* Static vs Dynamic resource+

1. Static Resource - Static resources are the resources which you cannot manipulate at runtime. The static resources are evaluated only once by the element which refers them during the loading of XAML.

2. Dynamic Resource - Dynamic resources are the resources which you can manipulate at runtime and are evaluated at runtime. If your code behind changes the resource, the elements referring resources as dynamic resources will also change.

Using Static Resources improves the performance of your WPF application. When you are using Dynamic Resources, make sure that the Element Properties must be Dependency Property. This behavior is because, DPs have Change Notification support implicitly implemented, which in turn updates your UI.

Еще одно различие между статическими и динамическими ресурсами касается поиска системой нужного ресурса. Так, при определении статических ресурсов ресурсы элемента применяются только к вложенным элементам, но не к внешним контейнерам. Например, ресурс кнопки мы не можем использовать для грида, а только для тех элементов, которые будут внутри этой кнопки. Поэтому, как правило, большинство ресурсов определяются в коллекции Window.Resources в качестве ресурсов всего окна, чтобы они были доступны для любого элемента данного окна.

В случае с динамическими ресурсами такого ограничения нет.

<Window.Resources>

<ResourceDictionary Source="Dictionary1.xaml" />

</Window.Resources>

StaticResource uses first value. DynamicResource uses last value.

DynamicResource can be used for nested styling, StaticResource cannot.

In a composite WPF scenario, your user control can make use of resources defined in any other parent window/control (that is going to host this user control) by referring to that resource as DynamicResource.

* DataTemplate vs ControlTemplate+

Control template – шаблон контрола, устанавливается через свойство Template любого UI элемента

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

<Grid>

<Ellipse Fill="{TemplateBinding Background}"/>

<ContentPresenter HorizontalAlignment="Center"

VerticalAlignment="Center"/>

</Grid>

</ControlTemplate>

</Setter.Value>

DataTemplate – шаблон для элементов списка. Устанавливается через ItemTemplate свойство списков

The DataTemplate class has a DataType property that is very similar to the TargetType property of the Style class. Therefore, instead of specifying an x:Key for the DataTemplate in the above example

<ListBox Width="400" Margin="10"

ItemsSource="{Binding Source={StaticResource myTodoList}}">

<ListBox.ItemTemplate>

<DataTemplate>

<StackPanel>

<TextBlock Text="{Binding Path=TaskName}" />

<TextBlock Text="{Binding Path=Description}"/>

<TextBlock Text="{Binding Path=Priority}"/>

</StackPanel>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

* DataTemplateSelector+

To supply logic to choose which DataTemplate to use based on the Priority value of the data object, create a subclass of DataTemplateSelector and override the SelectTemplate method. In the following example, the SelectTemplate method provides logic to return the appropriate template based on the value of the Priority property. The template to return is found in the resources of the enveloping Window element.

To use the template selector resource, assign it to the ItemTemplateSelector property of the ListBox. The ListBox calls the SelectTemplate method of the TaskListDataTemplateSelector for each of the items in the underlying collection. The call passes the data object as the item parameter. The DataTemplate that is returned by the method is then applied to that data object.

Typically, you create a DataTemplateSelector when you have more than one DataTemplate for the same type of objects and you want to supply your own logic to choose a DataTemplate to apply based on the properties of each data object. Note that if you have objects of different types you can set the DataType property on the DataTemplate. If you do that then there is no need to create a DataTemplateSelector. Furthermore, if you have objects of the same type but with different properties, you can also consider using a DataTrigger or a data converter.

ItemTemplateSelector="{StaticResource myDataTemplateSelector}"

public class TaskListDataTemplateSelector : DataTemplateSelector

{

public override DataTemplate

SelectTemplate(object item, DependencyObject container)

{

FrameworkElement element = ***container*** as FrameworkElement;

if (element != null && item != null && item is Task)

{

Task taskitem = item as Task;

if (taskitem.Priority == 1)

return

element.FindResource("importantTaskTemplate") as DataTemplate;

else

return

element.FindResource("myTaskTemplate") as DataTemplate;

}

return null;

}

}

* Style+

Стиль создается как ресурс с помощью объекта Style, который представляет класс System.Windows.Style. И как любой другой ресурс, он обязательно должен иметь ключ. С помощью коллекции Setters определяется группа свойств, входящих в стиль. В нее входят объекты Setter, которые имеют следующие свойства:

