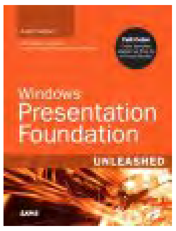
Lectures on WPF
Dependency Properties



This lecture is based on this book of Adam Nathan and Daniel Lehenbauer: Chapter 3

Why Dependency Properties ?

Classes are made of fields, properties, methods, and events. Fields are like variables in that they can be read or set directly. For example, if you have an class named "`Car`" you could store its color in a field named "`Color`": `myCar.Color="White"`.
Properties are retrieved and set like fields, but are implemented using `PropertySet` and `PropertyGet` procedures, which provide more control on how values are set or returned.
WPF replaced 90% of both fields and properties with much more complicated objects,
the **Dependency Properties**.
What was good with the good old fields and properties ? They have been so simple !
What was wrong with the simple good old fields and properties ?
1) Whenever a value changes, they can't inform their relatives.
2) Whenever a value change has effects on the surface, they can't inform the render engine.
3) They are unable to propagate their values down the element tree.
4) They cannot take over values from higher levels of the element tree.
5) There is no way to attach nor detach fields and properties at run-time.
6) Each object wastes memory by instancing its own complete set of fields and properties instead of sharing them with other elements in the element tree.

One of the primary architectural philosophies used in building WPF was
a preference for properties over methods or events.
Properties are declarative and allow you to more easily specify intent instead of action.
This also supported a model driven system for displaying visual content. In order to have more of the system driven by properties, a richer property system than what the CLR provides was needed.

WPF uses both:
a) the traditional properties of the CLR.
b) the new dependency properties which expose themselves disguised as CLR properties.
At a basic level, you interact with them directly and never know that they are implemented as a dependency property objects. They can be used as simply as the good old ones: `myCar.Color="White"` but in the backstage they are tuned with metadata, validation, coercion, sharing, up and down propagation, CallBack-functions, data binding and triggers.

`DependencyObjects` (i.e. all `Visuals`) have an incredible number of properties. Samples:
`Label` has 18 traditional and 71 dependency properties.
`Button` has 18 traditional and 78 dependency properties.
The key to WPF's declarative-friendly design is its heavy use of properties.
Properties can be easily set in XAML (directly or by a design tool).

The biggest feature of a dependency property is its built-in ability to provide change notification to all interested neighbors.
Additionally they enable styling, automatic data binding, animation, and more.
And they are the basis of all the magic automatic re-rendering of WPF-generated surfaces whenever somewhere on the surface a property like `TextElement.FontSizeProperty` may change.

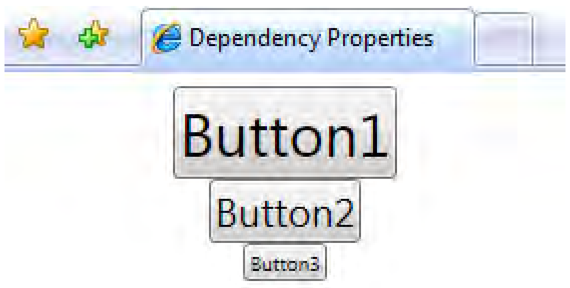
The property system also provides for sparse storage of property values.
Because objects can have hundreds of properties, and most of the values are in their default state (inherited, set by styles, etc.), not every instance of an object needs to have the full weight of every property defined on it.

Understanding most of the nuances of dependency properties is usually only important for custom control authors. However, even casual users of WPF end up needing to be aware of what dependency properties are and how they work.
See: MSDN: Dependency Properties Overview
See: MSDN: WPF Architecture

Copy the following code into a text editor and store the text file to: `C:\temp\FontSize.xaml`

```
<Page
xmlns="http://schemas.microsoft.com/winfx/2006
             /xaml/presentation"
 WindowTitle="Dependency Properties"
 WindowWidth="300" WindowHeight="400"
 FontSize="30">
 <StackPanel VerticalAlignment="Center">
  <Button HorizontalAlignment="Center"
          Content="Button1"/>
  <StackPanel TextBlock.FontSize="20">
   <Button HorizontalAlignment="Center"
           Content="Button2"/>
   <Button HorizontalAlignment="Center"
           Content="Button3" FontSize="10"/>
  </StackPanel>
 </StackPanel>
</Page>
```

Start the Internet Explorer and open `C:\temp\FontSize.xaml` or drag and drop `C:\temp\FontSize.xaml` onto the Internet Explorer icon of your screen. The browser will display three useless Buttons. The two nested StackPanels remain invisible.



