 **Contents**

[Dependency Property Basics](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070006)

[Binding Sources and Targets](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070044)

[Custom Collections with Notifications](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070059)

[Dependency Properties and Events](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070065)

[Dynamic Binding Technique](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070077)

[The Freezable Difference](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070100)

[Using DependencyPropertyDescriptor](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070111)

[Collections of DependencyObjects](http://msdn.microsoft.com/en-us/magazine/cc794276.aspx#id0070128)

Objects these days seem to get bossed around a lot. Everybody wants to tell them what to do. A typical object in a Windows® Presentation Foundation (WPF) application can be bombarded by a variety of different requests: bind to this data, style yourself like this, inherit from your visual parent, and now entertain us with some animation.

How can objects establish boundaries and priorities? The WPF answer is a feature known as dependency properties. By providing a structured way for WPF classes to respond to changes that result from data bindings, styles, inheritance, and other sources, dependency properties have become as important as events and event handling were to an earlier generation of .NET programmers.

Yet there are some traditional tasks where dependency properties don't quite provide everything you might need. Suppose you have access to an object with dependency properties and you want to be notified when one of those properties changes. This sounds like a straightforward job for an event handler—until you realize that the event you need simply does not exist!

This problem becomes crucial when you're working with collections of objects. In some cases you might want to be informed of changes to particular dependency properties of the objects in this collection. The only existing solution is FreezableCollection<T>, which tells you when something has changed—but doesn't tell you what that something is.

In other words, dependency properties don't always play well with others. Compensating for their lack of notification events is what this column is all about.

**Dependency Property Basics**

Suppose you're designing a WPF class named PopArt and you want to define a property named SwirlyBrush of type Brush. If PopArt descends from DependencyObject, you can define SwirlyBrush as a DependencyProperty. The first step is a public static read-only field:

public static readonly DependencyProperty SwirlyBrushProperty;

The dependency property has the same name as the property, but with the word "Property" appended. As part of either the field declaration or the static constructor, you then register the dependency property:

SwirlyBrushProperty = DependencyProperty.Register("SwirlyBrush", typeof(Brush), typeof(PopArt), new PropertyMetadata(OnSwirlyBrushChanged));

You also need a regular property definition (sometimes called the CLR property) that provides normal access to this property:

public Brush SwirlyBrush { set { SetValue(SwirlyBrushProperty, value); } get { return (Brush) GetValue(SwirlyBrushProperty); } }

The SetValue and GetValue methods are defined by DependencyObject, which is why any class that defines dependency properties must derive from that class. The CLR property should not include any other code except the calls to these two methods. The CLR property is often said to be "backed" by the dependency property.

The OnSwirlyBrushChanged method referenced in the dependency property registration is a callback method that is invoked whenever the SwirlyBrush property changes. Because this method is associated with a static field, it must be a static method:

static void OnSwirlyBrushChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args) { ... }

The first argument is the particular instance of the class whose property is changing. If you've defined this dependency property in a class named PopArt, then the first argument is always an object of type PopArt. I like to define an instance method with the same name as the static method. The static method then just calls the instance method like this:

static void OnSwirlyBrushChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args) { (obj as PopArt).OnSwirlyBrushChanged(args); } void OnSwirlyBrushChanged(DependencyPropertyChangedEventArgs args) { ... }

The instance method includes whatever is needed to accommodate changes to the value of SwirlyBrush.

In code, you can set the value of the SwirlyBrush property in an instance of PopArt in the normal manner:

popart.SwirlyBrush = new SolidColorBrush(Colors.AliceBlue);

However, it's important to note that there exists a public static field in the PopArt class named PopArt.SwirlyBrushProperty. This field is a valid object of type DependencyProperty that exists independently of any object of type PopArt. This allows you to reference a particular property of a class prior to the creation of objects that have that property.

