e)
        {
            try
            {
                throw new NotImplementedException();
            }
            catch (NotImplementedException ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private void btnThrowUnhandledException_Click(object sender,
                                                      RoutedEventArgs e)
        {
            throw new NotImplementedException();
        }

        private void btnThrowUnhandledExceptionFromThread_Click(object sender,
                                                                RoutedEventArgs e)
        {
            BackgroundWorker backgroundWorker = new BackgroundWorker();

            backgroundWorker.DoWork += delegate
                {
                    throw new NotImplementedException();
                };

            backgroundWorker.RunWorkerAsync();
        }
    }
}
```

1-3. Create and Use a Dependency Property

Problem

You need to add a property to a `System.Windows.DependencyObject` that provides support for any or all of the following:

- Data bindings

- Animation
- Setting with a dynamic resource reference
- Automatically inheriting a property value from a super-class
- Setting in a style
- Using property value inheritance (see recipe 1-9)
- Notification through a callback when the value changes

This could be for a new UI control you are creating or simply a type that descends from `DependencyObject`.

Solution

Use a `System.Windows.DependencyProperty` as the backing store for the required property on your class.

How It Works

A dependency property is implemented using a standard Common Language Runtime (CLR) property, but instead of using a private field to back the property, a `DependencyProperty` is used. A `DependencyProperty` is instantiated using the static method `DependencyProperty.Register(string name, System.Type propertyType, Type ownerType)`, which returns a `DependencyProperty` instance that is stored using a static, read-only field. There are also two overrides that allow you to specify metadata and a callback for validation.

The first argument passed to the `DependencyProperty.Register` method specifies the name of the dependency property being registered. This name must be unique within registrations that occur in the owner type's namespace (see recipe 1-5 for how to use the same name for a dependency property on several objects inside a common namespace). The next two arguments simply give the type of property being registered and the class against which the dependency property is being defined, respectively. It is important to note that the owning type derives from `DependencyObject`; otherwise, an exception will be raised when the dependency property is initialized.

The first override for the `Register` method allows a `System.Windows.PropertyMetadata` object, or one of the several derived types, to be specified for the property. Property metadata is used to define characteristics of a dependency property, allowing for greater richness than simply using reflection or common CLR characteristics. The use of property metadata can be broken down into three areas:

- Specifying a default value for the property
- Providing callback implementations for property changes and value coercion
- Reporting framework-level characteristics used in layout, inheritance, and so on

■ **Caution** Because values for dependency properties can be set in several places, a set of rules define the precedence of these values and any default value specified in property metadata. These rules are beyond the scope of this recipe; for more information, you can look at the subject of dependency property value precedence at `http://msdn.microsoft.com/en-us/library/ms743230.aspx`.

In addition to specifying a default value, property-changed callbacks, and coercion callbacks, the `System.Windows.FrameworkPropertyMetadata` object allows you to specify various options given by the `System.Windows.FrameworkPropertyMetadataOptions` enumeration. You can use as many of these options as required, combining them as flags. Table 1-1 details the values defined in the `FrameworkPropertyMetadataOptions` enumeration.

Table 1-1. *Values for the `FrameworkPropertyMetadataOptions` Class*

Property	Description
`None`	The property will adopt the default behavior of the WPF property system.
`AffectsMeasure`	Changes to the dependency property's value affect the owning control's measure.
`AffectsArrange`	Changes to the dependency property's value affect the owning control's arrangement.
`AffectsParentMeasure`	Changes to the dependency property's value affect the parent of the owning control's measure.
`AffectsParentArrange`	Changes to the dependency property's value affect the parent of the owning control's arrangement.
`AffectsRender`	Changes to the dependency property's value affect the owning control's render or layout composition.
`Inherits`	The value of the dependency property is inherited by any child elements of the owning type.
`OverridesInheritenceBehaviour`	The value of the dependency property spans disconnected trees in the context of property value inheritance.
`NotDataBindable`	Binding operations cannot be performed on this dependency property.
`BindsTwoWayByDefault`	When used in data bindings, the `System.Windows.BindingMode` is `TwoWay` by default.
`Journal`	The value of the dependency property saved or restored through any journaling processes or URI navigations.
`SubPropertiesDoNotAffectRender`	Properties of the value of the dependency property do not affect the owning type's rendering in any way.

■ **Caution** When implementing a dependency property, it is important to use the correct naming convention. The identifier used for the dependency property must be the same as the identifier used to name the CLR property it is registered against, appended with `Property`. For example, if you were defining a property to store the velocity of an object, the CLR property would be named `Velocity`, and the dependency property field would be named `VelocityProperty`. If a dependency property isn't implemented in this fashion, you may experience strange behavior with property system–style applications and some visual designers not correctly reporting the property's value.

Value coercion plays an important role in dependency properties and comes into play when the value of a dependency property is set. By supplying a `CoerceValueCallback` argument, it is possible to alter the value to which the property is being set. An example of value coercion is when setting the value of the `System.Windows.Window.RenderTransform` property. It is not valid to set the `RenderTransform` property of a window to anything other than an identity matrix. If any other value is used, an exception is thrown. It should be noted that any coercion callback methods are invoked before any `System.Windows.ValidateValueCallback` methods.

The Code

The following example demonstrates the definition of a custom `DependencyProperty` on a simple `System.Windows.Controls.UserControl` (`MyControl`, defined in `MyControl.xaml`). The `UserControl` contains two text blocks: one of which is set by the control's code-behind; the other is bound to a dependency property defined in the control's code-behind.

```
<UserControl
  x:Class="Recipe_01_03.MyControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="20" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <TextBlock x:Name="txblFontWeight" Text="FontWeight set to: Normal."  />

    <Viewbox Grid.Row="1">
      <TextBlock
        Text="{Binding Path=TextContent}"
        FontWeight="{Binding Path=TextFontWeight}" />
    </Viewbox>
  </Grid>
</UserControl>
```

The following code block details the code-behind for the previous markup (MyControl.xaml.cs):

```
using System.Windows;
using System.Windows.Controls;

namespace Recipe_01_03
{
    public partial class MyControl : UserControl
    {
        public MyControl()
        {
            InitializeComponent();
            DataContext = this;
        }

        public FontWeight TextFontWeight
        {
            get { return (FontWeight)GetValue(TextFontWeightProperty); }
            set { SetValue(TextFontWeightProperty, value); }
        }

        public static readonly DependencyProperty TextFontWeightProperty =
            DependencyProperty.Register(
                "TextFontWeight",
                typeof(FontWeight),
                typeof(MyControl),
                new FrameworkPropertyMetadata(FontWeights.Normal,
                            FrameworkPropertyMetadataOptions.AffectsArrange
                            & FrameworkPropertyMetadataOptions.AffectsMeasure
                            & FrameworkPropertyMetadataOptions.AffectsRender,
                            TextFontWeight_PropertyChanged,
                            TextFontWeight_CoerceValue));

        public string TextContent
        {
            get { return (string)GetValue(TextContentProperty); }
            set { SetValue(TextContentProperty, value); }
        }

        public static readonly DependencyProperty TextContentProperty =
            DependencyProperty.Register(
                "TextContent",
                typeof(string),
                typeof(MyControl),
```