Experiments varying the three Dependency Properties `FontSize` of `C:\temp\FontSize.xaml`:
(Restore the original code after any experiment.)

1	Change `FontSize="30"` of `Page` to `FontSize="15"`	Button1 shrinks because it inherits its `FontSize` from `Page`.
2	Change `TextBlock.FontSize="20"` of the inner `StackPanel` to `TextBlock.FontSize="30"`	Button2 grows because it inherits the `TextBlock.FontSize` from the inner `StackPanel`.
3	Change `FontSize="10"` of Button3 to `FontSize="30"`	Button3 grows because it carries its own `FontSize`.
4	Remove `FontSize="10"` from Button3.	Button3 inherits the same `TextBlock.FontSize` as Button2 from the inner `StackPanel`.
5	Remove `FontSize="10"` from Button3 and insert it into Button2.	Button2 shrinks and Button3 inherits its `FontSize` from the inner `StackPanel`.
6	Remove `FontSize="10"` from Button2 and `TextBlock.FontSize="20"` from the inner `StackPanel`.	All Buttons inherit their `FontSize` from `Page`.
7	Remove all `FontSizes` from Button3, from the inner `StackPanel` and from `Page`.	All Buttons inherit their `FontSize` from the default Button Template of WPF (and from the current Vista-theme).

Summary: WPF cares about us in many ways.
1. It provides a default `FontSize` for lazy programmers (Experiment 7).
2. It provides a property inheritance sytem where the property is propagated down the object tree.
3. It adjusts automatically further properties as `Button.Width` and `Button.Height` to the string length, to the `Font` and to its `FontSize`.

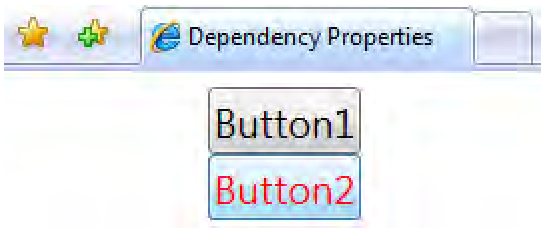
Change Notification by Triggers

Whenever the value of a dependency property changes, WPF can trigger actions.
Such actions can re-render the appropriate elements, update the current layout, refresh data bindings, and much more. One of the most interesting features is property triggers, which enable you to perform actions when a property value changes. E.g. to turn the text red when the mouse pointer hovers over it.

Copy the following code into a text editor and store the text file to: `C:\temp\Trigger.xaml`

```
<Page
 xmlns="http://schemas.microsoft.com/winfx
            /2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com
              /winfx/2006/xaml"
 WindowTitle="Dependency Properties"
 WindowWidth="300" WindowHeight="400"
 FontSize="20">
 <Page.Resources>
  <Style TargetType="{x:Type Button}">
   <Setter Property=
    "HorizontalAlignment" Value="Center"/>
   <Style.Triggers>
    <Trigger Property="IsMouseOver"
             Value="True">
     <Setter Property="Foreground"
             Value="Red"/>
    </Trigger>
   </Style.Triggers>
  </Style>
 </Page.Resources>
 <StackPanel VerticalAlignment="Center">
  <Button Content="Button1"/>
  <Button Content="Button2"/>
 </StackPanel>
</Page>
```

Start the Internet Explorer and open `C:\temp\Trigger.xaml` or drag and drop `C:\temp\Trigger.xaml` onto the Internet Explorer icon of your screen.



This trigger acts upon Button's `IsMouseOver` property, which becomes `True` at the same time the `MouseEnter` event is raised and `False` at the same time the `MouseLeave` event is raised. There is no need to worry about reverting `Foreground` to black when `IsMouseOver` changes to `False`. This is automatically done by WPF!
Property triggers are just one of three types of triggers. A data trigger is a form of property trigger that works for all .NET properties (not just dependency properties). An event trigger invokes animations or sounds.

Property Inheritance

The term property inheritance (more precise: property value inheritance) doesn't refer to traditional object oriented class-based inheritance, but rather the flowing of property values down the element tree.
A simple example of property inheritance can be seen in paragraph **Why Dependency Properties ?**.

Important: Not every dependency property participates in property value inheritance.
Internally, dependency properties can opt in to inheritance or they can refuse it.
A few controls such as `StatusBar`, `Menu`, and `ToolTip` internally set their font properties to match current system `Control Panel` settings.
The result can be confusing, however, because such controls end up "swallowing" any inheritance from proceeding further down the element tree. For example, if you add a `Button` as a logical child of the `StatusBar`, its `FontSize` and `FontStyle` would be the default values of `12` and `Normal`, respectively, unlike the other Buttons outside of the `StatusBar`.