The SetValue method defined by DependencyObject is also public, so you can also set the SwirlyBrush property like this:

popart.SetValue(PopArt.SwirlyBrushProperty, new SolidColorBrush(Colors.AliceBlue));

What's more, if you have the three objects described in **Figure 1**, you can set that property with this completely generalized code:

target.SetValue(property, value);

 **Figure 1 Objects**

|  |  |
| --- | --- |
| Object | Description |
| target | An instance of a class that derives from DependencyObject. |
| property | An object of type DependencyProperty. |
| value | An object of the correct type of the dependency property. |

If the type of value does not agree with the type associated with the DependencyProperty (in my example, Brush), an exception will be raised. But the IsValidType method defined by Dependency­Property can help avoid problems like that.

The process of referencing and setting properties using Dependency­Property objects with SetValue and GetValue is cleaner than earlier methods of referring to properties with character strings such as "SwirlyBrush." Properties specified this way required reflection to actually set the property.

**Binding Sources and Targets**

The extensive use of dependency properties in WPF is not so evident when you set data bindings, styles, and animations in XAML, but it's obvious when you do these tasks in code. The SetBinding methods defined by BindingOperations and FrameworkElement require an object of type DependencyProperty as the binding target. The Setter class used in WPF styles requires DependencyProperty objects. The BeginAnimation method also requires a Dependency­Property object for the animation target.

These targets must all be DependencyProperty objects so that WPF can properly impose precedence rules. For example, property values set by an animation have precedence over values set by styles. (You can use the DependencyPropertyHelper class if you need to know which possible source was responsible for the value of a particular property.)

Although data-binding targets must be dependency properties, binding sources do not need to be dependency properties. Obviously the binding source must implement some kind of notification mechanism if the binding is to be successful in monitoring changes in the source, but WPF actually allows three different types of source notifications. These are mutually exclusive.

The first (and preferred) approach is to use dependency properties for both sources and targets. This is a particularly good solution when the same properties serve as both binding sources and targets in different contexts.

The second approach involves defining an event based on the property name. If a class defines a property named Flavor, for example, it would also define an event named FlavorChanged. As the name of the event suggests, the class fires this event whenever the value of Flavor changes. This type of notification was used extensively in Windows Forms (for example, the Enabled property defined by Control had a corresponding EnabledChanged event). But it should be avoided for WPF code, and I won't be discussing it further.

The final allowable approach is implementing the INotifyProperty­Changed interface, which requires the class to define an event named PropertyChanged based on the PropertyChangedEvent­Handler delegate. The associated PropertyChangedEventArgs object has a property named PropertyName that is a text string of the name of the property that's changed.

For example, if the class has a property named Flavor and a private field named flavor, the set accessor for the property would look something like this:

flavor = value; if (PropertyChanged != null) PropertyChanged(this, new PropertyChangedEventArgs("Flavor");

A class interested in changes to the Flavor property can simply subscribe to the PropertyChanged event and be notified when this property (or any other property in the class) changes.

The INotifyPropertyChanged interface is not foolproof. All the interface mandates is an event named PropertyChanged. There is no guarantee that the class will ever fire this event. In many cases, a class will fire the event for some of its public properties but not all. If you don't have access to the source code, there's no good way to tell beforehand which properties fire the PropertyChanged event and which do not.

Regardless, the INotifyPropertyChanged remains a good, simple solution for properties that will not be targets of WPF data bindings, styles, or animation, but that must provide change notifications to other classes.

**Custom Collections with Notifications**

It is sometimes necessary to have a collection of objects and to be notified whenever a property of one of these objects change. Most valuable would be a notification that tells you exactly which item in the collection changed and also which specific property of that item changed.

This is a fairly easy task when the objects in the collection implement the INotifyPropertyChanged interface and the collection itself implements INotifyCollectionChanged. A collection that implements this interface fires a CollectionChanged event whenever new items are added to the collection or existing items are removed. The CollectionChanged event provides lists of these added and removed items. Perhaps the most popular collection object that implements INotifyCollectionChanged is the generic ObservableCollection<T> class.

