FREE eBook

LEARNING xaml

Free unaffiliated eBook created from
Stack Overflow contributors.

#xaml

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: xaml

It is an unofficial and free xaml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official xaml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with xaml

Remarks

E**X**tensible **A**pplication **M**arkup **L**anguage (XAML) is a XML based markup language developed by Microsoft. It is used in several Microsoft technologies like Windows Presentation Foundation (WPF), Silverlight, WinRT, Universal Windows Platform, etc. to define the User Interface for applications.

Versions

Version	Release Date
WPF XAML	2006-11-21
Silverlight 3	2009-07-09
Silverlight 4	2010-04-15
Windows 8 XAML	2011-09-01

Examples

Installation or Setup

The easiest way to get writing your first XAML is to install Microsoft Visual Studio. This is avaliable free from Microsoft.

Once installed you can create a new project, of type WPF Application, either with a VB.NET or C# code.

This is similar to windows forms in the sense that you have a series of windows, the main difference being that these windows are written in XAML and are much more responsive to different devices.

Still needs improvment.

Hello World

Here is a simple example of an XAML page in WPF. It consists of a `Grid`, a `TextBlock` and a `Button` - the most common elements in XAML.

```
<Window x:Class="FirstWpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="MainWindow"
        Height="350"
        Width="525">
    <Grid>
        <TextBlock Text="Welcome to XAML!"
                   FontSize="30"
                   Foreground="Black"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"/>

        <Button Content="Hello World!"
                Background="LightGray"
                Foreground="Black"
                FontSize="25"
                Margin="0,100,0,0"
                VerticalAlignment="Center"
                HorizontalAlignment="Center"/>
    </Grid>
</Window>
```

Syntax	Description
`<Window>`	The root container which hosts the content that visualizes data and enable users to interact with it. A WPF window is a combination of an XAML (.xaml) file, where the element is the root, and a CodeBehind (.cs) file.
`<Grid>`	A layout panel that arranges its child elements in a tabular structure of rows and columns.
`<TextBlock>`	Provides a lightweight control for displaying string text in its Text property or Inline flow content elements, such as Bold, Hyperlink, and InlineUIContainer, in its Inlines property.
`<Button>`	Represents a button control which reacts with the user click on it.

Property	Description
`Title`	Gets or sets the title of a window.
`Height`	Gets or sets the height of an element.
`Width`	Gets or sets the width of an element.
`Text`	Gets or sets the text content of a text element.
`FontSize`	Gets or sets the top-level font size for the text.
`Background`	Gets or sets the brush color that paints the background of an element.
`Foreground`	Gets or sets the brush color that paints the font of a text in an element.

Property	Description
`Margin`	Gets or sets the value that describes the outer space between an element and the others.
`HorizontalAlignment`	Gets or sets the horizontal alignment characteristics applied to the element when it is composed within a parent element, such as a panel or items control.
`VerticalAlignment`	Gets or sets the vertical alignment characteristics applied to the element when it is composed within a parent element such as a panel or items control.

Read Getting started with xaml online: https://riptutorial.com/xaml/topic/903/getting-started-with-xaml

Chapter 2: Control Templates

Examples

Control Templates

The default user interfaces for WPF controls are typically constructed from other controls and shapes. For example, a Button is composed of both ButtonChrome and ContentPresenter controls. The ButtonChrome provides the standard button appearance, while the ContentPresenter displays the button's content, as specified by the Content property. Sometimes the default appearance of a control may be incongruent with the overall appearance of an application. In this case, you can use a ControlTemplate to change the appearance of the control's user interface without changing its content and behavior.

XAML

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="SDKSample.ControlTemplateButtonWindow"
   Title="Button with Control Template" Height="158" Width="290">

   <!-- Button using an ellipse -->
   <Button Content="Click Me!" Click="button_Click">
     <Button.Template>
       <ControlTemplate TargetType="{x:Type Button}">
         <Grid Margin="5">
           <Ellipse Stroke="DarkBlue" StrokeThickness="2">
             <Ellipse.Fill>
               <RadialGradientBrush Center="0.3,0.2" RadiusX="0.5" RadiusY="0.5">
                 <GradientStop Color="Azure" Offset="0.1" />
                 <GradientStop Color="CornflowerBlue" Offset="1.1" />
               </RadialGradientBrush>
             </Ellipse.Fill>
           </Ellipse>
           <ContentPresenter Name="content" HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
         </Grid>
       </ControlTemplate>
     </Button.Template>

   </Button>

  </Window>
```

C# Code

```
using System.Windows;

   namespace SDKSample
   {
```

```
        public partial class ControlTemplateButtonWindow : Window
        {
            public ControlTemplateButtonWindow()
            {
                InitializeComponent();
            }

            void button_Click(object sender, RoutedEventArgs e)
            {
                MessageBox.Show("Hello, Windows Presentation Foundation!");
            }
        }
    }
```

Read Control Templates online: https://riptutorial.com/xaml/topic/6782/control-templates

Chapter 3: Converters

Parameters

Parameter	Details
value	The value to convert from
targetType	The type being converted to
parameter	Optional value to control how the conversion works
culture	CultureInfo object - required if localisation needed

Remarks

The `Convert` method converts the value from the source (usually the view model) to the target (usually a property of a control).

The `ConvertBack` method converts the value from the target back to the source. It is only needed if the binding is `TwoWay` or `OneWayToSource`.

When a `ConvertBack` is not supported, i.e. there is no one-to-one mapping between the pre-conversion value and the post-conversion value, it's common practice to have the `ConvertBack` method return `DependencyProperty.UnsetValue`. It's a better option than throwing an exception (e.g. `NotImplementedException`) as it avoids unexpected runtime errors. Also, bindings can benefit of their `FallbackValue` when `DependencyProperty.UnsetValue` is returned by a converter.

Examples

String to IsChecked Converter

In XAML:

```
<RadioButton IsChecked="{Binding EntityValue, Mode=TwoWay,
                         Converter={StaticResource StringToIsCheckedConverter},
                         ConverterParameter=Male}"
             Content="Male"/>

<RadioButton IsChecked="{Binding EntityValue, Mode=TwoWay,
                         Converter={StaticResource StringToIsCheckedConverter},
                         ConverterParameter=Female}"
             Content="Female"/>
```

The C# class:

```
public class StringToIsCheckedConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        string input = (string)value;
        string test = (string)parameter;
        return input == test;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        if (value == null || !(value is bool))
        {
            return string.Empty;
        }
        if (parameter == null || !(parameter is string))
        {
            return string.Empty;
        }
        if ((bool)value)
        {
            return parameter.ToString();
        }
        else
        {
            return string.Empty;
        }
    }
}
```

Converters 101

XAML controls may have dependency properties that can be bound to objects from `DataContext` or other controls. When the type of the object being bound is different from the type of the target `DependencyProperty`, a **converter** may be used to adapt one type to another.

Converters are classes implementing `System.Windows.Data.IValueConverter` or `System.Windows.Data.IMultiValueConverter`; WPF implements some out of the box converters, but developers may see use in custom implementations, as it is frequently the case.

To use a converter in XAML, an instance can be instantiated in the `Resources` section. For the example below, `System.Windows.Controls.BooleanToVisibilityConverter` will be used:

```
<UserControl.Resources>
    <BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
</UserControl.Resources>
```

Notice the `x:Key` element defined, which is then used to reference the instance of `BooleanToVisibilityConverter` in the binding:

```
<TextBlock Text="This will be hidden if property 'IsVisible' is true"
           Visibility="{Binding IsVisible,
                                Converter={StaticResource BooleanToVisibilityConverter}}"/>
```

In the example above, a boolean `IsVisible` property is converted to a value of the `System.Windows.Visibility` enumeration; `Visibility.Visible` if true, or `Visibility.Collapsed` otherwise.

Creating and using a Converter: BooleanToVisibilityConverter and InvertibleBooleanToVisibilityConverter

To extend and expand upon the binding experience we have converters to convert one a value of one type into another value of another type. To leverage Converters in a Databinding you first need to create a DataConverter class tht extens either

- `IValueConverter`**(WPF & UWP)**

or

- `IMultiValueConverter`**(WPF)**

if you want to convert multiple types into one type
In this case we focus on converting a `boolean True/False` value to the correspionding Visibilities `Visibility.Visible` and `Visibility.Collapsed`:

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return (value is bool && (bool) value) ? Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter, string
language)
    {
        return (value is Visibility && (Visibility) value == Visibility.Visible);
    }
}
```

The **`Convert`** method is called whenever you **`GET`** data **`FROM`** the **`ViewModel`**.
The **`ConvertBack`** is called upon **`SET`** ing data **`TO`** the **`ViewModel`** for **`BindingMode.TwoWay`** bindings.

Of course you can also utilize properties within your converter. Take a look at this one:

```
public class InvertibleBooleanToVisibilityConverter : IValueConverter
{
    public bool Invert { get; set; } = false;

    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return (value is bool && (bool) value != Invert) ? Visibility.Visible :
Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter, string
language)
    {
        return (value is Visibility && ((Visibility) value == Visibility.Visible) != Invert);
```

```
    }
}
```

If you want to use a converter in a `Binding`, simply declare it as a resource in your page, window, or other element, give it a key and supply potentially needed properties:

```
<Page ...
    xmlns:converters="using:MyNamespce.Converters">
<Page.Resources>
    <converters:InvertibleBooleanToVisibilityConverter
        x:Key="BooleanToVisibilityConverter"
        Invert="False" />
</Page.Resources>
```

and use it as a `StaticResource` in a binding:

```
<ProgressRing
        Visibility="{Binding ShowBusyIndicator,
            Converter={StaticResource BooleanToVisibilityConverter},
            UpdateSourceTrigger=PropertyChanged,
            Mode=OneWay}" />
```

Read Converters online: https://riptutorial.com/xaml/topic/2996/converters

Chapter 4: Data Binding

Syntax

- `<TextBlock Text="{Binding Title}"/>`
- `<TextBlock Text="{Binding Path=Title}"/>`
- `<TextBlock> <TextBlock.Text> <Binding Path="Title"/> </TextBlock.Text> </TextBlock>`

Remarks

All these tags produce the same result.

Examples

Binding string to Text property

To change UI content in runtime, you can use `Binding`. When binded property is changed from the code, it will be displayed to the UI.

```
<TextBlock Text="{Binding Title}"/>
```

To notify UI about changes, property must raise `PropertyChanged` event from `INotifyPropertyChanged` interface or you can use `Dependency Property`.

The Binding is working if the property "Title" is in the xaml.cs file or in the Datacontext class from the `XAML`.

