Overall tutorial: <https://www.tutorialspoint.com/wpf/index.htm>

# Grid Panel

A Grid Panel, which consists of **columns and rows**, arranges controls in a **tabular format**. It makes dividing up the available space into individual cells very easily.

## Create Grid Panels in XAML

The following example creates a grid with 3 columns and 3 rows:

<Window x:Class="GridSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="GridSample" Height="300" Width="450" WindowStyle="ThreeDBorderWindow">

<Grid Name="MyGrid" Width="400" Background="LightSteelBlue" ShowGridLines="True">

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="45" />

<RowDefinition Height="45" />

<RowDefinition Height="45" />

</Grid.RowDefinitions>

<TextBlock

FontSize="14" FontWeight="Bold" Grid.Row="0" Grid.Column="0"

Foreground="Green" Text="Author Name" Height="20" VerticalAlignment="Top" />

<TextBlock

FontSize="14" FontWeight="Bold" Grid.Row="0" Grid.Column="1"

Foreground="Green" Text="Age" Height="20" VerticalAlignment="Top" />

<TextBlock

FontSize="14" FontWeight="Bold" Grid.Row="0" Grid.Column="2"

Foreground="Green" Text="Book" Height="20" VerticalAlignment="Top"/>

<TextBlock FontSize="12" Grid.Row="1" Grid.Column="0"> Mahesh Chand </TextBlock>

<TextBlock FontSize="12" Grid.Row="1" Grid.Column="1"> 33</TextBlock>

<TextBlock FontSize="12" Grid.Row="1" Grid.Column="2"> GDI+ Programming </TextBlock>

<TextBlock FontSize="12" Grid.Row="2" Grid.Column="0"> Mike Gold </TextBlock>

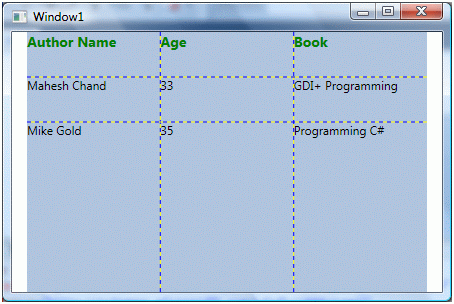
<TextBlock FontSize="12" Grid.Row="2" Grid.Column="1"> 35 </TextBlock>

<TextBlock FontSize="12" Grid.Row="2" Grid.Column="2"> Programming C# </TextBlock>

</Grid>

</Window>

Output:

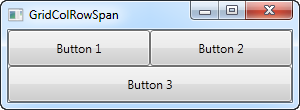


Grid has three major properties:

* Grid.ColumnDefinitions: is a collection of ColumnDefinition.
* Grid.RowDefinitions: is a collection of RowDefintion.
* Grid.ShowGridLines: property represents if grid lines of a Grid are visible or not.

Any control in WPF can be placed within a grid by using its Grid.Row and Grid.Column properties which represent **what column and what row a control will be placed in**. These values **start with** 0. That means, if there are three columns in a grid, columns will be represented by the number 0, 1 and 2 respectively.

In addition, there are two common-used properties named Grid.RowSpan and Grid.ColumnSpan which **allow controls in Grid to take up more than one row or column**. For example, in the window below, “Button3” takes two columns; which means Grid.ColumnSpan="2".



## Managing Column Width and Row Height

The ColumnDefinition has three properties that are used to manage the width of a column in a Grid:

* Width: represents the width of a column.
* MaxWidth and MinWidth: set maximum and minimum width of a column (useful when resizing controls).

The RowDefinition has three properties that are used to manage the height of a row in a Grid:

* Height: represents the height of a row.
* MaxHeight and MinHeight: set maximum and minimum height of a row (useful when resizing controls).

The column width and row height can be defined by one of three ways:

### Using Pixels

In the example above, we set Height="45" (45 pixels) for RowDefinition.

### Using "\*"

Star sizing distributes remaining available space **proportionally**. By default, column width and row height are set to "\*" if Width and Height are missed.

For example:

<!-- give 30% to column 1 and 70% to column 2 -->

<ColumnDefinition Width="3\*" />

<ColumnDefinition Width="7\*" />

IMG_256

Tip: The numbers don’t have to be integers. If Width for RowDefinition (or Height for ColumnDefinition) is omitted, 1\* is implied. For example:

<!-- column 1 is 1.5 times wider than column 2 -->

<ColumnDefinition Width="1.5\*" />

<ColumnDefinition Width="" />

IMG_257

Note: In addition to distributing space proportionally, using (\*) has a very big benefit – it **allows controls to change their sizes automatically when the user resizes (eg: maximize) the window** holding these controls. If we use fixed sizes (in pixels), we CANNOT do that.

For example: <https://stackoverflow.com/a/37782394>

### Using "Auto"

Auto-fit sizing distributes space **evenly based on the size of the content** that is within a column or row. In other words, "Auto" means that the size of an element in a row will determine the height of that row in Grid.