```
                new FrameworkPropertyMetadata(
                   "Default Value",
                   FrameworkPropertyMetadataOptions.AffectsArrange
                   & FrameworkPropertyMetadataOptions.AffectsMeasure
                   & FrameworkPropertyMetadataOptions.AffectsRender));


        private static object TextFontWeight_CoerceValue(DependencyObject d,
                object value)
        {
            FontWeight fontWeight = (FontWeight)value;


            if (fontWeight == FontWeights.Bold
                || fontWeight == FontWeights.Normal)
            {
                return fontWeight;
            }

            return FontWeights.Normal;
        }

        private static void TextFontWeight_PropertyChanged(DependencyObject d,
                                    DependencyPropertyChangedEventArgs e)
        {
            MyControl myControl = d as MyControl;

            if (myControl != null)
            {
                FontWeight fontWeight = (FontWeight)e.NewValue;
                string fontWeightName;

                if (fontWeight == FontWeights.Bold)
                    fontWeightName = "Bold";
                else
                    fontWeightName = "Normal";

                myControl.txblFontWeight.Text =
                        string.Format("Font weight set to: {0}.", fontWeightName);
            }
        }
    }
}
```

1-4. Create a Read-Only Dependency Property

Problem

You need to add a read-only dependency property to an object that inherits from `System.Windows.DependencyObject`.

Solution

When registering a dependency property, use `System.Windows.DependencyProperty.RegisterReadOnly` instead of `DependencyProperty.Register` to obtain a reference to a `System.Windows.DependencyPropertyKey`. This is stored in a private field and used to look up the value of the property.

How It Works

The `RegisterReadOnly` method of `DependencyProperty` is similar to the `Register` method in terms of their parameters, although they differ in their return values. Where `Register` returns a reference to a `DependencyProperty` object, `RegisterReadOnly` returns a reference to a `DependencyPropertyKey` object. The `DependencyPropertyKey` exposes two members: a `DependencyProperty` property containing a reference to the `DependencyProperty` created against the key and an `OverrideMetadata` method allowing you to alter the metadata used to describe the property's characteristics.

The `DependencyProperty` property on `DependencyPropertyKey` can directly be used in calls to the `SetValue` and `ClearValue` methods. The `GetValue` method, though, has no such signature. To make a call to `GetValue`, simply pass in the value of `DependencyPropertyKey.DependencyProperty`.

When defining the access modifiers for the various members, it is important to remember that if the field that stores the `DependencyPropertyKey` is public, then other objects will be able to set the value of the property, defeating the object of making the property read-only. The `DependencyProperty` property of the `DependencyPropertyKey` can be exposed, though, and it is recommended that you do so as a `public static readonly DependencyProperty` property. This ensures that certain property system operations can still take place whilst the property remains read-only to external types. Any attempt to create a two-way binding against a read-only property will result in a runtime exception.

The Code

The following code demonstrates a simple XAML file that defines a `System.Windows.Window`. The window contains a `System.Windows.Controls.Viewbox`, which is used to display a `System.Windows.Controls.TextBlock`. The value of the `TextBlock`'s `Text` property is bound to a custom dependency property defined in the window's code-behind file.

```
<Window
  x:Name="winThis"
  x:Class="Recipe_01_04.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Recipe_01_04"
  Height="300"
  Width="300">
    <Grid>
      <Viewbox>
        <TextBlock
          Text="{Binding ElementName=winThis, Path=Counter}"
        />
      </Viewbox>
    </Grid>
</Window>
```

The following code demonstrates a simple application that contains a single `System.Windows.Window`. The window defines a single read-only CLR property that is backed by a `System.Windows.DependencyProperty`, referenced using a `System.Windows.DependencyPropertyKey`. A `System.Windows.Threading.DispatcherTimer` is used to increment the value of the `Counter` property every second.

```
using System;
using System.Windows;
using System.Windows.Threading;

namespace Recipe_01_04
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        /// <summary>
        /// Contructor for the demo's main window. Here a simple dispatcher
        /// timer is created simply for the purpose of demonstrating how
        /// a read-only dependency property can be set.
        /// </summary>
        public Window1()
        {
            InitializeComponent();

            DispatcherTimer timer =
                new DispatcherTimer(TimeSpan.FromSeconds(1),
                                    DispatcherPriority.Normal,
                                    delegate
                                    {
```

```
                                    //Increment the value stored in Counter
                                    int newValue = Counter == int.MaxValue
                                                   ? 0
                                                   : Counter + 1;
                                    //Uses the SetValue that accepts a
                                    //System.Windows.DependencyPropertyKey
                                    SetValue(counterKey, newValue);
                                },
                                Dispatcher);
        }

        /// <summary>
        /// The standard CLR property wrapper. Note the wrapper is
        /// read-only too, so as to maintain consistency.
        /// </summary>
        public int Counter
        {
            get { return (int)GetValue(counterKey.DependencyProperty); }
        }

        /// <summary>
        /// The <see cref="System.Windows.DependencyPropertyKey"/> field which
        /// provides access to the <see cref="System.Windows.DependencyProperty"/>
        /// which backs the above CLR property.
        /// </summary>
        private static readonly DependencyPropertyKey counterKey =
            DependencyProperty.RegisterReadOnly("Counter",
                                                typeof(int),
                                                typeof(Window1),
                                                new PropertyMetadata(0));
    }
}
```

1-5. Override a Dependency Property's Metadata

Problem

You need to override the metadata for a `System.Windows.DependencyProperty` defined in a class higher up in the type's inheritance tree.

Solution

Use the `OverrideMetadata` method of the dependency property for which you want to override the metadata.

■ **Note** It is also possible to override metadata for attached properties, which are, after all, dependency properties. Overridden metadata for an attached property is used only when the attached property is set on an instance of the class performing the overriding.

How It Works

The `OverrideMetadata` method on a `DependencyProperty` allows you to specify new property metadata for a type that has inherited the `DependencyProperty` in question. This is particularly useful if you want to alter any characteristics of the property's metadata. This can be the property's default value, `System.Windows.PropertyMetadata.PropertyChangedCallback` or `System.Windows.PropertyMetadata.CoerceValueCallback`.

There are some things to be aware of when overriding a dependency property's metadata that may make it more favorable to implement your own custom dependency property (see recipe 1-3). It is important to note that a new `ValidateValueCallback` cannot be specified because this is defined outside the scope of the property's metadata. It is also not possible to override the property's value type.

Another caveat is that when overriding property metadata, the overriding metadata's object type must match that of the metadata being overridden. For example, if you are overriding the property metadata on a `System.Windows.FrameworkElement.DataContextProperty`, where the original metadata is defined using a `System.Windows.FrameworkPropertyMetadata` object, overriding metadata must also be a `FrameworkPropertyMetadata` object.

Each characteristic that can be overridden behaves slightly differently in the way it handles existing values, either replacing the existing data with the new data or merging the new data with any existing values. Table 1-2 covers the three characteristics and describes the way in which it acts if any other data is present.

Table 1-2. *Merge/Replacement Behavior for Overidden Metadata*

Characteristic	Description
`DefaultValue`	A new default value will replace the default value of the dependency property for the new owner's type. If no new default value is supplied in the overridden metadata, the default value in the closest ancestor will be used.
`PropertyChangedCallback`	A new `PropertyChangedCallback` implementation will be merged with any existing ones. `PropertyChangedCallback` implementations are executed starting with the most derived type. If no `PropertyChangedCallback` is supplied in the overridden metadata, the `PropertyChangedCallback` in the closest ancestor will be used.
`CoerceValueCallback`	A new `CoerceValueCallback` implementation will replace any existing implementation. If a `CoerceValueCallback` implementation is provided, it will be the only callback to be invoked. If the `CoerceValueCallback` is not overridden, the implementation in the closest ancestor will be used.

The Code

The following example demonstrates a simple `System.Windows.Window` that contains a `System.Windows.Controls.TextBox` and a `System.Windows.Controls.Button`. In the window's code-behind, the metadata for the window's `DataContextProperty` is overridden. The overriding data specifies a new default value and registers a property-changed callback. The first code block defines the window's markup file, and the second defines the window's code-behind.

```
<Window
  x:Class="Recipe_01_05.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="102"
  Width="200">
  <StackPanel>
    <TextBox
      x:Name="tbxUserText"
      Text="Enter some text..."
      Margin="10,10,10,5"
    />
    <Button
      Click="Button_Click"
      Content="Update DataContext"
      Margin="10,5,10,10"
    />
  </StackPanel>
</Window>
```

The following code block defines the code-behind file for the window:

```
using System.Windows;
using System;

namespace Recipe_01_05
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();

            // Override the metadata for the DataContextProperty
            // of the window, altering the default value and
            // registering a property-changed callback.
```

```
            DataContextProperty.OverrideMetadata(
                typeof(Window1),
                new FrameworkPropertyMetadata(
                        100d,
                        new PropertyChangedCallback(DataContext_PropertyChanged)));

        }

        private static void DataContext_PropertyChanged(DependencyObject d,
                                    DependencyPropertyChangedEventArgs e)
        {
            string msg =
                string.Format(
                    "DataContext changed.{0}{0}Old Value: {1}{0}New Value: {2}",
                    Environment.NewLine,
                    e.OldValue.ToString(),
                    e.NewValue.ToString());