The Datacontext can be set up in the XAML directly

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Application">
<Window.DataContext>
   <local:DataContextClass/>
</Window.DataContext>
```

Formatting String Bindings

When making a bind of something, for example a date you may want to show it in a specific format without messing around with it in the code.

To do this we can use the StringFormat property.

Here are some examples:

```
Text="{Binding Path=ReleaseDate, StringFormat=dddd dd MMMM yyyy}"
```

This formats my date to the following:

Tuesday 16 August 2016

Here is another example for temperature.

```
Text="{Binding Path=Temp, StringFormat={}{0}°C}"
```

This formats to:

25°C

The basics of INotifyPropertyChanged

If you do not only wish to display static objects, but have your UI respond to changes to correlating objects, you need to understand the basics of the `INotifyPropertyChanged` interface.

Assuming we have our `MainWindow`defined as

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:vm="clr-namespace:Application.ViewModels>
    <Window.DataContext>
       <vm:MainWindowViewModel/>
    </Window.DataContext>
    <Grid>
        <TextBlock Text={Binding Path=ApplicationStateText}" />
    </Grid>
</Window>
```

With our Viewmodel-Class `MainWindowViewModel` defined as

```
namespace Application.ViewModels
{
    public class MainWindowViewModel
    {
        private string _applicationStateText;

        public string ApplicationStateText
        {
            get { return _applicationStateText; }
            set { _applicationStateText = value; }
        }
        public MainWindowViewModel()
        {
            ApplicationStateText = "Hello World!";
        }

    }
```

```
}
```

the TextBlock of our Application will display the Text *Hello World* due to its binding. If our ApplicationStateText changes during runtime, our UI will not be notified of such change.
In order to implement this, our DataSource, in this case our `MainWindowViewModel`, needs to implement the Interface `INotifyPropertyChanged`. This will cause our `Bindings`to be able to subscribe to the `PropertyChangedEvent`.
All we need to do is to Invoke the `PropertyChangedEventHandler` whenever we change our `ApplicationStateText` Property like this:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Application.ViewModels
{
    public class MainWindowViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        public void NotifyPropertyChanged( [CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }

        private string _applicationStateText;

        public string ApplicationStateText
        {
            get { return _applicationStateText; }
            set
            {
                if (_applicationStateText != value)
                {
                    _applicationStateText = value;
                    NotifyPropertyChanged();
                }
            }
        }
        public MainWindowViewModel()
        {
            ApplicationStateText = "Hello World!";
        }
    }
}
```

and make sure, that our `Binding` of `TextBlock.Text` actually listens to a `PropertyChangedEvent`:

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:vm="clr-namespace:Application.ViewModels">
    <Window.DataContext>
       <vm:MainWindowViewModel/>
    </Window.DataContext>
    <Grid>
        <TextBlock Text={Binding Path=ApplicationStateText,
UpdateSourceTrigger=PropertyChanged }" />
```

```
    </Grid>
</Window>
```

Binding to a Collection of Objects with INotifyPropertyChanged and INotifyCollectionChanged

Let's assume you have a `ListView` wich is supposed to display every `User` object listed under the `Users` Property of the `ViewModel` where Properties of the User object can get updated programatically.

```
<ListView ItemsSource="{Binding Path=Users}" >
    <ListView.ItemTemplate>
        <DataTemplate DataType="{x:Type models:User}">
            <StackPanel Orientation="Horizontal">
                <TextBlock Margin="5,3,15,3"
                           Text="{Binding Id, Mode=OneWay}" />
                <TextBox Width="200"
                           Text="{Binding Name, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged, Delay=450}"/>
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Despite for `INotifyPropertyChanged` beeing implemented correctly for the `User` object

```
public class User : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private int _id;
    private string _name;

    public int Id
    {
        get { return _id; }
        private set
        {
            if (_id == value) return;
            _id = value;
            NotifyPropertyChanged();
        }
    }
    public string Name
    {
        get { return _name; }
        set
        {
            if (_name == value) return;
            _name = value;
            NotifyPropertyChanged();
        }
    }

    public User(int id, string name)
    {
```

```
        Id = id;
        Name = name;
    }

    private void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

and for your `ViewModel` object

```
public sealed class MainWindowViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private List<User> _users;
    public List<User> Users
    {
        get { return _users; }
        set
        {
            if (_users == value) return;
            _users = value;
            NotifyPropertyChanged();
        }
    }
    public MainWindowViewModel()
    {
        Users = new List<User> {new User(1, "John Doe"), new User(2, "Jane Doe"), new User(3,
"Foo Bar")};
    }

    private void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

your UI wont update, if a change to a User is made programmatically.

This is simply because you have only set INotifyPropertyChanged on the Instance of the List itself. Only if you completely re-instantiate the List if one property of an Element changes your UI will update.