Property: указывает на свойство, к которому будет применять данный сеттер. Имеет следующий синтаксис: Property="Тип\_элемента.Свойство\_элемента". Выше в качестве типа элемента использовался Control, как общий для всех элементво. Поэтому данный стиль мы могли бы применить и к Button, и к TextBlock, и к другим элементам. Однако мы можем и конкретизировать элемент, например, Button:

<Setter Property="Button.FontFamily" Value="Arial" />

Value: устанавливает значение

<Style TargetType="Button">

<Style x:Key="ButtonChildStyle" BasedOn="{StaticResource ButtonParentStyle}">

* Behavior

Behavior. В WPF есть абстрактный шаблонный класс Behavior. Сделав наследника, мы можем присоединить его к объекту в дереве отображения. В этом классе есть методы OnAttached и OnDetaching, которые вызовутся в соответствующих случаях.

Ниже пример CloseBehavior, который можно присоединить любой кнопке и сделать ее таким образом кнопкой закрытия приложения.

At a glance, a behavior looks similar to an action: a self-contained unit of functionality. The main difference is that actions expect to be invoked, and when invoked, they will perform some operation. A behavior does not have the concept of invocation; instead, it acts more as an add-on to an object: optional functionality that can be attached to an object if desired. It may do certain things in response to stimulus from the environment, but there is no guarantee that the user can control what this stimulus is: it is up to the behavior author to determine what can and cannot be customized.

* EventAggregator(взаимодействие между окнами)

Видимо рассказать через какой механизм по МВВМ реализовать уведомление по окнам, вкладкам, тд. Посмотреть призмовский аггрегатор!

* Freezable

Defines an object that has a modifiable state and a read-only (frozen) state. Classes that derive from Freezable provide detailed change notification, can be made immutable, and can clone themselves.

Deriving From Freezable

A class that derives from Freezable gains the following features:

Special states: a read-only (frozen) state and a writable state.

Thread safety: a frozen Freezable object can be shared across threads.

Detailed change notification: Unlike other DependencyObject objects, a Freezable object provides change notifications when sub-property values change.

Easy cloning: the Freezable class has already implemented several methods that produce deep clones.

* Virtualization

Виртуализация позволяет создавать элементу управления контейнер только для непосредственно отображаемых объектов списка. Только для них выделяется память, при этом элемент хранит схему структуры данных, чтобы при прокрутке или изменении видимых объектов соответственно изменить содержимое контейнера.

Для элемента TreeView виртуализацию можно включить, присвоив вложенному свойству VirtualizingStackPanel.IsVirtualizing значение true

Если же надо использовать виртуализацию для каких-то своих элементов, производных от ItemsControl, или для уже существующих элементов управления, которые используют StackPanel (например, ComboBox), то в этих случаях надо установить свойство ItemsPanel для класса VirtualizingStackPanel и присвоить свойству IsVirtualizing значение true.

Однако при наличии некоторых условий виртуализация отключается:

если контейнеры элементов добавляются напрямую к элементу управления ItemsControl. Например, если объекты ListBoxItem добавляются к ListBox, то ListBox не будет виртуализировать эти ListBoxItem

если объект ItemsControl содержит контейнеры элементов различных типов. Например, объект Menu, может содержать объекты типа Separator и MenuItem

если для прикрепленного свойства VirtualizingStackPanel.IsVirtualizing установлено значение false

если для прикрепленного свойства ScrollViewer.CanContentScroll установлено значение false

VirtualizingStackPanel.VirtualizationMode="Recycling"

Для большей оптимизации производительности VirtualizingStackPanel производит кэширование объектов. С помощью свойств CacheLength и CacheLengthUnit класса VirtualizingStackPanel мы можем настроить кэширование видимых объектов в списках.

* Converter

 <TextBlock Text="{Binding Source={StaticResource myDate},Converter={StaticResource myDateConverter}}" />

Конвертеры используются при работе с binding-ами и позволяют преобразовывать значения в одностороннем или двустороннем порядке (В зависимости от режима биндинга). Конвертеры также бывают двух типов — с одним значением и со множеством. За них отвечают интерфейсы **IValueConverter** и **IMultiValueConverter** соответственно.