### Combining

You can mix auto-fit, fixed widths and proportional widths. In that case, the proportional columns are apportioned to the remainder after the auto-fit and fixed widths have been calculated:

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" /> <!-- Auto-fit to content, 'Hi' -->

<ColumnDefinition Width="50.5" /> <!-- Fixed width: 50.5 pixels -->

<ColumnDefinition Width="69\*" /> <!-- Take 69% of remainder -->

<ColumnDefinition Width="31\*"/> <!-- Take 31% of remainder -->

</Grid.ColumnDefinitions>

<TextBlock Text="Hi" Grid.Column="0" />

IMG_258

## Allowing Users to Resize WPF Grid Rows

What if you want to allow the user to change the Grid Panel? This is where the **GridSplitter** control comes into play.

The GridSplitter is used simply by adding it to a column or a row in a Grid, with the proper amount of space for it (e.g. 5 pixels). It will then allow the user to drag it from side to side or up and down, while changing the size of the column or row on each of the sides of it.

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="5" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<TextBlock FontSize="55" TextWrapping="Wrap"

HorizontalAlignment="Center" VerticalAlignment="Center" >

Left side</TextBlock>

<GridSplitter Grid.Column="1" Width="5" HorizontalAlignment="Stretch" />

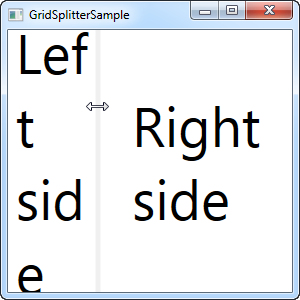
<TextBlock Grid.Column="2" FontSize="55" TextWrapping="Wrap"

HorizontalAlignment="Center" VerticalAlignment="Center" >

Right side</TextBlock>

</Grid>

Output:



# Stack Panels

Stack Panel acts like a stack of **things placed one after another** either horizontally or vertically. Unlike Grid, you cannot access particular place in a Stack Panel; every next element will be placed after one another in a sequence.

## Without a Stack Panel

<Canvas>

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="80" Height="80" Fill="Orange" />

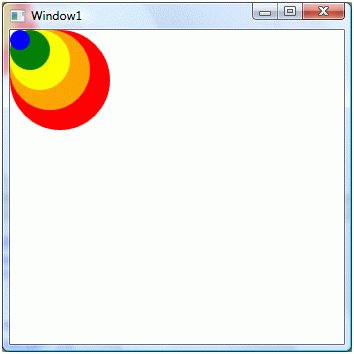
<Ellipse Width="60" Height="60" Fill="Yellow" />

<Ellipse Width="40" Height="40" Fill="Green" />

<Ellipse Width="20" Height="20" Fill="Blue" />

</Canvas>

Output:



## With a Stack Panel

<StackPanel>

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="80" Height="80" Fill="Orange" />

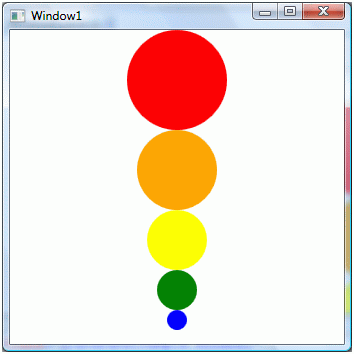
<Ellipse Width="60" Height="60" Fill="Yellow" />

<Ellipse Width="40" Height="40" Fill="Green" />

<Ellipse Width="20" Height="20" Fill="Blue" />

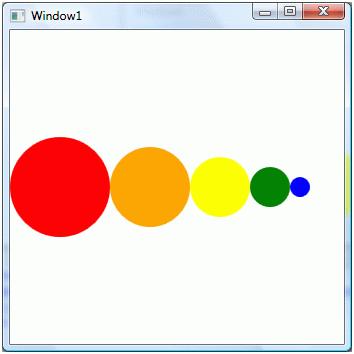
</StackPanel>

Output:



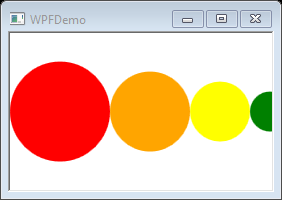
Now let's change the Orientation property to horizontal by:

<StackPanel Orientation="Horizontal" >



**Notes:**

If child elements on a Stack Panel do not fit in the Stack Panel area, they go outside of the visible area, like this:



In this case, you have two solutions:

* Use [Wrap Panel](#_Wrap_Panel) to wrap child elements.
* Add a scrolling feature to a Stack Panel, using CanHorizontallyScroll and CanVerticallyScroll properties.