```
// DO NOT DO THIS
User[] userCache = Users.ToArray();
Users = new List<User>(userCache);
```

This however is very tiresome and unbelievably bad for performance.
If you have a List of 100'000 Elements showing both the ID and Name of the User, there will be 200'000 DataBindings in place wich each have to be re-created. This results in noticable Lag to the User whenever a change is made to anything.

To partly solve this issue, you can use `System.ComponentModel.ObservableCollection<T>` instead of `List<T>`:

```
private ObservableCollection<User> _users;
public ObservableCollection<User> Users
{
    get { return _users; }
    set
    {
        if (_users == value) return;
        _users = value;
        NotifyPropertyChanged();
    }
}
```

The `ObservableCollection` provides us with the `CollectionChanged`Event and implements `INotifyPropertyChanged` itself. According to MSDN the Event will rise, "[..]when an item is *added*, *removed*, *changed*, *moved*, or the *entire list is refreshed*".
You will however quickly come to realize that with .NET 4.5.2 and prior, the `ObservableCollection` will not raise a CollectionChanged Event if a Property of an Element in the Collection Changes as discussed here.

Following this solution we can simply implement our own `TrulyObservableCollection<T>` without the `INotifyPropertyChanged` constraint for `T` having everything we need and exposing wether `T` implements `INotifyPropertyChanged` or not:

```
/*
 * Original Class by Simon @StackOverflow http://stackoverflow.com/a/5256827/3766034
 * Removal of the INPC-Constraint by Jirajha @StackOverflow
 * according to to suggestion of nikeee @StackOverflow
http://stackoverflow.com/a/10718451/3766034
 */
public sealed class TrulyObservableCollection<T> : ObservableCollection<T>
{
    private readonly bool _inpcHookup;
    public bool NotifyPropertyChangedHookup => _inpcHookup;

    public TrulyObservableCollection()
    {
        CollectionChanged += TrulyObservableCollectionChanged;
        _inpcHookup =
typeof(INotifyPropertyChanged).GetTypeInfo().IsAssignableFrom(typeof(T));
    }
    public TrulyObservableCollection(IEnumerable<T> items) : this()
    {
        foreach (var item in items)
        {
            this.Add(item);
        }
    }

    private void TrulyObservableCollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)
    {
        if (NotifyPropertyChangedHookup && e.NewItems != null && e.NewItems.Count > 0)
        {
            foreach (INotifyPropertyChanged item in e.NewItems)
            {
                item.PropertyChanged += ItemPropertyChanged;
            }
```

```
        }
        if (NotifyPropertyChangedHookup && e.OldItems != null && e.OldItems.Count > 0)
        {
            foreach (INotifyPropertyChanged item in e.OldItems)
            {
                item.PropertyChanged -= ItemPropertyChanged;
            }
        }
    }
    private void ItemPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        var args = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Replace,
sender, sender, IndexOf((T)sender));
        OnCollectionChanged(args);
    }
}
```

and define our Property `Users` as `TrulyObservableCollection<User>` in our ViewModel

```
private TrulyObservableCollection<string> _users;
public TrulyObservableCollection<string> Users
{
    get { return _users; }
    set
    {
        if (_users == value) return;
        _users = value;
        NotifyPropertyChanged();
    }
}
```

Our UI will now get notified about once a INPC-Property of an element within the Collection changes without the need to re-create every single `Binding`.

Read Data Binding online: https://riptutorial.com/xaml/topic/3329/data-binding

Chapter 5: Data templates

Examples

Using DataTemplate in a ListBox

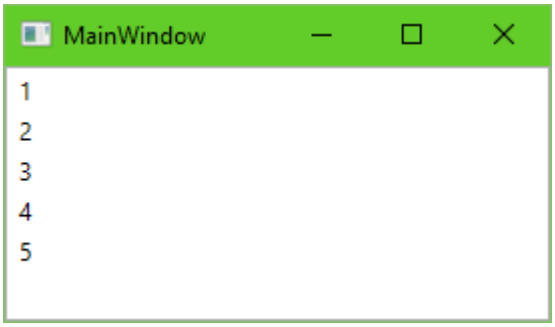
Suppose we have the following XAML snippet:

```
<ListBox x:Name="MyListBox" />
```

Then in the code-behind for this XAML file, we write the following in the constructor:

```
MyListBox.ItemsSource = new[]
{
    1, 2, 3, 4, 5
};
```

Running the application, we get a list of numbers we entered.



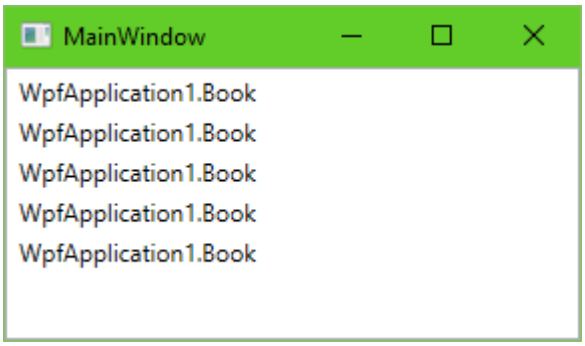
However, if we try to display a list of objects of a custom type, like this

```
MyListBox.ItemsSource = new[]
{
    new Book { Title = "The Hitchhiker's Guide to the Galaxy", Author = "Douglas Adams" },
    new Book { Title = "The Restaurant at the End of the Universe", Author = "Douglas Adams"
},
    new Book { Title = "Life, the Universe and Everything", Author = "Douglas Adams" },
    new Book { Title = "So Long, and Thanks for All the Fish", Author = "Douglas Adams" },
    new Book { Title = "Mostly Harmless", Author = "Douglas Adams" }
};
```

assuming we have a class called `Book`

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
}
```