<TextBlock>

<TextBlock.Text>

<MultiBinding Converter="{StaticResource conv:IntToStringConverter}" ConverterParameter="plusOne"> <Binding Path="IntProp" /> <Binding Path="StringProp" />

</MultiBinding> </TextBlock.Text> </TextBlock>

* MarkupExtensions

Generally speaking, a XAML parser can either interpret an attribute value as a literal string that can be converted to a primitive, or convert it to an object by some means. One such means is by referencing a type converter; this is documented in the topic TypeConverters and XAML. However, there are scenarios where different behavior is required. For example, a XAML processor can be instructed that a value of an attribute should not result in a new object in the object graph. Instead, the attribute should result in an object graph that makes a reference to an already constructed object in another part of the graph, or a static object. Another scenario is that a XAML processor can be instructed to use a syntax that provides non-default arguments to the constructor of an object. These are the types of scenarios where a markup extension can provide the solution.

<object property="{x:Type prefix:typeNameValue}" .../>

<x:Type TypeName="prefix:typeNameValue"/>

WPF-Specific Markup Extensions:

**StaticResource** provides a value for a property by substituting the value of an already defined resource. A StaticResource evaluation is ultimately made at XAML load time and does not have access to the object graph at run time

**DynamicResource** provides a value for a property by deferring that value to be a run-time reference to a resource. A dynamic resource reference forces a new lookup each time that such a resource is accessed and has access to the object graph at run time. In order to get this access, DynamicResource concept is supported by dependency properties in the WPF property system, and evaluated expressions. Therefore you can only use DynamicResource for a dependency property target.

**Binding** provides a data bound value for a property, using the data context that applies to the parent object at run time. This markup extension is relatively complex, because it enables a substantial inline syntax for specifying a data binding

**RelativeSource** provides source information for a [Binding](https://docs.microsoft.com/en-us/dotnet/api/system.windows.data.binding) that can navigate several possible relationships in the run-time object tree. This provides specialized sourcing for bindings that are created in multi-use templates or created in code without full knowledge of the surrounding object tree

**TemplateBinding** enables a control template to use values for templated properties that come from object-model-defined properties of the class that will use the template. In other words, the property within the template definition can access a context that only exists once the template is applied. For details, see TemplateBinding Markup Extension.

**ColorConvertedBitmap** supports a relatively advanced imaging scenario. For details, see ColorConvertedBitmap Markup Extension.

**ComponentResourceKey** and **ThemeDictionary** support aspects of resource lookup, particularly for resources and themes that are packaged with custom controls.

* DependencyObject Property

Inheritance: Object -> DispatcherObject -> DependencyObject

The DependencyObject class enables Windows Presentation Foundation (WPF) property system services on its many derived classes.

The property system's primary function is to compute the values of properties, and to provide system notification about values that have changed. Another key class that participates in the property system is DependencyProperty. DependencyProperty enables the registration of dependency properties into the property system, and provides identification and information about each dependency property, whereas DependencyObject as a base class enables objects to use the dependency properties.

[DependencyObject](https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyobject?view=netframework-4.8) services and characteristics include the following:

Dependency property hosting support. You register a dependency property by calling the [Register](https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyproperty.register?view=netframework-4.8) method, and storing the method's return value as a public static field in your class.

Attached property hosting support. You register an attached property by calling the RegisterAttached method, and storing the method's return value as a public static read-only field in your class. (There are also additional member requirements; note that this represents a WPF specific implementation for attached properties. For details, see [Attached Properties Overview](https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/attached-properties-overview?view=netframework-4.8).) Your attached property can then be set on any class that derives from DependencyObject.

Get, set, and clear utility methods for values of any dependency properties that exist on the DependencyObject.

Metadata, coerce value support, property changed notification, and override callbacks for dependency properties or attached properties. Also, the DependencyObject class facilitates the per-owner property metadata for a dependency property.

A common base class for classes derived from ContentElement, [Freezable](https://docs.microsoft.com/en-us/dotnet/api/system.windows.freezable?view=netframework-4.8), or [Visual](https://docs.microsoft.com/en-us/dotnet/api/system.windows.media.visual?view=netframework-4.8). (UIElement, another base element class, has a class hierarchy that includes [Visual](https://docs.microsoft.com/en-us/dotnet/api/system.windows.media.visual?view=netframework-4.8).)