Let's create a custom collection class that derives from Observable­Collection<T> and implements a new event named ItemPropertyChanged. This event will be fired whenever a property of any item in the collection changes, and it will be accompanied by event arguments that include the item and the changed property.**Figure 2** shows the ItemPropertyChangedEventArgs class that derives from PropertyChangedEventArgs to include a new property named Item of type object. **Figure 2** also shows the ItemPropertyChanged­EventHandler delegate.

 **Figure 2 ItemPropertyChangedEventArgs**

public class ItemPropertyChangedEventArgs : PropertyChangedEventArgs { object item; public ItemPropertyChangedEventArgs(object item, string propertyName) : base(propertyName) { this.item = item; } public object Item { get { return item; } } } public delegate void ItemPropertyChangedEventHandler(object sender, ItemPropertyChangedEventArgs args);

The ObservableNotifiableCollection<T> that derives from ObservableCollection<T> and defines the ItemPropertyChanged event is shown in **Figure 3**. Notice that the new class constrains the type parameter to objects of type INotifyPropertyChanged. During the OnCollectionChanged override, the class registers a PropertyChanged event handler for each item being added to the collection and removes it when an item is removed from the collection. During the item's PropertyChanged event, the collection fires an ItemPropertyChanged event. The sender object of the Property­Changed event (which is the item whose property is changing) becomes the Item property of the ItemPropertyChangedEventArgs.

 **Figure 3 The ObservableNotifiableCollection Class**

class ObservableNotifiableCollection<T> : ObservableCollection<T> where T : INotifyPropertyChanged { public ItemPropertyChangedEventHandler ItemPropertyChanged; protected override void OnCollectionChanged( NotifyCollectionChangedEventArgs args) { base.OnCollectionChanged(args); if (args.NewItems != null) foreach (INotifyPropertyChanged item in args.NewItems) item.PropertyChanged += OnItemPropertyChanged; if (args.OldItems != null) foreach (INotifyPropertyChanged item in args.OldItems) item.PropertyChanged -= OnItemPropertyChanged; } void OnItemPropertyChanged(object sender, PropertyChangedEventArgs args) { if (ItemPropertyChanged != null) ItemPropertyChanged(this, new ItemPropertyChangedEventArgs(sender, args.PropertyName)); } }

The code shown inFigures **2** and **3** are part of the ObservableNotifiableCollectionDemo project included in the downloadable code for this column. The project also includes a small demonstration program.

**Dependency Properties and Events**

Although dependency properties make fine binding sources, they do not provide a general notification mechanism in the form of a public event. This annoying fact usually comes as a bit of a shock the first time you go searching for such an event.

However, DependencyObject does define a protected OnProperty­Changed method. This method is fundamental to the way ­DependencyObject works. Judging from my experimentation, OnPropertyChanged is called as a result of calls to SetValue, and it is OnPropertyChanged that invokes the callback methods associated with the individual dependency properties. The documentation of OnPropertyChanged contains plenty of warnings about the dangers of overriding this method.

If you are writing a class that implements dependency properties, you might anticipate that a user of the class will want events to be fired whenever particular dependency properties change. In this case, you can explicitly provide these events. Give the event the name of the property with the word "Changed" appended (similar to the way events were named in Windows Forms), but define the event using the DependencyPropertyChangedEventHandler delegate. As models, you'll find many events in UIElement that are associated with dependency properties, including FocusableChanged and IsEnabledChanged.

For the PopArt example I showed earlier, you can define a SwirlyBrushChanged event like this:

public event DependencyPropertyChangedEventHandler SwirlyBrushChanged;

During the instance version of the OnSwirlyBrushChanged method I showed earlier, fire the event like so:

if (SwirlyBrushChanged != null) SwirlyBrushChanged(this, args);

This code is so astonishingly trivial that it's surprising more dependency properties don't have associated events.

Unfortunately, if the author of the class you need to use has not kindly provided the events you need, you must find another way to receive notifications when a particular dependency property changes. Happily, there are at least three solutions: one strange, one sledgehammer, and one from left field.