## Stretching in Stack Panels

**Without stretching:**

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPFDemo"

Width="300"

Height="300"

WindowStyle="ThreeDBorderWindow">

<Grid>

<StackPanel

Height="200" Width="200" Margin="0,0,0,0"

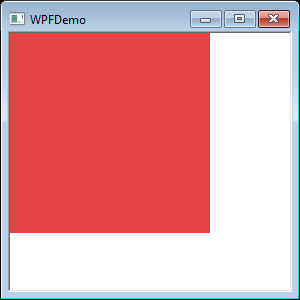
HorizontalAlignment="Left" VerticalAlignment="Top"

Background="#FFE44343" />

</Grid>

</Window>

Ouput:



**With stretching:**

Now if we want to fill a parent control or Window with a Stack Panel, you set VerticalAlignment and HorizontalAlignment properties to “Stretch”, and Width and Height properties to “Auto”.

<Grid>

<StackPanel

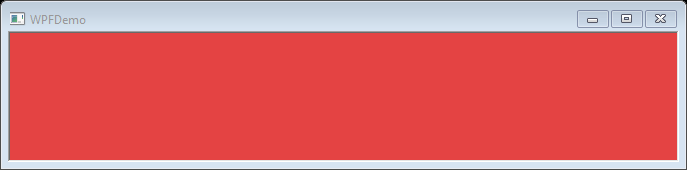
Width="Auto" Height="Auto" Margin="0,0,0,0"

HorizontalAlignment="Stretch" VerticalAlignment="Stretch"

Background="#FFE44343" />

</Grid>

Output:



Now when you resize the window, the Stack Panel will be resized too.

# Wrap Panel

A Wrap Panel and a Stack Panel share many similarities. In Wrap Panel, child elements are positioned in sequential order, from left to right or from top to bottom based on the orientation property. The only difference is that while Stack Panel stacks all the child elements in a single line, **Wrap Panel wraps the remaining elements to another line if there is no space left**.

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPFDemo"

Width="400"

Height="200"

WindowStyle="ThreeDBorderWindow">

<WrapPanel Width="400" Height="150" Background="LightBlue" Orientation="Horizontal" >

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="80" Height="80" Fill="Orange" />

<Ellipse Width="60" Height="60" Fill="Yellow" />

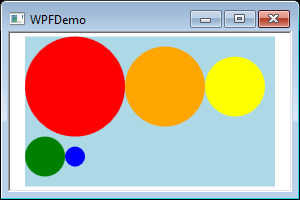
<Ellipse Width="40" Height="40" Fill="Green" />

<Ellipse Width="20" Height="20" Fill="Blue" />

</WrapPanel>

</Window>

Now if you resize the window and child elements do not fit in the Wrap Panel area, remaining elements automatically go to another line, like this:



# Dock Panel

A Dock Panel is used to dock child elements in the **left, right, top, and bottom positions** of the relative elements **regardless its size**.

For example:

<DockPanel LastChildFill = "True">

<Button Name="TopBtn" Content="Top" Background="LightGreen"

DockPanel.Dock="Top" Height="50" />

<Button Name="LeftBtn" Content="Left" Background="LightBlue"

DockPanel.Dock="Left" Width="50" />

<Button Name="RightBtn" Content="Right" Background="LightSalmon"

DockPanel.Dock="Right" Width="50" />

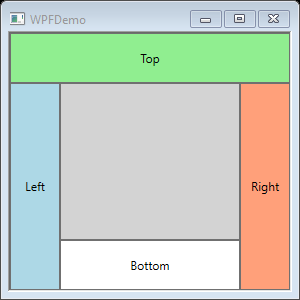
<Button Name="BottomBtn" Content="Bottom" Background="White"

DockPanel.Dock="Bottom" Height="50" />

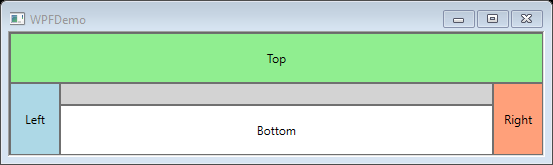
<Button Name="CenterFill" Background="LightGray" />

</DockPanel>

Output:



The child elements are automatically resized when the user resizes the window:



The position of child elements is determined by the DockPanel.Dock property of the respective child elements and the relative order of those child elements. The DockPanel.Dock property is an enumeration that has Left (default), Right, Top, and Bottom values.

With LastChildFill property, the last child element fills the remaining space regardless of any other dock value. By default, this property is True.

# Canvas Panel

A Canvas panel is used to position child elements by **using coordinates** that are relative to the canvas area.

Typically, a Canvas is used for 2D graphic elements (such as Ellipse, Rectangle, etc.), but **NOT for UI elements** because specifying absolute coordinates create trouble while resizing, localizing or scaling your XAML application.

## Basic Properties of Canvas Panels

Here are some of the properties of Canvas panels.

* The default values of Canvas.Height and Canvas.Width properties of a Canvas are 0. If you do not set these values, you will not see a canvas unless child elements are automatically resizable.
* Child elements on a Canvas are never resized.
* The vertical and horizontal alignments on child elements do not work. Child elements are placed on positions set by the Canvas.Left, Canvas.Top, Canvas.Right, and Canvas.Bottom properties.
* Margin does work partially. If Canvas.Left property of Canvas is set, Canvas.Right property does not work. If Canvas.Top property of Canvas is set, Canvas.Bottom property does not work.