            MessageBox.Show(msg, "Recipe_01_05");
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            DataContext = tbxUserText.Text;
        }
    }
}
```

Figure 1-1 shows the dialog box the user will see after clicking the button.

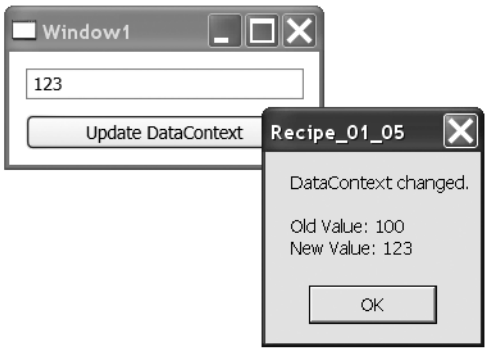

Figure 1-1. *The dialog box shown to the user after clicking the button*

1-6. Add a PropertyChangedValueCallback to Any Dependency Property

Problem

You need to add a `PropertyChangedValueCallback` to a dependency property but are not able to override the property's metadata or access the source property.

Solution

Use the static method `System.ComponentModel.DependencyPropertyDescriptor.FromProperty` to obtain a reference to a `System.ComponentModel.DependencyPropertyDescriptor`. Using the descriptor, you are able to add new `PropertyChangedValueCallback` handlers through the descriptor's `AddValueChanged` method.

How It Works

In obtaining a reference to a `DependencyPropertyDescriptor`, you have a collection of methods and properties that allow you to work with the underlying dependency property, even accessing the property directly. Some of the members of this type are aimed at designers, so they can be ignored.

The method you are interested in is `AddValueChanged`, which accepts two parameters. The first is a reference to the object that owns the dependency property in question. The second parameter is a `System.Windows.EventHandler`, pointing to the method to be called when the dependency property's value changes. There is also a conjugate method named `RemoveValueChanged`, which, as the name suggests, allows you to remove a value-changed event handler. It is best practice to remove any event handlers that are added to objects to ensure that they are properly cleaned up once they are done with.

The Code

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;

namespace Recipe_01_06
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
```

```
        private void Window1_Loaded(object sender, RoutedEventArgs e)
        {
            DependencyPropertyDescriptor descriptor =
                DependencyPropertyDescriptor.FromProperty(TextBox.TextProperty,
                                                          typeof(TextBox));

            descriptor.AddValueChanged(tbxEditMe, tbxEditMe_TextChanged);

        }

        private void tbxEditMe_TextChanged(object sender, EventArgs e)
        {
            //Do something
        }
    }
}
```

1-7. Add Validation to a Dependency Property

Problem

You need to validate a dependency property, ensuring the value being set is in the context of your application's business rules.

Solution

Add a `ValidationValueCallback` handler when specifying the metadata of a dependency property.

How It Works

When setting the value of a dependency property, there are several opportunities to inspect the value and, based on its validity, take some action. The first chance you get to handle the new value coming in is through the `CoerceValueCallback`, which allows you to shape the input data, if required (see recipe 1-3). The second chance is through a `ValidationValueCallback`. Simply supply a method to be used for validation when defining a dependency property's metadata object.

The validation method returns a `bool` indicating the validity of the new value. This allows you to determine whether the value meets your requirements of any business rules, returning true if the conditions are met and otherwise false. If the validation phase passes successfully, any supplied `PropertyChangedCallback` methods are invoked, notifying any listeners that the value of the property has changed.

Validation callbacks are used when the dependency property's value is set in several different scenarios. This covers the default value of the property, default value overrides through overridden property metadata, or setting the property through `System.Windows.DependencyObject.SetValue`, either explicitly through code or implicitly through data binding.

The validation callback takes only a value to validate as an object and does not take a reference to the owning `System.Windows.DependencyObject`. As such, validation callbacks are intended to

provide generic validation to the property in question. Because the type of the object is not known, it is assumed that validation callback methods are aware of the type they are validating. This makes them useful for validating numerical ranges, executing regular expression matches, and so on. They are not useful if the validity of the value in question is in any way influenced by other property values.

Should the validation callback handler return false, a `System.ArgumentException` exception will be raised and handled within the property system. The result of a validation callback returning false is that the value being validated will not be set, and the dependency property will not change and hence not invoke any `PropertyChangedCallback`s.

■ **Note** Because the validation callback parameter is not part of the property's metadata, the validation methods cannot be overridden.

The Code

The following code demonstrates a simple application with a single window. The window contains a text block into which the user can enter a number. The `Text` property of the `System.Windows.Controls.TextBox` control is bound to a `System.Windows.DependencyProperty`, `UserValue`, defined in the window's code-behind. The value of the `UserValue` dependency property is also bound to a text block in the window, reflecting the actual value of the property. The color of the text in the text box is set to green when the given value is valid; otherwise, it is set to red.

■ **Note** When the value input into the text box goes from being invalid to valid, through the deletion of the characters entered to invalidate it, the text box's text color will remain red. This is a result of the dependency property holding the last valid value it was set with. In this scenario, when the text goes from invalid to valid, the value will be equal to that held by the dependency property; therefore, the property's value doesn't actually change. For example, try entering the number 1000 and then delete the last 0.

```
<Window
  x:Class="Recipe_01_07.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="300"
  Width="300">
  <Grid>
    <StackPanel>
      <TextBlock
        Text="Please enter a value between 1 and 100, inclusive."
        Margin="5" />
```

```
      <TextBox
        x:Name="uv"
        Text="{Binding Path=UserValue, UpdateSourceTrigger=PropertyChanged}"
        Margin="5"
        PreviewKeyDown="TextBox_PreviewKeyDown" />
      <StackPanel Orientation="Horizontal">
        <TextBlock
          Margin="5"
          Text="UserValue1 Value:" />
        <TextBlock
          x:Name="userValueValue"
          Margin="5"
          Text="{Binding Path=UserValue}" />
      </StackPanel>
    </StackPanel>
  </Grid>
</Window>
```

The following code provides the listing for the window's code-behind file. The single dependency property is bound to the window's `TextBox.Text` property. A validation callback is supplied, as is a property-changed callback. The property-changed callback will be invoked only if the validation callback returns true. As text is entered into the text box, the text box's foreground color is set to red, indicating it is invalid. When the property-changed handler is invoked, it sets the color to green, indicating the entered value is valid.

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

namespace Recipe_01_07
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
            //Set the window's DataContext to itself to simplify
            //the binding paths for the UserValue property.
            DataContext = this;
        }

        //The CLR wrapper for the DependencyProperty
        public int UserValue
        {
            get { return (int)GetValue(UserValueProperty); }
            set { SetValue(UserValueProperty, value); }
        }
```

```
//The dependency property backing store.
public static readonly DependencyProperty UserValueProperty =
    DependencyProperty.Register("UserValue",
                                typeof(int),
                                typeof(Window1),
                                new PropertyMetadata(
                                        1,
                                        UserValue_PropertyChangedCallback),
                                ValidateIntRange_ValidationCallbackHandler);

//Validation callback for the above UserValue dependency property.
private static bool ValidateIntRange_ValidationCallbackHandler(object value)
{
    //Try to parse the value to an int.
    int intValue;

    if (int.TryParse(value.ToString(), out intValue))
    {
        //The value is an integer so test its value.
        if (intValue >= 1 && intValue <= 100)
        {
            return true;
        }
    }

    return false;
}

//Property-changed callback for the above UserValue dependency property.
private static void UserValue_PropertyChangedCallback(DependencyObject d,
                                    DependencyPropertyChangedEventArgs e)
{
    Window1 window1 = d as Window1;

    if (window1 != null)
    {
        window1.uv.Foreground = Brushes.SeaGreen;
    }
}

//Handler for the PreviewKeyDown event on the TextBox
private void TextBox_PreviewKeyDown(object sender, RoutedEventArgs e)
{
    TextBox textBox = sender as TextBox;
```

```
            if (textBox != null)
            {
                textBox.Foreground = Brushes.Firebrick;
            }
        }
    }
}
```

1-8. Create and Use an Attached Property

Problem

You need to add a dependency property to a class but are not able to access the class in a way that would allow you to add the property, or you want to use a property that can be set on any child objects of the type.

Solution

Create an attached property by registering a `System.Windows.DependencyProperty` using the static `DependencyProperty.RegisterAttached` method.

How It Works

You can think of an attached property as a special type of dependency property that doesn't get exposed using a CLR property wrapper. You will more than likely have come across a few examples of attached properties in your XAML travels. Examples of everyday attached properties are `System.Windows.Controls.Canvas.Top`, `System.Windows.Controls.DockPanel.Dock`, and `System.Windows.Controls.Grid.Row`.

As attached properties are registered in a similar way to dependency properties, you are still able to provide metadata for handling property changes, and so on. In addition to metadata, it is possible to enable property value inheritance on attached properties (see recipe 1-9).

Attached properties are not set like dependency properties using a CLR wrapper property; they are instead accessed through a method for getting and setting their value. These methods have specific signatures and naming conventions so that they can be matched up to the correct attached property. The signatures for the property's getter and setter methods can be found in the following code listing.

The Code

The following code defines a simple `System.Windows.Window` that contains a few controls. In the window's code-behind, an attached property is defined with `SystemWindows.UIElement` as the target type. The window's markup defines four controls, three of which have the value of `Window1.Rotation` set in XAML. The button's value for this property is not set and will therefore return the default value for the property, 0 in this case.