**Dynamic Binding Technique**

The strange approach to receiving notifications of changes in dependency properties involves dynamically creating both a dependency property and data binding. If you're writing a class that descends from DependencyObject, have access to an object that contains dependency properties, and need to be notified when one of those dependency properties changes, you can create a dependency property and a binding on the fly to give you the information you need.

For example, suppose your class has access to an object of type TextBlock named txtblk and you want to know when the Text property changes. The Text property is backed by a Dependency­Property named TextProperty, but TextBlock doesn't define a TextChanged event. To compensate for this, you first register a Dependency­Property of the same type as the property you want to monitor:

DependencyProperty MyTextProperty = DependencyProperty.Register("MyText", typeof(string), GetType(), new PropertyMetadata(OnMyTextChanged));

This DependencyProperty object does not need to be a public static field. As demonstrated here, it can be registered inside an instance method. The OnMyTextChanged callback can also be an instance method.

After you've registered the DependencyProperty, you can define a binding from the TextBlock object to this property:

Binding binding = new Binding("Text"); binding.Source = txtblk; BindingOperations.SetBinding(this, MyTextProperty, binding);

Any changes to the Text property of the txtblk object will now call your OnMyTextChanged callback.

You can even register this DependencyProperty and define the callback without knowing anything about the Dependency­Object and DependencyProperty you're monitoring. Suppose you only know you need to keep track of a DependencyProperty variable named property in a DependencyObject derivative named obj. Here's the code:

DependencyProperty OnTheFlyProperty = DependencyProperty.Register("OnTheFly", property.PropertyType, GetType(), new PropertyMetadata(OnTheFlyChanged)); Binding binding = new Binding(property.Name); binding.Source = obj; BindingOperations.SetBinding(this, OnTheFlyProperty, binding);

This solution is best for one or more properties of a single object. Once you start dealing with multiple objects of the same type (such as in a collection), you need to create unique on-the-fly dependency properties and bindings for each property of each object. The whole scheme quickly gets out of hand.

**The Freezable Difference**

The Freezable class is the sledgehammer approach to receiving notifications about changes to dependency objects. It certainly does the job, but you can't actually figure out what it is that the sledgehammer clobbered.

Freezable overrides the OnPropertyChanged method defined by DependencyObject and defines a new property named Changed that is documented as being fired whenever "this Freezable or any object it contains is modified" (emphasis added). That's a bold statement, and it is indeed true as long as "any object it contains" is understood as sub-properties of type Freezable, which are backed by dependency properties.

For example, suppose that you have a Freezable class named A with a property named B of type B. The B class also derives from Freezable and has a property named C of type C. The C class also derives from Freezable and has a property named D of type double. All these properties are backed by dependency properties. Create objects of all these types; change property D; and classes A, B, and C all fire Changed events. (Note that the NestedFreezableNotifications­Demo project included with the source code download for this column demonstrates this.)

The big problem is that this Changed event is based on the EventHandler delegate and provides no information at all about exactly which property (or sub-property) in the Freezable object has changed. There is also a lot of overhead in these notifications, and that's why the Freezable class is named for its ability to become unchangeable and make all the notifications disappear.

Freezable is used extensively in the WPF graphics system. The Brush, Pen, Geometry, and Drawing classes all derive from Freezable. Several collections derive from Freezable, including Double­Collection, PointCollection, VectorCollection, and Int32Collection. These collections also implement Changed events that are fired when the collection changes. Other Freezable collections that are collections of Freezable objects (such as GeometryCollection and TransformCollection) fire the Changed event whenever any property or sub-property of an item in the collection changes.

For example, PathGeometry derives from Freezable. It contains a Figures property of type PathFigureCollection, which also derives from Freezable. The PathFigureCollection contains objects of type PathFigure, which also derives from Freezable. PathFigure contains a property named Segments of type PathSegment­Collection, also a Freezable. PathSegment derives from Freezable and is the base class for classes like PolyLineSegment and PolyBezier­Segment. These classes contain properties named Points of type Point­Collection, which also derives from Freezable.