Example:

<Canvas Background="LightCyan">

<Rectangle Canvas.Left="10" Canvas.Top="10" Height="200" Width="200"

Stroke="Black" StrokeThickness="10" Fill="Red" />

<Rectangle Canvas.Left="60" Canvas.Top="60" Height="200" Width="200"

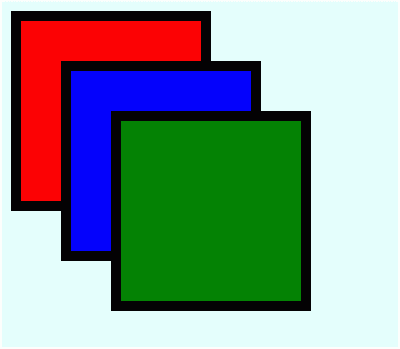
Stroke="Black" StrokeThickness="10" Fill="Blue" />

<Rectangle Canvas.Left="110" Canvas.Top="110" Height="200" Width="200"

Stroke="Black" StrokeThickness="10" Fill="Green" />

</Canvas>

Output:



## Order of Controls in Canvas Panels

The z-order of a control determines whether the control is in front of or behind another overlapping control. The default z-order of controls is the order controls are created in. The ZIndex property of Canvas represents the z-order of a control. The maximum value of Canvas.ZIndex is 32766.

<Canvas Background="LightCyan">

<Rectangle Canvas.Left="10" Canvas.Top="10" Canvas.ZIndex="2"

Height="200" Width="200" Stroke="Black" StrokeThickness="10" Fill="Red" />

<Rectangle Canvas.Left="60" Canvas.Top="60" Canvas.ZIndex="1"

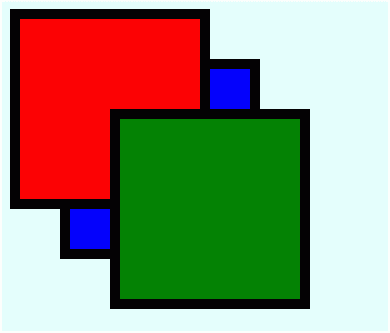
Height="200" Width="200" Stroke="Black" StrokeThickness="10" Fill="Blue" />

<Rectangle Canvas.Left="110" Canvas.Top="110" Canvas.ZIndex="3"

Height="200" Width="200" Stroke="Black" StrokeThickness="10" Fill="Green" />

</Canvas>

Output:



# Nesting of Layout

Nesting of layout means the use **layout panel inside another layout**, e.g. define stack panels inside a grid. This concept is widely used to take the advantages of multiple layouts in an application.

In the following example, we will be using stack panels inside a grid:

<Grid Background = "AntiqueWhite">

<Grid.RowDefinitions>

<RowDefinition Height = "\*" />

<RowDefinition Height = "\*" />

<RowDefinition Height = "\*" />

<RowDefinition Height = "\*" />

<RowDefinition Height = "\*" />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "\*" />

</Grid.ColumnDefinitions>

<Label Content = "Employee Info" FontSize = "15"

FontWeight = "Bold" Grid.Column = "0" Grid.Row = "0"/>

<StackPanel Grid.Column = "0" Grid.Row = "1" Orientation = "Horizontal">

<Label Content = "Name" VerticalAlignment = "Center" Width = "70"/>

<TextBox Name = "txtName" Text = "Muhammad Ali" VerticalAlignment = "Center"

Width = "200">

</TextBox>

</StackPanel>

<StackPanel Grid.Column = "0" Grid.Row = "2" Orientation = "Horizontal">

<Label Content = "ID" VerticalAlignment = "Center" Width = "70"/>

<TextBox Name = "txtCity" Text = "421" VerticalAlignment = "Center"

Width = "50">

</TextBox>

</StackPanel>

<StackPanel Grid.Column = "0" Grid.Row = "3" Orientation = "Horizontal">

<Label Content = "Age" VerticalAlignment = "Center" Width = "70"/>

<TextBox Name = "txtState" Text = "32" VerticalAlignment = "Center"

Width = "50"></TextBox>

</StackPanel>

<StackPanel Grid.Column = "0" Grid.Row = "4" Orientation = "Horizontal">

<Label Content = "Title" VerticalAlignment = "Center" Width = "70"/>

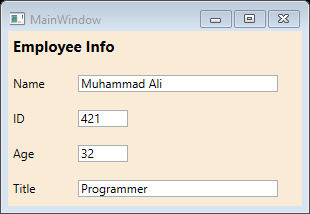
<TextBox Name = "txtCountry" Text = "Programmer" VerticalAlignment = "Center"

Width = "200"></TextBox>

</StackPanel>

</Grid>

Output:



# Data Binding

Data binding is a WPF technique that binds two data sources together and maintains synchronization of data. It allows the flow of data between UI elements and data object. **When data changes, it reflects the updates automatically to the UI elements and vice versa**. It is also possible to bind, not to a standard data source, but to another element on the page.