```
<Window
  x:Class="Recipe_01_08.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:Recipe_01_08"
  Title="Recipe_01_08"
  Height="350"
  Width="350">
  <UniformGrid>
    <Button
      Content="Click me!"
      Click="UIElement_Click"
      Margin="10"
    />

    <Border
      MouseLeftButtonDown="UIElement_Click"
      BorderThickness="1"
      BorderBrush="Black"
      Background="Transparent"
      Margin="10"
      local:Window1.Rotation="3.14"
    />

    <ListView
      PreviewMouseLeftButtonDown="UIElement_Click"
      Margin="10"
      local:Window1.Rotation="1.57">
      <ListViewItem Content="Item 1" />
      <ListViewItem Content="Item 1" />
      <ListViewItem Content="Item 1" />
      <ListViewItem Content="Item 1" />
    </ListView>

    <local:UserControl1
      Margin="10"
      local:Window1.Rotation="1.0"
    />
  </UniformGrid>
</Window>

using System.Windows;
using System.Windows.Controls;
```

```
namespace Recipe_01_08
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void UIElement_Click(object sender, RoutedEventArgs e)
        {
            UIElement uiElement = (UIElement)sender;

            MessageBox.Show("Rotation = " + GetRotation(uiElement), "Recipe_01_08");
        }

        public static readonly DependencyProperty RotationProperty =
            DependencyProperty.RegisterAttached("Rotation",
                                                typeof(double),
                                                typeof(Window1),
                                                new FrameworkPropertyMetadata(
                                0d, FrameworkPropertyMetadataOptions.AffectsRender));

        public static void SetRotation(UIElement element, double value)
        {
            element.SetValue(RotationProperty, value);
        }

        public static double GetRotation(UIElement element)
        {
            return (double)element.GetValue(RotationProperty);
        }
    }
}
```

The following markup and code-behind define a simple `System.Windows.Controls.UserControl` that demonstrates the use of the custom attached property in code:

```
<UserControl
  x:Class="Recipe_01_08.UserControl1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  MouseLeftButtonDown="UserControl_MouseLeftButtonDown"
  Background="Transparent">
  <Viewbox>
    <TextBlock Text="I'm a UserControl" />
  </Viewbox>
</UserControl>

using System.Windows;
using System.Windows.Controls;

namespace Recipe_01_08
{
    /// <summary>
    /// Interaction logic for UserControl1.xaml
    /// </summary>
    public partial class UserControl1 : UserControl
    {
        public UserControl1()
        {
            InitializeComponent();
        }

        private void UserControl_MouseLeftButtonDown(object sender,
                                                     RoutedEventArgs e)
        {
            UserControl1 uiElement = (UserControl1)sender;

            MessageBox.Show("Rotation = " + Window1.GetRotation(uiElement),
                            "Recipe_01_08");
        }
    }
}
```

Figure 1-2 shows the result of clicking the button. A value for the `Window1.Rotation` property is not explicitly set on the button; therefore, it is displaying the default value.

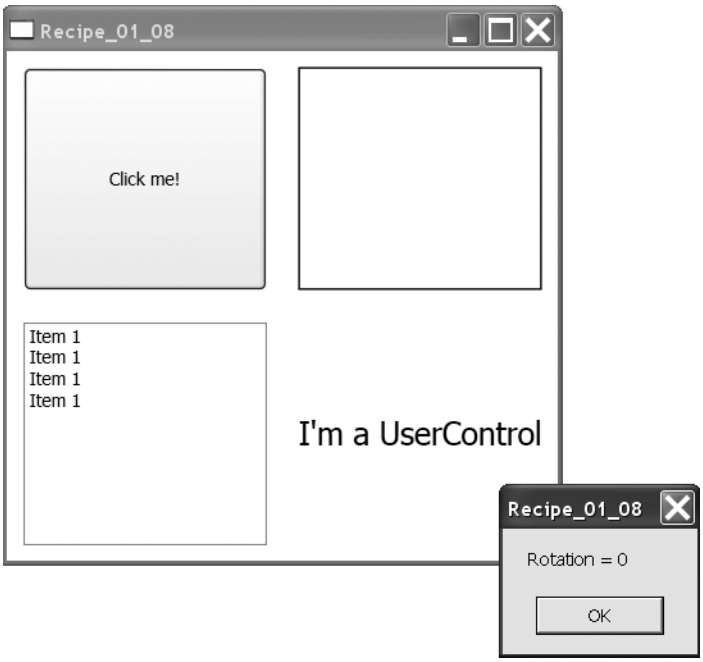

Figure 1-2. *The result of clicking the button*

1-9. Create a Dependency Property with Property Value Inheritance

Problem

You need to create a dependency property on a dependency object where the value is pushed down the visual tree.

Solution

When registering the dependency property, use a `FrameworkPropertyMetadata` object and include `FrameworkPropertyMetadataOptions.Inherits` when specifying the framework-level options.

How It Works

Property value inheritance can be extremely useful under the right circumstances. It applies only to objects that reside in a visual tree and have one or more child objects. Property value inheritance is used to push the value of a property down onto each child object that also contains the same property.

One of the most common examples of property value inheritance can be found in the `DataContext` dependency property defined in `System.Windows.FrameworkElement`. The behavior

can be observed when the data context of a parent control is set. For example, if you have a grid that contains several text elements, the value set on the data context on the grid will be pushed down to each of the text elements. This is useful for reducing the code required when the value of a property is applicable to its children.

The Code

The following code details the content of the markup for the application's main window. The window contains four `System.Windows.Controls.TextBlock` controls to display the values of four properties. Two of the properties belong to a control that is the child of another control.

The `PropertyThatInherits` property uses property value inheritance. Any value assigned to that property will be pushed down to any child controls. The `PropertyThatDoesNotInherit` property is used to show that a property that does not use the `FrameworkPropertyMetadataOptions.Inherits` does not employ property value inheritance.

```
<Window
  x:Class="Recipe_01_09.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:local="clr-namespace:Recipe_01_09"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="175"
  Width="320"
  Loaded="Window1_Loaded">
  <StackPanel>

    <Border
      BorderThickness="1"
      BorderBrush="Black"
      Margin="10,10,10,5">
      <StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10,10,10,5">
          <TextBlock Text="Parent.PropertyThatInherits: " />
          <TextBlock Text="{Binding Path=[0].PropertyThatInherits}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10,5,10,5">
          <TextBlock Text="Child.PropertyThatInherits: " />
          <TextBlock Text="{Binding Path=[1].PropertyThatInherits}" />
        </StackPanel>
      </StackPanel>
    </Border>

    <Border
      BorderThickness="1"
      BorderBrush="Black"
      Margin="10,5,10,10">
```

```
      <StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10,5,10,10">
          <TextBlock Text="Parent.PropertyThatDoesNotInherit: " />
          <TextBlock Text="{Binding Path=[0].PropertyThatDoesNotInherit}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10,5,10,10">
          <TextBlock Text="Child.PropertyThatDoesNotInherit: " />
          <TextBlock Text="{Binding Path=[1].PropertyThatDoesNotInherit}" />
        </StackPanel>
      </StackPanel>
    </Border>
  </StackPanel>
</Window>
```

The following code defines the content of the code-behind for the window defined in the previous markup:

```
using System.Windows;

namespace Recipe_01_09
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Window1_Loaded(object sender, RoutedEventArgs e)
        {
            Parent parent = new Parent();
            parent.PropertyThatInherits = "Still Inherits.";
            parent.PropertyThatDoesNotInherit = "Still not inheriting.";