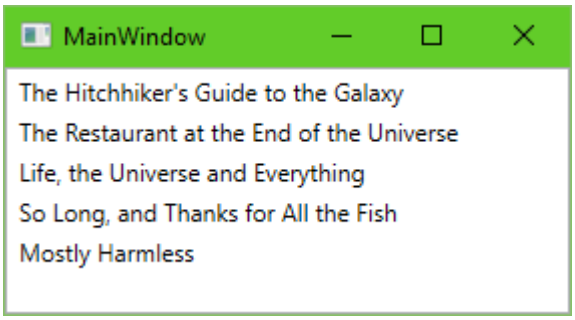
then the list would look something like this:



While we might assume the ListBox will be "smart enough" to display our book objects just right, what we actually see is the full name of the `Book` type. What ListBox actually does here is calling the built-in `ToString()` method on objects it wants to display, and while that produces the desirable outcome in the case of numbers, calling `ToString()` on objects of custom classes results in getting the name of their type, as seen on the screenshot.

We could solve that by writing `ToString()` for our book class, i.e.

```
public override string ToString()
{
    return Title;
}
```



However, that's not very flexible. What if we want to display the author as well? We could write that into the `ToString` too, but what if we don't want that in all lists in the app? How about a nice book cover to display?

That's where DataTemplates can help. They are snippets of XAML that can be "instantiated" as needed, filled in with details according to the source data they are created for. Simply put, if we extend our ListBox code as follows:

```
<ListBox x:Name="MyListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Title}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

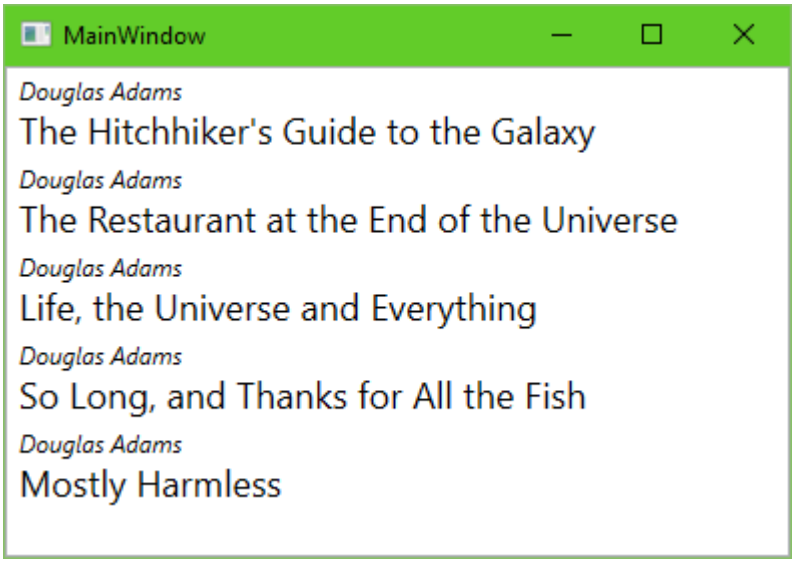
then the list will create a `TextBox` for each item in its source, and those TextBoxes will have their `Text` properties "filled in" with values from the `Title` property of our objects.

If we run the application now, we get the same output as above, *even if we delete the custom

`ToString` implementation. What's beneficial about this is that we can then customize this template well beyond the capabilities of a simple `string` (and `ToString`). We can use any XAML element we want, including custom ones, and can "bind" their values to actual data from our objects (like `Title` in the example above). For example, extend the template as follows:

```
<ListBox x:Name="MyListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock FontStyle="Italic" Text="{Binding Author}" />
                <TextBlock FontSize="18" Text="{Binding Title}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Then we get a nice formatted view of our books!



Read Data templates online: https://riptutorial.com/xaml/topic/3858/data-templates

Chapter 6: Differences in the various XAML dialects

Remarks

XAML is used in Silverlight, Windows Phone, Windows RT and UWP apps. Sharing code or converting code between these is sometimes harder than desirable due to subtle differences between the various XAML dialects. This topic strives to give an overview of these differences with a short explanation.

Examples

Compiled data bindings: The {x:Bind} markup extension

Databings are essential for working with XAML. The XAML dialect for UWP apps provides a type of binding: the {x:Bind} markup extension.

Working with {Binding XXX} and {x:Bind XXX} is mostly equivalent, with the difference that the x:Bind extension works at compile time, which enables better debugging capabilities (e.g. break points) and better performance.

```
<object property="{x:Bind bindingPath}" />
```

The x:Bind markup extension is only available for UWP apps. Learn more about this in this MSDN article: https://msdn.microsoft.com/en-us/windows/uwp/data-binding/data-binding-in-depth.

Alternatives for Silverlight, WPF, Windows RT: Use the standard {Binding XXX} syntax:

```
<object property="{Binding bindingPath}" />
```

Importing namespaces in XAML

Most of the time you need to import namespaces in your XAML file. How this is done is different for the different XAML variants.