## First Example

**All the magic happens between the curly braces**, which in XAML encapsulates a Markup Extension. For data binding, we use the Binding extension to bind a property to another property on the data context. The simple syntax of binding is as below:

{Binding Path=PropertyToBindTo, ElementName=ElementToBindTo}

* The Path implies the **property** that you want to bind to. Note that it’s the default property of a binding, you may leave it out if you want to, like this: {Binding NameOfProperty}
* The ElementName allows us to connect directly to another **UI element** as the source. This element must be defined within the XAML for reference of the object.
* Other important properties: [UpdateSourceTrigger](https://www.wpf-tutorial.com/data-binding/the-update-source-trigger-property/), [Mode](#_Modes), [Converter](#_Value_Conversion), [StringFormat](#_StringFormat), etc.

The example below demonstrates a very simple data binding:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPFBinding"

Width="210" Height="110">

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock FontWeight="Bold" Text="Value: " />

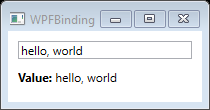
<TextBlock Text="{Binding Path=Text, ElementName=txtValue}" />

</WrapPanel>

</StackPanel>

</Window>

Output:



As you see, the TextBlock is automatically updated when you enter text into the TextBox.

Notes:

* Without binding, this would require us to listen to an **event** on the TextBox and then update the TextBlock each time the text changes.
* In this example, if you don’t want to define the binding using XAML, you can do this from Code-behind as [here](https://www.wpf-tutorial.com/data-binding/data-binding-via-code-behind/).

The source of databinding can be a normal .NET property or Dependency property. However, the target property must be a Dependency property.

For making binding work properly, both sides of the property must provide a change in notification which will tell the binding to update the target value. In normal .NET properties, it can be achieved by using INotifyPorpertyChanged interface. And in Dependency properties, it is done by PropertyChanged callback of property metadata.

## Binding Modes

Data binding has 4 types (described by Mode property):

* OneWay
* TwoWay
* OneTime
* OneWayToSource

### OneWay

In one-way binding, data is bound **from its source (the object holding the data) to its target (the object that displaying the data)**.

For example:

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="200" />

</Grid.ColumnDefinitions>

<Label Name="nameLabel" Grid.Row="0" Margin="2"> Name: </Label>

<TextBox

Name="nameText" Grid.Column="1" Margin="2"

Text="{Binding Name, Mode=OneWay}" />

<Label Name="ageLabel" Grid.Row="1" Margin="2"> Age: </Label>

<TextBox

Name="ageText" Grid.Row="1" Grid.Column="1" Margin="2"

Text="{Binding Age, Mode=OneWay}" />

<StackPanel Grid.Row="2" Grid.Column="1">

<Button Click="Button\_Click" Content="Show..." />

</StackPanel>

</Grid>

And here is C# code:

public partial class MainWindow : Window

{

Person person = new Person { Name = "Salman", Age = 26 };

public MainWindow()

{

InitializeComponent();

this.DataContext = person;

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

string message = person.Name + " is " + person.Age;

MessageBox.Show(message);

}

}

class Person

{

public string Name { set; get; }

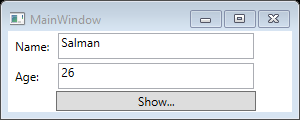
public double Age { set; get; }

}

In XAML code, we are respectively binding values of textboxes “Name” and “Age” to properties Name and Age (Name = "Salman", Age = 26) of class Person, but we have not selected what object (class) these properties belong to.

Then we assign an object of Person class to [DataContext](https://www.wpf-tutorial.com/data-binding/using-the-datacontext/)in the constructor of MainWindow. So the textbox can get or set the data context when it participates in data binding.

Output:



When clicking “Show…” button:



The problem with one-way data binding is that **despite which number we input to the “Age” textbox to, the message box always displays “Salman is 26”**.

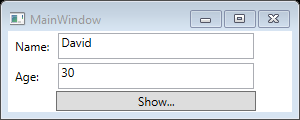
To show the updated data, we need to use two-way data binding.

### TwoWay

**In two-way binding, the user can modify the data through the user interface and have that data updated in the source**.

Let’s take the same example but here, we will change the binding mode from Mode=OneWay to Mode=TwoWay. Or you can remove the Mode property because it’s **TwoWay by default**.

Now when we change the Name and Age values:



The message box is updated:



## Binding To Your Own Data Objects

So far, we’ve created bindings between UI elements and existing classes, but in real life applications, you will also **create binding to your own data objects**. This is easy, but you might discover something disappointing you: Changes are not automatically reflected. As a result, you need a bit of extra work for this to happen.

There are two different scenarios that you may want to handle when dealing with data source changes: Changes to the list of items and changes in the bound properties in each of the data objects. How to handle them will vary, depending on what you're doing and what you're looking to accomplish, but WPF comes with two very easy solutions that you can use:

* The ObservableCollection (reflecting changes in the list data source)
* The INotifyPropertyChanged interface (reflecting changes in the data objects).