            Child child = new Child();
            parent.Children.Add(child);

            DataContext = new object[]{parent, child};
        }
    }
}
```

The next code block details the `Parent` class in which the two dependency properties are defined:

```
using System.Windows;
using System.Windows.Controls;

namespace Recipe_01_09
{
    public class Parent : StackPanel
    {
        public string PropertyThatInherits
        {
            get { return (string)GetValue(PropertyThatInheritsProperty); }
            set { SetValue(PropertyThatInheritsProperty, value); }
        }

        public static readonly DependencyProperty PropertyThatInheritsProperty
            = DependencyProperty.RegisterAttached("PropertyThatInherits",
                                                  typeof(string),
                                                  typeof(UIElement),
                            new FrameworkPropertyMetadata("Inherits.",
                                    FrameworkPropertyMetadataOptions.Inherits));

        public string PropertyThatDoesNotInherit
        {
            get { return (string)GetValue(PropertyThatDoesNotInheritProperty); }
            set { SetValue(PropertyThatDoesNotInheritProperty, value); }
        }

        public static readonly DependencyProperty PropertyThatDoesNotInheritProperty
            = DependencyProperty.RegisterAttached("PropertyThatDoesNotInherit",
                                                  typeof(string),
                                                  typeof(UIElement),
                                new FrameworkPropertyMetadata("Does not inherit."));
    }
}
```

The final code block details the `Child` class. This class inherits from `Parent` and contains no new members. Its sole purpose is to illustrate property value inheritance.

```
namespace Recipe_01_09
{
    public class Child : Parent
    {
    }
}
```

Figure 1-3 shows the values of the parent and child's dependency properties.

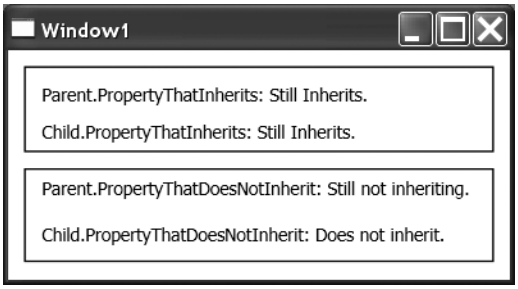

Figure 1-3. *Values of the parent and child's dependency properties*

1-10. Merge Two Resource Dictionaries

Problem

You need to reference objects contained in some `System.Windows.ResourceDictionary` that is not part of your `System.Windows.Controls.Control`.

Solution

Use the `ResourceDictionary.MergedDictionaries` property to merge a `ResourceDictionary` into any other `ResourceDictionary`.

How It Works

The `ResourceDictionary.MergedDictionaries` property gets a `System.Collections.ObjectModel.Collection<ResourceDictionary>`, containing any `ResourceDictionary` objects that are to be merged into the original `ResourceDictionary`. Each `ResourceDictionary` that is added to the collection of resource dictionaries does not contain any elements but instead references a `ResourceDictionary` by setting the `Source` property with a `System.Uri`. The URI can point to a `ResourceDictionary` in the same assembly or in an entirely different assembly (see recipe 1-12); the only stipulation is that the destination of the URI must be a XAML file with `ResourceDictionary` as its root element.

Each time a `ResourceDictionary` is added to the collection, it becomes the first `ResourceDictionary` to be searched when a resource is referenced. The search will stop as soon as it finds the required key. One would assume that this would mean a `ResourceDictionary` cannot be merged into another `ResourceDictionary` if they have any common keys. In reality, duplicate keys are valid because each merged `ResourceDictionary` is held in its own scope, just under the scope of the parent `ResourceDictionary`.

The fact that merged resource dictionaries are held outside the scope of the parent `ResourceDictionary` means that if a key exists in both the parent and a merged resource dictionary, the value that is returned will always be the object that maps to the key found in the parent.

The Code

The following example demonstrates the merging of an external `ResourceDictionary` into a local `ResourceDictionary`, defined within a window's local resources. The `System.Windows.Window` defines a `System.Windows.Media.SolidColorBrush` resource in its local resources that is used as the color for the `System.Windows.Controls.Button` background property. A second `SolidColorBrush` is defined in a resource dictionary that lies outside the window's XAML file. This second brush is used to provide the background color for the window (see Figure 1-4).

```
<Window
  x:Class="Recipe_01_10.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Recipe_01_10"
  Height="300"
  Width="300"
  Background="{DynamicResource WindowBackgroundBrush}">
  <Window.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="ExternalResourceDictionary.xaml" />
      </ResourceDictionary.MergedDictionaries>

      <SolidColorBrush x:Key="ButtonBackground" Color="Yellow" />
    </ResourceDictionary>
  </Window.Resources>

  <Button
    Background="{DynamicResource ButtonBackground}"
    Height="20"
    Width="20"
  />
</Window>
```

The following code gives the content of the external resource dictionary that is merged into the previous window's local resources:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <SolidColorBrush x:Key="WindowBackgroundBrush" Color="HotPink" />

</ResourceDictionary>
```

Figure 1-4. *Using resources from both local and external resource dictionaries*

1-11. Define Application-wide Resources

Problem

You have several resources that you want to make available throughout your application.

Solution

Merge all the required `System.Windows.ResourceDictionary` objects into the application's `ResourceDictionary`.

How It Works

`ResourceDictionary` objects are by default available to all objects that are within the scope of the application. This means that some `System.Windows.Controls.Control` that is placed within a `System.Windows.Window` will be able to reference objects contained within any of the `ResourceDictionary` objects referenced at the application level. This ensures the maintainability of your styles because you will need to update the objects only in a single place.

It is important to know that each time a `ResourceDictionary` is referenced by a `System.Windows.Controls.Control`, a local copy of that `ResourceDictionary` is made for each instance of the control. This means that if you have several large `ResourceDictionary` objects that are referenced by a control that is instantiated several times, you may notice a performance hit.

■ **Note** `System.Windows.Controls.ToolTip` styles need to be referenced once per control. If several controls all use a `ToolTip` style referenced at the application level, you will observe strange behavior in your tool tips.

The Code

The following example demonstrates the content of an application's `App.xaml`. Two `System.Windows.Media.SolidColorBrush` resources are defined that are referenced in other parts of the application.

```
<Application
  x:Class="Recipe_01_11.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml">
  <Application.Resources>
    <SolidColorBrush x:Key="FontBrush" Color="#FF222222" />
    <SolidColorBrush x:Key="BackgroundBrush" Color="#FFDDDDDD" />
  </Application.Resources>
</Application>
```

The following example demonstrates the content of the application's `Window1.xaml` file. The two resources that were defined in the application's resources are used by controls in the `System.Windows.Window`. The first resource is used to set the background property of the outer `System.Windows.Controls.Grid`, and the second resource is used to set the foreground property of a `System.Windows.Controls.TextBlock` (see Figure 1-5).

```
<Window
  x:Class="Recipe_01_11.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Recipe_01_11"
  Height="300"
  Width="300">
  <Grid Background="{StaticResource BackgroundBrush}">
    <Viewbox>
      <TextBlock
        Text="Some Text"
        Foreground="{StaticResource FontBrush}"
        Margin="5"
      />
    </Viewbox>
  </Grid>
</Window>
```

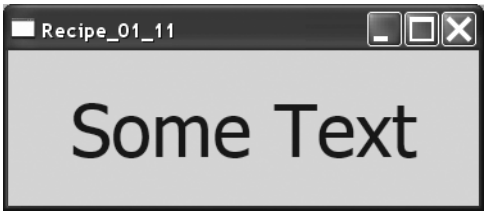

Figure 1-5. *Using an application-level resource to set properties on controls*

1-12. Reference a ResourceDictionary in a Different Assembly

Problem

You need to reference a `System.Windows.ResourceDictionary` that is held in a different assembly than the current one.

Solution

You add the required `ResourceDictionary` using the same approach as merging a `ResourceDictionary` within the same assembly (see recipe 1-10) but use a Pack URI to specify the value for the resource dictionary's `Source` property.

How It Works

The only difference between referencing a local `ResourceDictionary` and a `ResourceDictionary` contained in some other assembly, referenced by the project, is that the `Source` for the external `ResourceDictionary` is specified using a Pack URI.