The overall effect: when any individual point is changed, change notifications go rippling up the object hierarchy and cause the graphical object to be entirely redrawn.

Does PathGeometry attempt to figure out exactly which point has changed and fix only that part of the geometry? In other words, does it perform an incremental update? Judging by the information it has available, it simply cannot. All that PathGeometry knows is that something has changed and it's time to start over from scratch.

If this is what you want—a simple notification that something has changed somewhere in a collection of nested sub-properties—the Freezable class may be ideal. There is also a generic FreezableCollection<T> that is useful. The type parameter doesn't have to be a Freezable type. It merely needs to be a DependencyObject. But if it's a Freezable type, the collection will fire Changed events for changes in the properties or Freezable sub-properties of the collection items.

Keep in mind that there are certain restrictions with Freezable classes. If you derive a class from FreezableCollection<T>, you must override CreateInstanceCore (and simply call the class's constructor and return that object) or you'll get strange errors that will drive you batty.

**Using DependencyPropertyDescriptor**

A third technique to get notification events from dependency properties involves the DependencyPropertyDescriptor class. It's certainly not an obvious solution because it's documented as being "primarily used by designer applications," but it works and that is what's important.

Suppose again you have an object of type TextBlock and you want to know when the Text property changes. You first create an object of type DependencyPropertyDescriptor:

DepedencyPropertyDescriptor descriptor = DependencyPropertyDescriptor.FromProperty( TextBlock.TextProperty, typeof(TextBlock));

The first argument is the DependencyProperty object and the second is the class that owns that dependency property or inherits it. Notice that the DependencyPropertyDescriptor is not associated with any particular TextBlock object.

You are then able to register a handler to detect changes in the Text property for a particular TextBlock object (named txtblk, for example):

descriptor.AddValueChanged(txtblk, OnTextChanged);

Remember that there is also a RemoveValueChanged method to remove the handler.

The OnTextChanged handler is based on the simple Event­Handler delegate signature and will probably be defined something like this:

void OnTextChanged(object sender, EventArgs args) { ... }

Note that the EventArgs argument doesn't provide any information. However, the first argument to the handler is the TextBlock object whose Text value has changed.

You won't want to share these handlers among multiple dependency properties—you'll want a separate handler for each. But you can easily share these handlers among multiple objects because you can distinguish the objects by the first argument to the handler.

There is now sufficient information present to allow the final step, which is to build a collection that fires notifications when dependency properties of items in the collection change.

**Collections of DependencyObjects**

In theory, if you have a collection of objects of a type that derives from DependencyObject, you should be able to create Dependency­PropertyDescriptor objects for particular dependency properties that allow you to be notified whenever these properties change on any item in the collection.

One obstacle involves the method that handles notifications from DependencyPropertyDescriptor. You won't want to share one method among multiple dependency properties because you can't tell which dependency property has changed. You'll want separate methods for each dependency property. In general, the number of methods you'll need won't be available until run time. You can create methods dynamically at run time, but that involves generating intermediate language. It's easier defining a class with a method specifically for handling these notifications and then to create an instance of this class for each dependency property you want to monitor.

I created the Observable­Depend­encyObjectCollection<T> collection class to do this, and it derives from ObservableCollection<T>. Note that the items in this collection must be instances of a class that derives from DependencyObject. The collection class creates DependencyPropertyDescriptor objects for each Dependency­Property defined or inherited by that type parameter or, alternatively, for a selected list of Dependency­Property objects.

ObservableDependencyObjectCollection<T> defines a new event named ItemDependencyPropertyChanged.**Figure 4** shows the definition of the event arguments and the delegate for this event. ItemDependencyPropertyChangedEventArgs is similar to the normal DependencyPropertyChangedEventArgs except that it includes an Item property and no OldValue property.