The following example shows you why we need these two things:

In XAML:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPFBinding" Width="300" Height="150">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">

<Button Name="btnAddUser" Click="btnAddUser\_Click">

Add user</Button>

<Button Name="btnChangeUser" Click="btnChangeUser\_Click" Margin="0,5">

Change user</Button>

<Button Name="btnDeleteUser" Click="btnDeleteUser\_Click">

Delete user</Button>

</StackPanel>

<ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>

</DockPanel>

</Window>

In CS:

using System.Windows;

using System.Collections.ObjectModel;

using System.ComponentModel;

namespace WPFDemo

{

public partial class MainWindow : Window

{

// List<T> vs ObservableCollection<T>:

// <https://stackoverflow.com/a/41680322>

// The ObservableCollection notifies any destination of changes to its content

public ObservableCollection<User> UserList;

public MainWindow()

{

InitializeComponent();

User user1 = new User { Name = "User 1" };

User user2 = new User { Name = "User 2" };

UserList = new ObservableCollection<User> { user1, user2 };

// Bind the ObservableCollection to item list of the listbox

lbUsers.ItemsSource = UserList;

}

private void btnAddUser\_Click(object sender, RoutedEventArgs e)

{

UserList.Add(new User { Name = "User 3"});

}

private void btnChangeUser\_Click(object sender, RoutedEventArgs e)

{

if (lbUsers.SelectedItem != null)

{

(lbUsers.SelectedItem as User).Name = "Replaced User";

}

}

private void btnDeleteUser\_Click(object sender, RoutedEventArgs e)

{

if (lbUsers.SelectedItem != null)

{

UserList.Remove(lbUsers.SelectedItem as User);

}

}

}

// If we replace the below User class by:

// public class User

// {

// public string Name { get; set; }

// }

// we cannot change user (after clicking "Change user" button) on GUI

// That's because "ObservableCollection represents a dynamic data collection that provides notifications

// when items get added, removed, or when the whole list is refreshed."

// NOT when changing value of a specific item.

public class User : INotifyPropertyChanged // Notifies clients that a property value has changed.

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

if (\_name != value)

{

\_name = value;

NotifyPropertyChanged("Name");

}

}

}

// Occurs when a property value changes.

// Implement INotifyPropertyChanged::PropertyChangedEventHandler PropertyChanged

public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(string propName)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

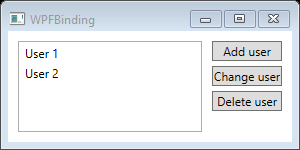
}

}

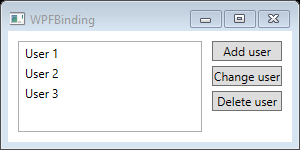
}

Output:

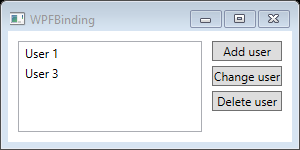
Initial state:



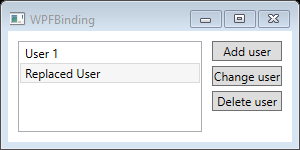
After clicking “Add user” button:



After selecting “User 2” item and clicking “Delete user” button:



After selecting “User 3” and clicking “Change user” button:



## Value Conversion

So far we’ve used simple data bindings, where the data types of sending and receiving properties were always compatible. However, you will soon run into situations where you want to **use a bound value of one type and then present it into another type**.

**When to use a value converter?**

Here are some examples:

* You want to check a CheckBox based on a value, but the value is a string like "yes" or "no" instead of a Boolean value.
* You have a file size in bytes, but you want to show it as bytes, kilobytes, megabytes or gigabytes based on how big it is.
* You can use a converter to generate an image for an ImageSource, based on the value, like a green sign for true or a red sign for false.
* Many more …

**How to use a value converter?**

You’ll need to write small classes, which implement the IValueConverter interface (or alternatively, the IMultiValueConverter interface). They’ll act like middlemen and translate a value between the source and the destination. So, in any situation where you need to transform a value before it reaches its destination or back to its source again, you likely need a converter.

Both IValueConverter and IMultiValueConverter interfaces just requires to implement 2 methods: Convert() and ConvertBack(). These methods will be used to convert the value to the destination format and then back again.

Note: There are more than [20 built-in converters](https://stackoverflow.com/questions/505397/built-in-wpf-ivalueconverters) you may take advantage of.

**Example**

Let's implement a simple converter which takes a string as input and then returns a Boolean value.