Amongst other things, the Pack URI approach allows you to reference components in assemblies referenced by your project. A Pack URI uses the `pack` prefix to indicate the pack scheme is being used. The scheme has two components that define an authority and a path, taking the form of `pack://authority/path`. Of the two supported authorities, we are interested in `application:///`, which is used when the target resources are files known at compile time. Because the authority is an embedded URI, the / character must be replaced with a comma (`,`) character.

■ **Note** As well as / characters, other reserved characters such as ? and & must be escaped. For a full definition of the restrictions, see the open packaging conventions on naming restrictions at `http://www.ecma-international.org/news/TC45_current_work/Office%20Open%20XML%20Part%201%20-%20Fundamentals.pdf`, section 9.1.1 in particular.

The path component of the URI consists of the name of the external assembly being referenced, which is `;component:` to indicate that the assembly being referenced in the URI is referenced by the current assembly. The remainder of the path component is then used to define the path to the target resource, relative to the project folder of the assembly being referenced. For example, to reference a resource file called `ResourceFile.xaml` located in a folder named `Resources` within the assembly `MyExternalAssembly`, the URI would take the form of `pack://application:,,,/MyExternalAssembly;component:Resources/ResourceFile.xaml`.

■ **Note** The Pack URI also supports two optional parameters in the path component that can be used when referencing an assembly different from the owning one. These parameters can be used to indicate a specific version of the assembly to be used or a public key that was used to sign the assembly. The full format for the path component is `pack://application,,,/AssemblyName[;Version][;PublicKey];component/Path`.

At least one of these parameters is required when different versions of the same assembly are referenced by the assembly. For example, to specify that version 1.2.3.4 of the assembly `MyExternalAssembly` should be used when accessing the `ResourceFile.xaml` resource, the following URI would be used: `pack://application:,,,/MyExternalAssembly;1.2.3.4;component:Resources/ResourceFile.xaml`.

The Code

The following example demonstrates the merging of an external `ResourceDictionary` object into a local resource dictionary. The external dictionary is defined in the `Recipe_01_10` project; thus, that project must be referenced by this project. The `App.xaml` and `Window1.xaml.cs` files are unchanged and have been omitted.

```
<Window
  x:Class="Recipe_01_12.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="200"
  Width="200"
  Background="{DynamicResource WindowBackgroundBrush}">
  <Window.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source=➥
"pack://application:,,,/Recipe_01_10;component/ExternalResourceDictionary.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Window.Resources>
</Window>
```

1-13. Share Properties Throughout an Application

Problem

You need to share a set of properties throughout the scope of your application, such as user preferences, and so on.

Solution

Use the `Properties` property on your `System.Windows.Application` object.

How It Works

The `Application` type provides a thread-safe method for sharing properties throughout your application, accessible by any thread within the current `AppDomain`. The `Properties` property returns a `System.Collections.IDictionary` in which objects can be stored using some predefined key.

The Code

The following example demonstrates a very trivial use of the `Application.Properties` property. Because the application's definition is unchanged, it has been omitted (see Figure 1-6). The next code block details markup for the main window in the application.

```
<Window
  x:Class="Recipe_01_13.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1"
  Height="102"
  Width="300">
  <StackPanel>
    <TextBox
      x:Name="tbxUserText"
      Text="Enter some text..."
      Margin="10,10,10,5"
    />
    <Button
      Click="Button_Click"
      Content="Open a New Window..."
      Margin="10,5,10,10"
    />
  </StackPanel>
</Window>
```

The following code block details a simple helper class that wraps interaction with the `Application.Properties` property:

```
using System.Windows;

namespace Recipe_01_13
{
    /// <remark>
    /// This helper is intended to provide the base for a helper,
    /// which simplifies use of the Application.Properties property.
    /// The obvious next step in extending the class is to add
```

```
/// argument validation.
/// </remark>
public static class ApplicationPropertiesHelper
{
    /// <summary>
    /// Tries to retrieve a property from the Application's Properties
    /// collection. If the object with the specified key cannot be found,
    /// the default value for the supplied type is returned.
    /// </summary>
    /// <typeparam name="T">The type of object to retrieve.</typeparam>
    /// <param name="key">The key with which the object was stored.</param>
    /// <returns>If the specified key exists, then the associated
    /// value is returned, otherwise the default value for the
    /// specified type.</returns>
    public static T GetProperty<T>(object key)
    {
        if (Application.Current.Properties.Contains(key)
            && Application.Current.Properties[key] is T)
        {
            return (T)Application.Current.Properties[key];
        }

        return default(T);
    }

    /// <summary>
    /// Retrieves the property associated with the given key.
    /// </summary>
    /// <param name="key">The key with which the object was stored.</param>
    /// <returns>If the specified key exists, the associated
    /// value is returned, otherwise the return value is null.</returns>
    public static object GetProperty(object key)
    {
        if (Application.Current.Properties.Contains(key))
        {
            return Application.Current.Properties[key];
        }

        return null;
    }

    /// <summary>
    /// Adds a value to the Application's properties collection,
    /// indexed by the supplied key.
    /// </summary>
    /// <param name="key">
    /// The key against which the value should be stored.</param>
```

```
        /// <param name="value">The value to be stored.</param>
        public static void SetProperty(object key, object value)
        {
            Application.Current.Properties[key] = value;
        }
    }
}
```

The following code block details the code for the previous window's code-behind. When the button in the window is clicked, the `ApplicationPropertiesHelper` object is used to set a value in the `Application.Properties` property. A second window is then opened.

```
using System.Windows;

namespace Recipe_01_13
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            ApplicationPropertiesHelper.SetProperty("PropertyKey",
                                                    tbxUserText.Text);

            Window2 window2 = new Window2();

            window2.ShowDialog();
        }
    }
}
```

The final two code blocks define the markup and code-behind for a second window. This second window looks up a known key in the application's shared properties collection and displays the value in a `System.Windows.Controls.TextBlock`.

```
<Window
  x:Class="Recipe_01_13.Window2"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window2"
  Height="300"
  Width="300"
```

```
  Loaded="Window_Loaded">
  <Viewbox Margin="10">
    <TextBlock x:Name="tbxUserText" />
  </Viewbox>
</Window>
```

```
using System.Windows;

namespace Recipe_01_13
{
    /// <summary>
    /// Interaction logic for Window2.xaml
    /// </summary>
    public partial class Window2 : Window
    {
        public Window2()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            string value =
                ApplicationPropertiesHelper.GetProperty<string>("PropertyKey");

            if (string.IsNullOrEmpty(value))
            {
                value = "Nothing to display!";
            }

            tbxUserText.Text = value;
        }
    }
}
```

Figure 1-6. *The second window, shown in the foreground, opened from clicking the button in the first window, shown in the background*

1-14. Create a Single-Instance Application

Problem

You need to ensure that only one instance of your application is running at any one time. Each time a new instance of the application is started, you need to receive any startup arguments.

Solution

Create a class that derives from `Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase`, and use it to wrap your WPF `System.Windows.Application`. The wrapper is initialized by supplying your own implementation of `Main`.

How It Works

The `WindowsFormsApplicationBase` class provides all the necessary functionality to ensure only a single instance of your application is running, whilst providing notification of any command-line arguments passed to new instances of your application. It also allows you to do this without getting your hands dirty with `System.Thread.Mutex` objects, and so on. This is ideally suited to some form of MDI object viewer, for example, a tabbed XPS reader where any new documents that are opened are added to an existing instance in a new tab.

To use this functionality, you need to change the basic structure of your project and create the class that will wrap your application's `Application` object. This will allow you to control what happens when a new instance of your application is started. The first step is to create a class that inherits from `WindowsFormsApplicationBase`. In the constructor of your wrapper, you will need to set the `IsSingleInstance` to true, indicating you want only one instance of the application to be permitted.

The next step is to override the `OnStartup` method. This method is invoked when the application first runs and is used to start up your WPF application. One more method will require overriding, but only if you need to be notified of attempts to create a new instance of your application. This is the aptly named `OnStartupNextInstance` method. The `Microsoft.VisualBasic.ApplicationServices.StartupNextInstanceEventArgs` passed to the method can be used to retrieve any arguments used when the creation of a new instance of the application is attempted.