Multiple Providers

An object 1 which influences a property p of another object 2 is called a "Property Provider" or shortly "provider" of property p of object 2. Object 2 can have and normally has a lot of providers for each of its properties.
A well-defined mechanism for handling these disparate property value providers prevents from chaos.
Providers a are classified in 8 classes of precedence:

Priority	Provider	Source
1 = highest	local value	From inside the element's definition tag
2	Style triggers	From the element's Style
3	Template triggers	From the element's Template
4	Style setters	From the element's Style
5	Theme Style triggers	From the current Vista-theme
6	Theme Style setters	From the current Vista-theme
7	Property value inheritance	Coming down from higher elements of the element tree
8	Default value	Initial value from the element's Template

Sample: The setting of `StatusBar`'s font properties is done via theme style setters.
Although this has precedence over property value inheritance, you can still override these font settings using any mechanism with a higher precedence, such as simply setting local values on the `StatusBar`.

Clearing the local value:
The earlier "Change Notification by Triggers" section demonstrated the use of procedural code to change a `Button's Foreground` to blue in response to the `MouseEnter` event,
and then changing it back to black in response to the `MouseLeave` event.
The problem with this approach is that black is set as a local value inside `MouseLeave`, which is much different from the Button's initial state in which its black `Foreground` comes from a setter in its theme style.
If the theme is changed and the new theme tries to change the default Foreground color (or if other providers with higher precedence try to do the same), it gets trumped by the local setting of black.
What you likely want to do instead is clear the local value and let WPF set the value from the relevant provider with the next-highest precedence.
Fortunately, `DependencyObject` provides exactly this kind of mechanism with its `ClearValue` method.
This can be called on a Button `b` as follows in C#:

```
b.ClearValue( Button.ForegroundProperty );
```

Attached Properties

An attached property is a special form of dependency property that can be attached to arbitrary objects.
This may sound strange at first.
We already used this feature in our first sample in **Why Dependency Properties ?**.
We set `TextElement.FontSize` on the inner StackPanel so it is inherited only by the two lower buttons.
Moving the property `FontSize` to the inner `StackPanel` element doesn't work, however,
because `StackPanel` doesn't have any font-related properties of its own!
Instead, we must use the `FontSize` attached property that happens to be defined on a class called `TextElement`.
The most confusing part about the `FontSize` attached property is that it isn't defined by `Button` or even `Control`, the base class that defines the normal `FontSize` dependency property!
Instead, it is defined by the seemingly unrelated `TextElement` class (and also by the `TextBlock` class, which could also be used).
How can this possibly work when `TextElement.FontSizeProperty` is a
separate `DependencyProperty` field from `Control.FontSizeProperty`
(and `TextElement.FontStyleProperty` is separate from `Control.FontStyleProperty`)?
The key is that Control internally borrows its `FontSize` property from `TextElement` and
exposes this foreign property as its own.
Advantage: Setting an attached property once in a layout-control is much simplier than to set it within each of its children.

Dependency Objects

It is much more complicated to define a new dependency property than to define a normal property, because dependency properties have to be registered within WPF's **Property Engine**.
Most developers will never define a new dependency property.
The pre-defined controls contain enough of them.
However it is important to keep in mind how different they are compared to normal properties.
Sample of how to define a (superfluous) normal property:

```
class myButton : Button
{ public double myFontSize; //normal property
}
```

A `DependencyProperty` must register a `DependencyObject` which has to contain:
1. `Metdata` containing a default value (in the sample: `11.0`), and
   some boolean flags for property inheritance and rendering,
2. a `get` and `set` method enabling to access it in the same way as a normal property.

Sample of how to define a (superfluous) dependency property:

```
class myButton : Button
{ public class myFontSize : DependencyObject
  { public static readonly DependencyProperty myFontSizeProperty = DependencyProperty.Register
    ( "myFontSize", typeof(double), typeof(Button),
       new FrameworkPropertyMetadata( 11.0,
                                      FrameworkPropertyMetadataOptions.Inherits |
                                      FrameworkPropertyMetadataOptions.AffectsRender )
    );
  public double myFontSize
  { get { return (double)GetValue( myFontSizeProperty ); }
    set {                SetValue( myFontSizeProperty, value ); }
  }
}
```