In XAML:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:self="clr-namespace:WPFDemo"

Title="ConverterSample" Width="250" Height="140">

<Window.Resources>

<self:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter" />

</Window.Resources>

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Current value is: " />

<TextBlock Text="{Binding ElementName=txtValue,

Path=Text,

Converter={StaticResource YesNoToBooleanConverter}}" />

</WrapPanel>

<CheckBox Content="Yes" IsChecked="{Binding ElementName=txtValue,

Path=Text,

Converter={StaticResource YesNoToBooleanConverter}}" />

</StackPanel>

</Window>

In CS:

using System;

using System.Windows;

using System.Windows.Data;

using System.Globalization;

namespace WPFDemo

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

}

public class YesNoToBooleanConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

switch (value.ToString().ToLower())

{

case "yes": return true;

case "no": return false;

}

return false;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is bool)

{

if ((bool)value == true) return "yes";

else return "no";

}

return "no";

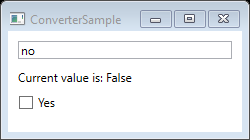
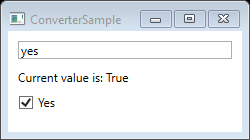
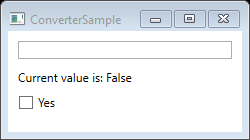
}

}

}

Output:

The TextBlock and CheckBox reflect changes when we type ‘yes’ or ‘no’ in the TextBox. The TextBlock and TextBox also reflect changes when we tick/untick the CheckBox:



## StringFormat

The cool thing about the converters is that they allow you to convert any data type into a completely different data type. However, for more simple usage scenarios, where you just want to change the way a certain value is shown and not necessarily convert it into a different type, the StringFormat property might very well be enough.

<https://www.wpf-tutorial.com/data-binding/the-stringformat-property/>

# Resources

Resources are normally definitions connected with some object to **store data** locally for controls or for the current window, or globally for the entire application.

Defining an object as a resource allows us to access it from another place. This means that the object can be **reused**. Resources are defined in resource dictionaries and any object can be defined as a resource effectively making it a shareable asset.

Two types of resources:

* StaticResource: is only assigned once and ignores any changes to resource dictionary.
* DynamicResource: works more like a data binding. It remembers that a property is associated with a particular resource key. If the object associated with that key changes, dynamic resource will update the target property.

## Example

In XAML:

<Window.Resources>

<SolidColorBrush x:Key="brushColor" Color="Blue" />

</Window.Resources>

<StackPanel>

<Rectangle

x:Name="topRect"

Height="50"

Margin="20"

Fill="{StaticResource brushColor}" />

<Rectangle

x:Name="bottomRect"

Height="50"

Margin="20"

Fill="{DynamicResource brushColor}" />

<Button

x:Name="changeResourceButton"

Margin="20"

Click="changeResourceButton\_Click"

Content="Change Resource" />

</StackPanel>

In C# code:

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void changeResourceButton\_Click(object sender, RoutedEventArgs e)

{

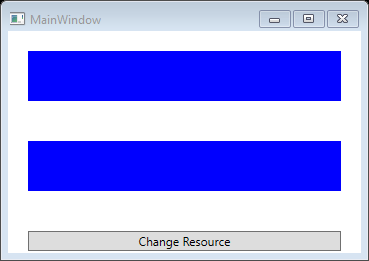
SolidColorBrush color = new SolidColorBrush(Color.FromRgb(255,0,0)); // Red

this.bottomRect.Resources["brushColor"] = color;

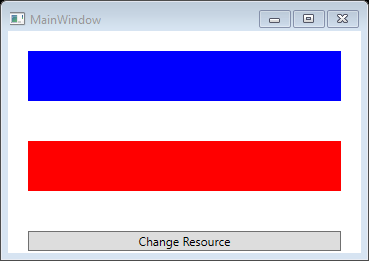
}

}

Output:



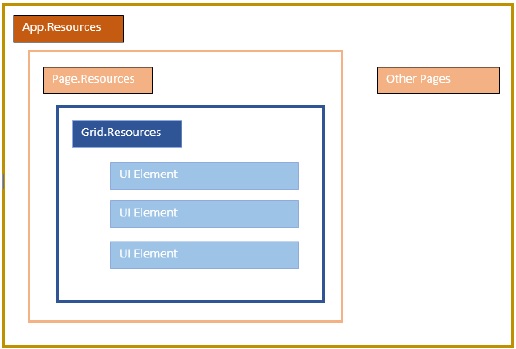
After clicking “Change Resource” button:



## Resource Scope

Resources are defined in resource dictionaries, but there are numerous places where a resource dictionary can be defined. **In what dictionary a resource is defined immediately limits the scope of that resource**.

* **On Window/page level**: It's accessible by all elements on that window/page. (above example)
* **In App.xaml**: It's the root of our application, so the resources defined here are scoped to the entire application.
* **In the resource dictionary of a grid**: It's accessible by that grid and its child elements only.
* **In the resource dictionary of a StackPanel**: It's accessible by that StackPanel and its child elements only.
* **...**



# XAML Debugging

<https://dailydotnettips.com/enabling-live-xaml-debugging-in-visual-studio>

<https://devblogs.microsoft.com/visualstudio/introducing-the-ui-debugging-tools-for-xaml>

<https://docs.microsoft.com/en-us/visualstudio/debugger/inspect-xaml-properties-while-debugging?view=vs-2019>

<http://www.wpftutorial.net/DebugDataBinding.html>

<https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-display-wpf-trace-information?view=vs-2019>

<https://www.wpf-tutorial.com/data-binding/debugging/>

<https://spin.atomicobject.com/2013/12/11/wpf-data-binding-debug/>

## Display WPF Trace Information

Visual Studio can receive debug trace information from WPF applications and display it in the Output window. To display debug trace information, [WPF tracing must be enabled](https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-display-wpf-trace-information?view=vs-2019).