Now that you have your wrapper, you need a place to create it when the application runs. The best place for this is in the application's `Main` method, which in WPF applications you no longer need to explicitly define. To define your own implementation of `Main`, add a new public class to your project in a file named `App.cs`. The class should contain a single method with the following signature:

```
public static void Main(string[] args)
```

It will be this method that is responsible for creating a new instance of your wrapper and calling its `Run` method, passing the application's startup arguments. The `Run` method carries out a lot of work behind the scenes so that it can intercept any new attempts at starting the application. The final modification required for the project is to modify the build action of your application's `App.xaml` file. All you need to do here is change the build action in the Properties

window from `ApplicationDefinition` to `Page`. Failure to do so will result in a compile-time error because the compiler will detect that your application has two entry points.

The Code

The following code demonstrates a simple single-instance application. The main window of the application contains controls that allow a new instance of the application to be started, with some user-defined arguments, entered into a `System.Windows.Controls.TextBox`. If a new instance of the application is started and another instance already exists, the `System.Windows.MessageBox` is shown, displaying the arguments that were passed to the already running instance. Because the code contained in the default `App.xaml` and `App.xaml.cs` files does not require changing, its code is not listed here.

The first code block details the implementation of the `SingleInstanceManager`, responsible for initializing the inner WPF application when `Run` is called and handling new arguments that are passed to an already running instance.

```
using System;
using System.Windows;
using Microsoft.VisualBasic.ApplicationServices;

namespace Recipe_01_14
{
    public class SingleInstanceManager : WindowsFormsApplicationBase
    {
        public SingleInstanceManager()
        {
            this.IsSingleInstance = true;
        }

        protected override bool OnStartup(
            Microsoft.VisualBasic.ApplicationServices.StartupEventArgs eventArgs)
        {
            base.OnStartup(eventArgs);

            App app = new App();
            app.Run();

            return false;
        }

        protected override void OnStartupNextInstance(
            StartupNextInstanceEventArgs eventArgs)
        {
            base.OnStartupNextInstance(eventArgs);

            string args = Environment.NewLine;
```

```
            foreach(string arg in eventArgs.CommandLine)
            {
                args += Environment.NewLine + arg;
            }

            string msg = string.Format("New instance started with {0} args.{1}",
                                       eventArgs.CommandLine.Count,
                                       args);
            MessageBox.Show(msg);
        }
    }
}
```

The next code block details the content of the App.cs file where the application's main entry point is defined:

```
using System;

namespace Recipe_01_14
{
    public class MyApp
    {
        [STAThread]
        public static void Main(string[] args)
        {
            //Create our new single-instance manager
            SingleInstanceManager manager = new SingleInstanceManager();
            manager.Run(args);
        }
    }
}
```

The next code block details the content of the application's main window markup:

```
<Window
  x:Class="Recipe_01_14.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="300" Width="300">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="30" />
    </Grid.RowDefinitions>
```

```
    <TextBox x:Name="tbxArgs" />

    <Button
      Content="Start New Process"
      Click="btnCreateNewInstance_Click"
      Grid.Row="1"
    />
  </Grid>
</Window>
```

The final code block details the content of the code-behind for the application's main window:

```
using System;
using System.Diagnostics;
using System.Windows;

namespace Recipe_01_14
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void btnCreateNewInstance_Click(object sender, RoutedEventArgs e)
        {
            Process proc = new Process();

            proc.StartInfo.FileName =
                string.Format("{0}{1}",
                              Environment.CurrentDirectory,
                              "\\Recipe_01_14.exe");

            proc.StartInfo.Arguments = tbxArgs.Text;

            proc.Start();
        }
    }
}
```

1-15. Manage Multiple Windows in an Application

Problem

You need to manage several different windows within your application, performing such tasks as preventing a window from closing, displaying a list of thumbnails of the windows, showing or hiding a window, and bringing a window into view or closing the window.

Solution

Use the `System.Windows.Application.Current.Windows` collection to get a `System.Windows.WindowCollection`, containing all the applications that are contained within your project.

How It Works

The `Windows` property of an `Application` object maintains a list of all the windows within the current `System.AppDomain`. Any window that is created on the UI thread is automatically added to the collection and removed when the window's `Closing` event has been handled but before the `Closed` event is raised (see Figure 1-7).

The Code

```
<Window
  x:Class="Recipe_01_15.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:system="clr-namespace:System.Windows;assembly=PresentationFramework"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Main Window"
  Height="310"
  Width="280"
  Loaded="Window1_Loaded"
  Closing="Window1_Closing">
  <Window.Resources>
    <DataTemplate DataType="{x:Type Window}" x:Key="WindowTemplate">
      <StackPanel>
        <Rectangle Height="50" Width="50">
          <Rectangle.Fill>
            <VisualBrush Visual="{Binding}" />
          </Rectangle.Fill>
        </Rectangle>

        <TextBlock Text="{Binding Path=Title}" />
      </StackPanel>
    </DataTemplate>
  </Window.Resources>
```

```
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="100" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <ListBox x:Name="lbxWindows" ItemTemplate="{StaticResource WindowTemplate}">
      <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
          <WrapPanel />
        </ItemsPanelTemplate>
      </ListBox.ItemsPanel>
    </ListBox>

    <StackPanel Grid.Row="1">
      <CheckBox
        x:Name="cbxIsVisibleInTaskBar"
        Content="IsVisibleInTaskbar"
        IsChecked="{Binding ElementName=lbxWindows,
                            Path=SelectedItem.ShowInTaskbar}"
        Margin="10"
      />

      <CheckBox
        x:Name="cbxIsVisible"
        Content="IsVisible"
        IsChecked="{Binding ElementName=lbxWindows,
                            Path=SelectedItem.IsVisible,
                            Mode=OneWay}"
        Checked="CheckBox_Checked_Changed"
        Unchecked="CheckBox_Checked_Changed"
        Margin="10"
      />

      <CheckBox
        x:Name="cbxCanClose"
        Content="CanClose"
        IsChecked="True"
        Margin="10"
      />

      <Button Content="Bring To Front" Click="btnBringToFront_Click" Margin="10" />
      <Button Content="Close" Click="btnClose_Click" Margin="10" />
    </StackPanel>
  </Grid>
</Window>
```

The following code block contains the code-behind for the previous markup file. The code handles several control events and sets up some windows.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Media;

namespace Recipe_01_15
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        /// <summary>
        /// When the main window is loaded, we want to spawn some
        /// windows to play with.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Window1_Loaded(object sender, RoutedEventArgs e)
        {
            Brush[] backgrounds = new Brush[5]{ Brushes.Red,
                                                Brushes.Blue,
                                                Brushes.Green,
                                                Brushes.Yellow,
                                                Brushes.HotPink};
            //Create 5 windows.
            for (int i = 1; i <= 5; i++)
            {
                Window window = new Window();

                SetupWindow(window, "Window " + i, backgrounds[i - 1]);
                //Show the window.
                window.Show();
            }

            RebuildWindowList();
        }
```

```
/// <summary>
/// When the main window closes, we want to close all the child windows.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Window1_Closing(object sender, CancelEventArgs e)
{
    Application.Current.Shutdown();
}

private void SetupWindow(Window window, string title, Brush background)
{
    // We want to know when a window is closing so we can prevent
    // it from being closed if required.
    window.Closing += new CancelEventHandler(Window_Closing);

    // We want to know when a window has been closed so we can
    // rebuild our list of open windows.
    window.Closed += new EventHandler(Window_Closed);

    // Give the window a title so we can track it.
    window.Title = title;
    window.Width = 100d;
    window.Height = 100d;

    // Create a text block displaying the window's title inside
    // a view box for the window's content.
    Viewbox viewBox = new Viewbox();
    TextBlock textBlock = new TextBlock();

    //Set the window's background to make it easier to identify.
    window.Background = background;
    viewBox.Child = textBlock;
    textBlock.Text = window.Title;
    window.Content = viewBox;
}

/// <summary>
/// This method iterates all the windows for this application
/// and adds them to the list box lbxWindows.
/// </summary>
private void RebuildWindowList()
{
    List<Window> windows = new List<Window>();
```