For Windows Phone, Silverlight, WPF use the clr-namespace syntax:

```
<Window ... xmlns:internal="clr-namespace:rootnamespace.namespace"
            xmlns:external="clr-namespace:rootnamespace.namespace;assembly=externalAssembly"
>
```

Windows RT, UWP use the using syntax:

```
<Page ... xmlns:internal="using:rootnamespace.namespace"
```

```
            xmlns:external="using:rootnamespace.namespace;assembly=externalAssembly"
>
```

Multi Binding

Multi Binding is a feature exclusive for WPF development. It allows a binding to multiple values at once (typically used with a MultiValueConverter).

```
<TextBox>
    <TextBox.Text>
        <MultiBinding Converter="{StaticResource MyConverter}">
            <Binding Path="PropertyOne"/>
            <Binding Path="PropertyTwo"/>
        </MultiBinding>
    </TextBox.Text>
</TextBox>
```

Platforms other than WPF don't support multi binding. You have to find alternative solutions (like moving the code from view and converters to the viewmodel) or resort 3rd party behaviours like in this article: http://www.damirscorner.com/blog/posts/20160221-MultibindingInUniversalWindowsApps.html)

Read Differences in the various XAML dialects online: https://riptutorial.com/xaml/topic/4498/differences-in-the-various-xaml-dialects

Chapter 7: Layout controls

Examples

Canvas

`Canvas` is the simplest of panels. It places items at the specified `Top/Left` coordinates.

```
<Canvas>
  <TextBlock
    Canvas.Top="50"
    Canvas.Left="50"
    Text="This is located at 50, 50"/>
  <TextBlock
    Canvas.Top="100"
    Canvas.Left="50"
    Width="150"
    Height="23"
    Text="This is located at 50, 100 with height 23 and width 150"/>
</Canvas>
```

DockPanel

`DockPanel` aligns the control according to the dock property, in the order it's placed in the control.

NOTE: `DockPanel` is part of the WPF framework, but does not come with Silverlight/WinRT/UWP. Open-source implementations are easy to find though.

```
<DockPanel LastChildFill="False">
  <!-- This will strech along the top of the panel -->
  <Button DockPanel.Dock="Top" Content="Top"/>
  <!-- This will strech along the bottom of the panel -->
  <Button DockPanel.Dock="Bottom" Content="Bottom"/>
  <!-- This will strech along the remaining space in the left of the panel -->
  <Button DockPanel.Dock="Left" Content="Left"/>
  <!-- This will strech along the remaining space in the right of the panel -->
  <Button DockPanel.Dock="Right" Content="Right"/>
  <!-- Since LastChildFill is false, this will be placed at the panel's right, to the left of
the last button-->
  <Button DockPanel.Dock="Right" Content="Right"/>
</DockPanel>
```

```
<!-- When lastChildFill is true, the last control in the panel will fill the remaining space,
no matter what Dock was set to it -->
<DockPanel LastChildFill="True">
  <!-- This will strech along the top of the panel -->
  <Button DockPanel.Dock="Top" Content="Top"/>
  <!-- This will strech along the bottom of the panel -->
  <Button DockPanel.Dock="Bottom" Content="Bottom"/>
  <!-- This will strech along the remaining space in the left of the panel -->
  <Button DockPanel.Dock="Left" Content="Left"/>
  <!-- This will strech along the remaining space in the right of the panel -->
```

```
  <Button DockPanel.Dock="Right" Content="Right"/>
  <!-- Since LastChildFill is true, this will fill the remaining space-->
  <Button DockPanel.Dock="Right" Content="Fill"/>
</DockPanel>
```

StackPanel

StackPanel places its controls one after another. It acts like a dock panel with all of its control's docks set to the same value.

```
<!-- The default StackPanel is oriented vertically, so the controls will be presented in order
from top to bottom -->
<StackPanel>
  <Button Content="First"/>
  <Button Content="Second"/>
  <Button Content="Third"/>
  <Button Content="Fourth"/>
</StackPanel>
```

```
<!-- Setting the Orientation property to Horizontal will display the control in order from
left to right (or right to left, according to the FlowDirection property) -->
<StackPanel Orientation="Horizontal">
  <Button Content="First"/>
  <Button Content="Second"/>
  <Button Content="Third"/>
  <Button Content="Fourth"/>
</StackPanel>
```

To stack items from the bottom up, use a dock panel.

Grid

`Grid` is used to create table layouts.

Basic rows and columns definitions

```
<Grid>
  <!-- Define 3 columns with width of 100 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="100"/>
  </Grid.ColumnDefinitions>
  <!-- Define 3 rows with height of 50 -->
  <Grid.RowDefinitions>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
  </Grid.RowDefinitions>
  <!-- This is placed at the top left (first row, first column) -->
  <Button
      Grid.Column="0"
      Grid.Row="0"
```

```
      Content="Top Left"/>
  <!-- This is placed at the top left (first row, second column) -->
  <Button
      Grid.Column="1"
      Grid.Row="0"
      Content="Top Center"/>
  <!-- This is placed at the center (second row, second column) -->
  <Button
      Grid.Column="1"
      Grid.Row="1"
      Content="Center"/>
  <!-- This is placed at the bottom right (third row, third column) -->
  <Button
      Grid.Column="2"
      Grid.Row="2"
      Content="Bottom Right"/>
</Grid>
```

NOTE: All the following examples will use only columns, but are applicable to rows as well.