Props: [GetValue HYPERLINK "https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyobject.getvalue?view=netframework-4.8"( HYPERLINK "https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyobject.getvalue?view=netframework-4.8"DependencyProperty HYPERLINK "https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyobject.getvalue?view=netframework-4.8")](https://docs.microsoft.com/en-us/dotnet/api/system.windows.dependencyobject.getvalue?view=netframework-4.8), SetValue (System.Windows.DependencyProperty dp, object value);

* Binding, Multibinding

Source for binding becomes first not-null datacontext in visual tree if none specified in some other way

{Binding ElementName=Имя\_объекта-источника, Path=Свойство\_объекта-источника}

ElementName, IsAsync, Mode, Path, TargetNullValue, RelativeSource, Source, XPath

"{Binding ElementName=textBox1, Path=Text, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />

Multibinding

<StackPanel>

<StackPanel.Resources>

<mylocalXAMLnamespace:AddConverter x:Key="XAMLResourceAddConverter" />

</StackPanel.Resources>

<TextBox Name="TextBox1" Text="10"></TextBox>

<TextBox Name="TextBox2" Text="20"></TextBox>

<Label Content="Sum of above two values:"></Label>

<TextBox Name="textBox3">

<TextBox.Text>

<MultiBinding Converter="{StaticResource XAMLResourceAddConverter}">

<Binding ElementName="TextBox1" Path="Text"/>

<Binding ElementName="TextBox2" Path="Text"/>

</MultiBinding>

</TextBox.Text>

</TextBox>

</StackPanel>

PriorityBinding

[PriorityBinding](https://docs.microsoft.com/en-us/dotnet/api/system.windows.data.prioritybinding?view=netframework-4.8) lets you associate a binding target (target) property with a list of bindings. The first binding that returns a value successfully becomes the active binding.

A binding returns a value successfully if:

The path to the binding source resolves successfully.

The value converter, if any, is able to convert the resulting value.

The resulting value is valid for the target property.

The value DependencyProperty.UnsetValue is not considered a successful return value.

The priority of the bindings is determined by their order in the list. The binding that appears first in the list has the highest priority.

The binding engine starts with the first binding in the list and verifies whether that binding returns a value successfully; if it does, the value from that binding is used. If the first binding does not return a value successfully, the binding engine examines the second binding to determine whether it returns a value successfully; if it does, the value from the second binding becomes the active value. This verification process continues to the end of the list of bindings. If none of the bindings returns a value successfully, the binding uses the FallbackValue.

The binding engine continues to listen for changes on all bindings. If at any point one of the bindings that has a higher priority returns a value successfully, the value for that binding becomes the active value and replaces the current value.

* RelativeSource

Свойство RelativeSource

Свойство RelativeSource позволяет установить привязку относительно элемента-источника, который связан какими-нибудь отношениями с элементом-приемником. Например, элемент-источник может быть одним из внешних контейнеров для элемента-приемника. Либо источником и приемником может быть один и тот же элемент.

Для установки этого свойства используется одноименный объект RelativeSource. У этого объекта есть свойство Mode, которое задает способ привязки. Оно принимает одно из значений перечисления RelativeSourceMode:

Self: привязка осуществляется к свойству этого же элемента. То есть элемент-источник привязки в то же время является и приемником привязки.

FindAncestor: привязка осуществляется к свойству элемента-контейнера.

<TextBox Text="{Binding RelativeSource={RelativeSource Mode=Self}, Path=Background, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />

<Grid Background="Black">

    <TextBlock Foreground="White"

        Text="{Binding RelativeSource={RelativeSource Mode=FindAncestor,AncestorType={x:Type Grid}}, Path=Background}" />

</Grid>

* Validation

Мы также можем реализовать свою логику валидации для класса модели. Для этого модель должна реализовать интерфейс IDataErrorInfo.

* ItemsControl

[Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.8)  [DispatcherObject](https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcherobject?view=netframework-4.8) -> DependencyObject -> [Visual](https://docs.microsoft.com/en-us/dotnet/api/system.windows.media.visual?view=netframework-4.8) -> UIElement -> FrameworkElement -> [Control](https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.control?view=netframework-4.8) -> ItemsControl

[System.Windows.Localizability(System.Windows.LocalizationCategory.None, Readability=System.Windows.Readability.Unreadable)]

[System.Windows.Markup.**ContentProperty**("Items")]

[System.Windows.**StyleTypedProperty**(Property="ItemContainerStyle", StyleTargetType=typeof(System.Windows.FrameworkElement))]

public class ItemsControl : System.Windows.Controls.Control, System.Windows.Controls.Primitives.IContainItemStorage, System.Windows.Markup.IAddChild

* Attached props
* Commands
* ME: Translate
* GridView
* Converter with ME (concat converter in Diams)
* Triggers
* Trigger
* Data Trigger
* Event Trigger
* MultiTrigger
* MultiDataTrigger

PRISM

* RegionAdapter