```
    foreach (Window window in Application.Current.Windows)
    {
        if (window == this)
            continue;

        windows.Add(window);
    }

    lbxWindows.ItemsSource = windows;
}

private void Window_Closed(object sender, EventArgs e)
{
    RebuildWindowList();
}

private void Window_Closing(object sender, CancelEventArgs e)
{
    Window w = sender as Window;

    if (w == null)
        return;

    e.Cancel = !cbxCanClose.IsChecked == true;
}

private void CheckBox_Checked_Changed(object sender, RoutedEventArgs e)
{
    //Get the selected window.
    Window window = lbxWindows.SelectedItem as Window;

    if (window == null)
        return;

    if (cbxIsVisible.IsChecked == true)
        window.Show();
    else
        window.Hide();
}

private void btnBringToFront_Click(object sender, RoutedEventArgs e)
{
    Window window = lbxWindows.SelectedItem as Window;

    if (window != null)
        window.Activate();
}
```

```
        private void btnClose_Click(object sender, RoutedEventArgs e)
        {
            Window window = lbxWindows.SelectedItem as Window;

            if (window != null)
                window.Close();
        }
    }
}
```

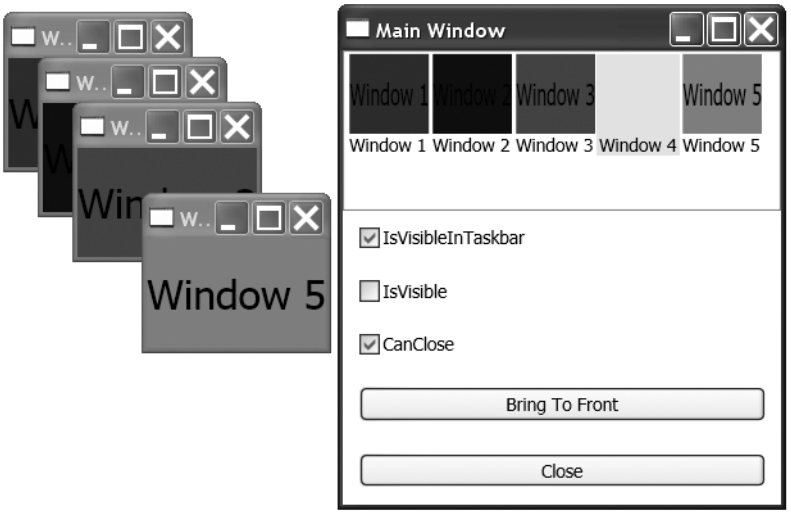

Figure 1-7. *Interacting with the windows in an application*

1-16. Debug Data Bindings Using an IValueConverter

Problem

You need to debug a binding that is not working as expected and want to make sure the correct values are going in.

Solution

Create a converter class that implements `System.Windows.Data.IValueConverter` (see Chapter 5) and returns the value it receives for conversion, setting a breakpoint or tracepoint within the converter.

How It Works

Debugging a data binding can sometimes be quite tricky and consume a lot of time. Because data bindings are generally defined in XAML, you don't have anywhere you can set a breakpoint to make sure things are working as you intended. In some cases, you will be able to place a breakpoint on a property of the object that is being bound, but that option isn't always available, such as when binding to a property of some other control in your application. This is where a converter comes in.

When using a simple converter that returns the argument being passed in, unchanged, you immediately have some code that you can place a breakpoint on or write some debugging information to the Output window or some log. This can tell you whether the value coming in is the wrong type, in a form that means it is not valid for the binding, or is coming in with a strange value. You'll also soon realize whether the binding is not being used, because the converter will never be hit.

The Code

The following example demonstrates a `System.Windows.Window` that contains a `System.Windows.Controls.Grid`. Inside the `Grid` are a `System.Windows.Controls.CheckBox` and a `System.Windows.Controls.Expander`. The `IsExpanded` property of the `Expander` is bound to the `IsChecked` property of the `CheckBox`. This is a very simple binding, but it gives an example where you are able to place a breakpoint in code.

```
<Window
  x:Class="Recipe_01_16.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:Recipe_01_16"
  Title="Recipe_01_13"
  Width="200"
  Height="200">
  <Window.Resources>
    <local:DummyConverter x:Key="DummyConverter" />
  </Window.Resources>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="0.5*" />
      <RowDefinition Height="0.5*"/>
    </Grid.RowDefinitions>

    <CheckBox
      x:Name="chkShouldItBeOpen"
      IsChecked="False"
      Content="Open Sesame!"
      Margin="10"
    />
```

```
    <Expander
      IsExpanded="{Binding
                    ElementName=chkShouldItBeOpen,
                    Path=IsChecked,
                    Converter={StaticResource DummyConverter}}"
      Grid.Row="1"
      Background="Black"
      Foreground="White"
      Margin="10"
      VerticalAlignment="Center"
      HorizontalAlignment="Center"
      Header="I'm an Expander!">
      <TextBlock Text="Sesame Open!" Foreground="White"/>
    </Expander>
  </Grid>
</Window>
```

The following code defines the code-behind for the previous XAML:

```
using System.Windows;

namespace Recipe_01_16
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

The following code defines a dummy converter class that returns the value passed to it:

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Recipe_01_16
{
    public class DummyConverter : IValueConverter
    {
        #region IValueConverter Members
```

```
        public object Convert(object value,
                              Type targetType,
                              object parameter,
                              CultureInfo culture)
        {
            return value;
        }

        public object ConvertBack(object value,
                                  Type targetType,
                                  object parameter,
                                  CultureInfo culture)
        {
            return value;
        }

        #endregion
    }
}
```

1-17. Debug Bindings Using Attached Properties

Problem

You need to debug a binding that is not working as expected and want to make sure the correct values are going in. Using a converter is either undesired or not feasible.

Solution

Use the `System.Diagnostics.PresentationTraceSources.TraceLevel` attached property defined in the `WindowsBase` assembly, setting the level of detail required. If the data binding is defined in code, use the static method `PresentationTraceLevel.SetTraceLevel`.

■ **Caution** Using the `PresentationTraceSources.TraceLevel` attached property can affect the performance of a WPF application and should be removed as soon as it is no longer required.

How It Works

The `PresentationTraceSources.TraceLevel` attached property allows you to specify the level of information written to the Output window for data bindings, on a per-binding basis. The higher the `System.Diagnostics.PresentationTraceLevel` value that is used, the more information that will be generated. The `PresentationTraceSources.TraceLevel` can be used on the following object types:

- System.Windows.Data.BindingBase
- System.Windows.Data.BindingExpressionBase
- System.Windows.Data.ObjectDataProvider
- System.Windows.Data.XmlDataProvider

It is important to remember to remove any trace-level attached properties from your code once you are finished debugging a binding; otherwise, your Output window will continue to be filled with binding information. Table 1-3 details the values of the PresentationTraceSource.TraceLevel enumeration.

Table 1-3. *Values for the PresentationTraceSources.TraceLevel*

Property	Description
None	Generates no additional information.
Low	Generates some information about binding failures. This generally details the target and source properties involved and any exception that is thrown. No information is generated for bindings that work properly.
Medium	Generates a medium amount of information about binding failures and a small amount of information for valid bindings. When a binding fails, information is generated for the source and target properties, some of the transformations that are applied to the value, any exceptions that occur, the final value of the binding, and some of the steps taken during the whole process. For valid bindings, information logging is light.
High	Generates the most amount of binding state information for binding failures and valid bindings. When a binding fails, a great deal of information about the binding process is logged, covering all the previous data in a more verbose manner.

The Code

The following markup demonstrates how to use the PresentationTraceSource.TraceLevel property in two different bindings. One of the bindings is valid and binds the value of the text block to the width of the parent grid; the other is invalid and attempts to bind the width of the parent grid to the height of the text block. Set the values of the PresentatonTraceSource.TraceLevel attached properties to see how they behave.

```
<Window
  x:Class="Recipe_01_17.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:diagnostics="clr-namespace:System.Diagnostics;assembly=WindowsBase"
  Title="Recipe_01_17"
  Height="300"
  Width="300">
```

```
  <Grid x:Name="gdLayoutRoot">
    <Viewbox>
      <TextBlock x:Name="tbkTextBlock">
        <TextBlock.Text>
          <Binding
            ElementName="gdLayoutRoot"
            Path="ActualWidth"
            diagnostics:PresentationTraceSources.TraceLevel="High"
          />
        </TextBlock.Text>
        <TextBlock.Height>
          <Binding
            ElementName="gdLayoutRoot"
            Path="Name"
            diagnostics:PresentationTraceSources.TraceLevel="High" />
        </TextBlock.Height>
      </TextBlock>
    </Viewbox>
  </Grid>
</Window>
```