 **Figure 4 ItemDependencyPropertyChangedEventArgs**

public struct ItemDependencyPropertyChangedEventArgs { DependencyObject item; DependencyProperty property; object newValue; public ItemDependencyPropertyChangedEventArgs( DependencyObject item, DependencyProperty property, object newValue) { this.item = item; this.property = property; this.newValue = newValue; } public DependencyObject Item { get { return item; } } public DependencyProperty Property { get { return property; } } public object NewValue { get { return newValue; } } } public delegate void ItemDependencyPropertyChangedEventHandler( Object sender, ItemDependencyPropertyChangedEventArgs args);

I will show you the actual ObservableDependencyObject­Col­lection<T> class in three installments. The first part of the class is shown in **Figure 5**. Although the class derives from ­Ob­serv­ableCollection<T>, it constrains the type parameter to ­De­pend­encyObject. ObservableCollection<T> has two constructors: one parameterless and one with a generic List<T> object for initializing the contents. The new class includes those two constructors, but it adds two more that additionally let you specify a list of Dependency­Property objects you want to monitor.

For example, suppose you want a collection of Button objects, but you only want to be notified when two font-related properties change. Here's the constructor:

ObservableDependencyObjectCollection<Button> buttons = new ObservableDependencyObjectCollection( Button.FontSize, Button.FontFamily);

You can then install a handler for the collection:

buttons.ItemDependencyPropertyChanged += OnItemDependencyPropertyChanged;

All the constructors in **Figure 5** end up calling either GetExplicit­DependencyProperties or GetAllDependencyProperties. The first method just calls CreateDescriptor for each of the Dependency­Property objects listed in the constructor arguments. The second method uses reflection to obtain all the public fields of type Depend­encyProperty defined by both the type parameter and its ancestors (that's what BindingFlags.FlattenHierarchy means) and then calls CreateDescriptor for each.

 **Figure 5 The ObservableDependencyObjectCollection Constructors**

public class ObservableDependencyObjectCollection<T> : ObservableCollection<T> where T : DependencyObject { public event ItemDependencyPropertyChangedEventHandler ItemDependencyPropertyChanged; List<DescriptorWrapper> descriptors = new List<DescriptorWrapper>(); public ObservableDependencyObjectCollection() { GetAllDependencyProperties(); } public ObservableDependencyObjectCollection(List<T> list) : base(list) { GetAllDependencyProperties(); } public ObservableDependencyObjectCollection( params DependencyProperty[] properties) { GetExplicitDependencyProperties(properties); } public ObservableDependencyObjectCollection(List<T> list, params DependencyProperty[] properties) : base(list) { GetExplicitDependencyProperties(properties); } void GetExplicitDependencyProperties( params DependencyProperty[] properties) { foreach (DependencyProperty property in properties) CreateDescriptor(property); } void GetAllDependencyProperties() { FieldInfo[] fieldInfos = typeof(T).GetFields(BindingFlags.Public | BindingFlags.Static | BindingFlags.FlattenHierarchy); foreach (FieldInfo fieldInfo in fieldInfos) if (fieldInfo.FieldType == typeof(DependencyProperty)) CreateDescriptor(fieldInfo.GetValue(null) as DependencyProperty); } void CreateDescriptor(DependencyProperty property) { DescriptorWrapper descriptor = new DescriptorWrapper(typeof(T), property, OnItemDependencyPropertyChanged); descriptors.Add(descriptor); }

Each call to the CreateDescriptor method creates a new object of type DescriptorWrapper, passing to it the type of the items in the collection, the DependencyProperty to be monitored, and a callback method named OnItemDependency­PropertyChanged.

The DescriptorWrapper class that is shown in **Figure 6** is internal to ObservableDependencyObjectCollection<T> and essentially wraps a DependencyPropertyDescriptor object. The constructor saves the particular DependencyProperty associated with this ­DependencyPropertyDescriptor and the callback method.