Auto size definitions

Columns and rows can be defined with "Auto" as their width/height. Auto size will take *as much space as it needs* to display its content, and no more.
Auto sized definitions can be used with fixed size definitions.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="50"/>
  </Grid.ColumnDefinitions>
  <!-- This column won't take much space -->
  <Button Grid.Column="0" Content="Small"/>
  <!-- This column will take much more space -->
  <Button Grid.Column="1" Content="This text will be very long."/>
  <!-- This column will take exactly 50 px -->
  <Button Grid.Column="2" Content="This text will be cut"/>
</Grid>
```

Simple star sized definitions

Columns and rows can be defined with `*` as their width/height. Star sized rows/columns will take *as much space as it has*, regardless of it's content.
Star sized definitions can be used with fixed and auto sized definitions. Star size is the default and thus the column width or row height can be omitted.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="50"/>
```

```
    </Grid.ColumnDefinitions>
    <!-- This column will be as wide as it can -->
    <Button Grid.Column="0" Content="Small"/>
    <!-- This column will take exactly 50 px -->
    <Button Grid.Column="2" Content="This text will be cut"/>
</Grid>
```

Proportional star sized definitions

Besides the fact that star takes as much space as it can, star definitions are also proportional to each other. If nothing else is mentioned, each star definition will take as much space as the others in the current grid.

However, it is possible to define a ratio between the sizes of different definitions by simply adding a multiplier to it. So a column defined as `2*` will be twice as wide as a column defined as `*`. The width of a single unit is calculated by dividing the available space by the sum of the multipliers (if there's non it's counted as 1).
So a grid with 3 columns defined as `*`, `2*`, `*` will be presented as 1/4, 1/2, 1/4.
And one with 2 columns defined as `2*`, `3*` will be presented 2/5, 3/5.

If there are auto or fixed definitions in the set, these will be calculated first, and the star definitions will take the remaining space after that.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <!-- This column will be as wide as the third column -->
    <Button Grid.Column="0" Content="Small"/>
    <!-- This column will be twice as wide as the rest -->
    <Button Grid.Column="1" Content="This text will be very long."/>
    <!-- This column will be as wide as the first column -->
    <Button Grid.Column="2" Content="This text will may be cut"/>
</Grid>
```

Column/Row Span

It's possible to make a control stretch beyond it's cell by setting it's Row/ColumnSpan. The value set is the number of rows/columns th

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <!-- This control will streach across most of the grid -->
    <Button Grid.Column="0" Grid.ColumnSpan="2" Content="Small"/>
```

```
    <Button Grid.Column="2" Content="This text will may be cut"/>
</Grid>
```

WrapPanel

The wrap panel acts in a similar way to stack panel. Except when it recognizes the items will exceed it's size, it would then wrap them to a new row/column, depending on it's orientation.

Horizontal orientation

```
<WrapPanel Width="100">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</WrapPanel>
```

Vertical wrap panel

```
<WrapPanel Height="70" Orientation="Vertical">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</WrapPanel>
```

UniformGrid

Uniform grid will place all it's children in a grid layout, each child in it's own cell. All the cells will have the same size. It can be thought to be a shorthand to a grid where all the row and column definitions are set to `*`

Default rows and columns

By default the UniformGrid will try to create an equal number of rows and columns. When a row will become to long, it will add a new column.

This code will produce a grid of 3x3 with the first 2 rows filled and the last with one button:

```
<UniformGrid>
```

```
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</UniformGrid>
```

Specified rows / columns

You can tell the UniformGrid exactly how many rows and/or column you wish to have.

```
<UniformGrid Columns="2" >
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</UniformGrid>
```

NOTE: in case both rows and columns are set, and there are more children than cells, the last children in the grid won't be displayed

---

FirstColumn Property

Once the Columns property is set, you can set the FirstColumn property. This property will enter x empty cells to the first row before the first child is displayed. FirstColumn must be set to a number smaller than the Columns property.

In this example the first button will be displayed in the first row's second column:

```
<UniformGrid Columns="2" FirstColumn="1">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</UniformGrid>
```

RelativePanel

`RelativePanel` has been introduced in Windows 10 and is used mainly to support adaptive layouts, where the child elements of the panel are laid out differently depending on available space. `RelativePanel` is typically used with visual states, which are used to switch the layout configuration, adapting to the screen or window size, orientation or use case. The child elements use attached

properties that define where they are in relation to the panel and each other.

```
<RelativePanel
    VerticalAlignment="Stretch"
    HorizontalAlignment="Stretch">
    <Rectangle
        x:Name="rectangle1"
        RelativePanel.AlignLeftWithPanel="True"
        Width="360"
        Height="50"
        Fill="Red"/>
    <Rectangle
        x:Name="rectangle2"
        RelativePanel.Below="rectangle1"
        Width="360"
        Height="50"
        Fill="Green" />
</RelativePanel>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState>
            <VisualState.StateTriggers>
                <!--VisualState to be triggered when window width is >=720 effective pixels.--
>
                <AdaptiveTrigger
                    MinWindowWidth="720" />
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter
                    Target="rectangle2.(RelativePanel.Below)"
                    Value="{x:Null}" />
                <Setter
                    Target="rectangle2.(RelativePanel.RightOf)"
                    Value="rectangle1" />
                <Setter
                    Target="rectangle1.(RelativePanel.AlignLeftWithPanel)"
                    Value="False" />
                <Setter
                    Target="rectangle1.(RelativePanel.AlignVerticalCenterWithPanel)"
                    Value="True" />
                <Setter
                    Target="rectangle2.(RelativePanel.AlignVerticalCenterWithPanel)"
                    Value="True" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Read Layout controls online: https://riptutorial.com/xaml/topic/3634/layout-controls

Chapter 8: Working with custom XAML files

Examples

Reading an object from XAML

Consider a structure of the following classes should be constructed in XAML an then read into a CLR object:

```
namespace CustomXaml
{
    public class Test
    {
        public string Value { get; set; }
        public List<TestChild> Children { get; set; } = new List<TestChild>();
    }

    public class TestChild
    {
        public string StringValue { get; set; }
        public int IntValue { get; set; }
    }
}
```

Classes should either have no explicit constructor or provide an empty one. To keep the XAML clean, collections need to be initialised. Initialising collections in XAML is also possible though.

To read XAML the `XamlServices` class can be used. It is defined in `System.Xaml` which needs to be added to references. The following line then reads the `test.xaml` file from disk:

```
Test test = XamlServices.Load("test.xaml") as Test;
```

The `XamlServices.Load` method has several overloads to load from streams and other sources. If reading XAML from an embedded file (like it is done in WPF) the `Build Action` property that is set to `Page` by default needs to be changed to i.e. `Embedded Resource`. Otherwise the compiler will ask for references to WPF assemblies.

The content of the XAML file to read should look something like this:

```
<Test xmlns="clr-namespace:CustomXaml;assembly=CustomXaml"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Value="test">
    <Test.Children>
        <TestChild StringValue="abc" IntValue="123"/>
        <TestChild StringValue="{x:Null}" IntValue="456"/>
    </Test.Children>
</Test>
```

The pure `xmlns` Definition allows the use of classes in the same namespace without prefix. The Definition of the `xmlns:x` is neccessary to use constructs like `{x:Null}`. Of course prefixes for other

namespaces or assemblies can be defined as needed.

Writing an object to XAML

Consider a structure of the following classes should be constructed in XAML an then read into a CLR object:

```
namespace CustomXaml
{
    public class Test
    {
        public string Value { get; set; }
        public List<TestChild> Children { get; set; } = new List<TestChild>();
    }

    public class TestChild
    {
        public string StringValue { get; set; }
        public int IntValue { get; set; }
    }
}
```

To write XAML the `XamlServices` class can be used. It is defined in `System.Xaml` which needs to be added to references. The following line then writes the instance `test` which is of type `Test` to the file `test.xaml` on disk:

```
XamlServices.Save("test.xaml", test);
```

The `XamlServices.Save` method has several overloads to write to streams and other targets. The resulting XAML should look something like this:

```
<Test Value="test" xmlns="clr-namespace:CustomXaml;assembly=CustomXaml"
                   xmlns:scg="clr-namespace:System.Collections.Generic;assembly=mscorlib"
                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Test.Children>
    <scg:List x:TypeArguments="TestChild" Capacity="4">
      <TestChild IntValue="123" StringValue="abc" />
      <TestChild IntValue="456" StringValue="{x:Null}" />
    </scg:List>
  </Test.Children>
</Test>
```

The pure `xmlns` Definition allows the use of classes in the same namespace without prefix. The Definition of the `xmlns:x` is neccessary to use constructs like `{x:Null}`. The writer automatically adds the `xmlns:scg` to initialize a `List<TestChild>` for the `Children` property of the `Test` object. It does not rely on the property being initialized by the constructor.

Read Working with custom XAML files online: https://riptutorial.com/xaml/topic/6693/working-with-custom-xaml-files

Chapter 9: XAML Development Tools

Examples

Microsoft Visual Studio & Microsoft Expression Blend

Create engaging user interfaces for Windows Desktop Applications with Blend for Visual Studio, the premier professional design tool for XAML applications. Build beautiful transitions and visualizations using Blend's full suite of vector drawing tools, powerful template editing features, real-time animation, visual state management and more.

Download Visual Studio

WPF Inspector

WPF Inspector is a utility that attaches to a running WPF application to troubleshoot common problems with layouting, databinding or styling. WPF Inspector allows you to explore a live view of the logical- and visual tree, read and edit property values of elements, watch the data context, debug triggers, trace styles and much more.

Download WPF Inspector from Codeplex

Snoop

Snoop is an open source tool available which allows you to browse the visual tree of a running WPF application without the need for a debugger and change properties.

Download WPF Snoop from GitHub

WPF Performance Suite

The Windows SDK includes a suite of performance profiling tools for Windows Presentation Foundation (WPF) applications called the WPF Performance Suite. The WPF Performance Suite enables you to analyze the run-time behavior of your WPF applications and determine performance optimizations that you can apply. The WPF Performance Suite includes performance profiling tools called Perforator and Visual Profiler. This topic describes how to install and use the Perforator and Visual Profiler tools in the WPF Performance Suite.

Read more MSDN

Read XAML Development Tools online: https://riptutorial.com/xaml/topic/6800/xaml-development-tools

Credits

S. No	Chapters	Contributors
1	Getting started with xaml	ChrisF, Community, Explisam, Raamakrishnan A., Rafael Costa , RenDishen, Ryan Thomas
2	Control Templates	Vimal CK
3	Converters	Adi Lester, AhammadaliPK, ChrisF, Jirajha, Rafael Costa
4	Data Binding	Jirajha, RenDishen, Ryan Thomas, SeeuD1, Tobias
5	Data templates	Dániel Kis-Nagy
6	Differences in the various XAML dialects	HeWillem, stefan.s
7	Layout controls	CKII, Filip Skakun
8	Working with custom XAML files	Alexander Mandt
9	XAML Development Tools	Vimal CK