 **Figure 6 The DescriptorWrapper Class**

class DescriptorWrapper { DependencyPropertyChangedEventHandler OnItemDependencyPropertyChanged; DependencyPropertyDescriptor desc; DependencyProperty dependencyProperty; public DescriptorWrapper(Type targetType, DependencyProperty dependencyProperty, DependencyPropertyChangedEventHandler OnItemDependencyPropertyChanged) { desc = DependencyPropertyDescriptor.FromProperty( dependencyProperty, targetType); this.dependencyProperty = dependencyProperty; this.OnItemDependencyPropertyChanged = OnItemDependencyPropertyChanged; } public void AddValueChanged(DependencyObject dependencyObject) { desc.AddValueChanged(dependencyObject, OnValueChanged); } public void RemoveValueChanged(DependencyObject dependencyObject) { desc.RemoveValueChanged(dependencyObject, OnValueChanged); } void OnValueChanged(object sender, EventArgs args) { OnItemDependencyPropertyChanged(sender, new DependencyPropertyChangedEventArgs( dependencyProperty, null, (sender as DependencyObject).GetValue( dependencyProperty))); } }

The class also includes two methods named AddValueChanged and Remove­ValueChanged that call the corresponding methods in the DependencyPropertyDescriptor object. The OnValueChanged handler funnels the information back to the collection class.

The final section of the ObservableDependencyObject­Collection<T>­ class is shown in **Figure 7**. The override of the OnCollectionChanged method is responsible for looping through all the items being added to the collection (and each DescriptorWrapper)­ and calling AddValueChanged. The handlers are removed for every item removed from the collection.

 **Figure 7 ObservableDependencyObjectCollection**

protected override void OnCollectionChanged( NotifyCollectionChangedEventArgs args) { base.OnCollectionChanged(args); if (args.NewItems != null) foreach (DependencyObject obj in args.NewItems) foreach (DescriptorWrapper descriptor in descriptors) descriptor.AddValueChanged(obj); if (args.OldItems != null) foreach (DependencyObject obj in args.OldItems) foreach (DescriptorWrapper descriptor in descriptors) descriptor.RemoveValueChanged(obj); } protected void OnItemDependencyPropertyChanged(object item, DependencyPropertyChangedEventArgs args) { if (ItemDependencyPropertyChanged != null) ItemDependencyPropertyChanged(this, new ItemDependencyPropertyChangedEventArgs( item as DependencyObject, args.Property, args.NewValue)); }

The OnItemDependencyPropertyChanged method that concludes the ObservableDependencyObjectCollection<T> class is called from DescriptorWrapper and fires the ItemDependencyProperty­Changed event, which is the purpose of this entire exercise.

The ObservableDependencyObjectCollection<T> class is structured so that it creates all the DescriptorWrapper objects it needs at the time the collection is constructed based on the type parameter. If you specify this type parameter as DependencyObject and use the parameterless constructor, no DescriptorWrapper objects will be created because DependencyObject defines no dependency properties of its own. However, you can specify the type parameter as DependencyObject and include an explicit list of DependencyProperty objects and the notifications will work.

In a more realistic scenario, you might want to set the type parameter of ObservableDependencyObjectCollection<T> to Frame­workElement and then fill the collection with all sorts of derivatives of FrameworkElement, such as TextBlock and Button. The collection will only report changes in dependency properties defined by UIElement and FrameworkElement and not those defined by derivative classes.

For more flexibility, the collection must be written to create new DescriptorWrapper objects as items are added, based on the types of those items and the dependency properties defined by those types. You don't want to create duplicate DescriptorWrapper objects, so you'll need to check whether a DescriptorWrapper has already been created for each particular dependency property first.

As items are removed from the collection, you will want to abandon DescriptorWrapper objects that are no longer needed. This implies that a usage count should be attached to each DescriptorWrapper, and a certain DescriptorWrapper should be removed from the descriptors collection in ObservableDependencyObject­Collection­<T> when the usage count drops to zero.

It would also be possible to create a collection class that is completely flexible and that monitors the properties of all its items, regardless of whether the items define dependency properties or implement the INotifyPropertyChanged interface.

As you can see, treated properly, even dependency properties can be taught to generate events just like their older predecessors in the Microsoft® .NET Framework.