There are some ways to enable WPF tracing:

### Configure VS to display trace info and its level of details to the Output window

<https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-display-wpf-trace-information?view=vs-2019>

### Use PresentationTraceSources Class

In the below sample, TextBlock has a missing data context. In this situation, you will not get any error in the VS Output window.

<Window x:Class="WpfApp.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="350" Width="525">

<Grid>

<TextBlock Text="{Binding ThereIsNoDataContext}"/>

</Grid>

</Window>

To enable tracing, I added a new XML namespace to include the System.Diagnostics namespace. You can also set the level of tracing to High, Medium, Low, or None. Now let’s add some tracing to the output window to see what is wrong with the data binding.

<Window x:Class="WpfApp.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase"

Title="MainWindow" Height="350" Width="525">

<Grid>

<TextBlock Text="{Binding ThereIsNoDataContext,

diag:PresentationTraceSources.TraceLevel=High}"/>

</Grid>

</Window>

Now the output window will contain the following helpful information:

System.Windows.Data Warning: 71 : BindingExpression (hash=38600745): DataContext is null

## Attach a Value Converter to Break into the Debugger

When you don’t see anything displayed in your UI, it is hard to tell whether it’s data binding causing your issue or a problem with the visual layout of the control. You can eliminate the data binding as the problem by adding a value converter and break into the debugger. If the value is what you expected, then data binding is not your issue.

Here is a simple value converter that breaks into the debugger.

In CS:

using System;

using System.Diagnostics;

using System.Globalization;

using System.Windows.Data;

namespace WpfApp

{

public class DebugDataBindingConverter : IValueConverter

{

public object Convert(object value, Type targetType,

object parameter, CultureInfo culture)

{

Debugger.Break();

return value;

}

public object ConvertBack(object value, Type targetType,

object parameter, CultureInfo culture)

{

Debugger.Break();

return value;

}

}

}

In XAML:

<Window x:Class="WpfApp.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:WpfApp"

Title="MainWindow" Height="350" Width="525">

<Grid Name="root">

<Grid.Resources>

<local:DebugDataBindingConverter x:Key="DebugBinding"/>

</Grid.Resources>

<TextBlock Text="{Binding ActualWidth, ElementName=root,

Converter = {StaticResource DebugBinding}}"/>

</Grid>

</Window>

## Know the Instant You Have a Data Binding Problem

Unless you are constantly checking every UI element and monitoring the Output window for binding errors, you will not always catch that you have a data binding problem.

Wouldn’t it be nice if you break into the debugger the instant you have a data binding error? By adding our own implementation of a TraceListener that breaks into the debugger, we will get notified the next time we get a data binding error. I also added the default ConsoleTraceListener alongside our new DebugTraceListener, so that our previous examples of tracing output would not be broken.

public partial class App : Application

{

protected override void OnStartup(StartupEventArgs e)

{

PresentationTraceSources.Refresh();

PresentationTraceSources.DataBindingSource.Listeners.Add(new ConsoleTraceListener());

PresentationTraceSources.DataBindingSource.Listeners.Add(new DebugTraceListener());

PresentationTraceSources.DataBindingSource.Switch.Level =

SourceLevels.Warning | SourceLevels.Error;

base.OnStartup(e);

}

}

public class DebugTraceListener : TraceListener

{

public override void Write(string message)

{

}

public override void WriteLine(string message)

{

Debugger.Break();

}

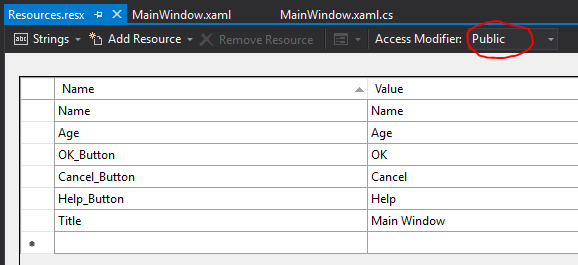
}

# Localization

Localization is the translation of application resources into localized versions for the specific cultures that the application supports.

In WPF, localizable applications are very easy to create with .resx file which is the simplest solution for localization. Let’s take a simple example to understand how it works:

1. Create a new WPF project with the name WPFLocalization.
2. In your solution explorer, you will see the Resources.resx file under Properties folder.
3. Open the file and **change the access modifier from internal to public so that it can be accessible in XAML file**.
4. Now add the following string’s name and values which we will be using in our application.

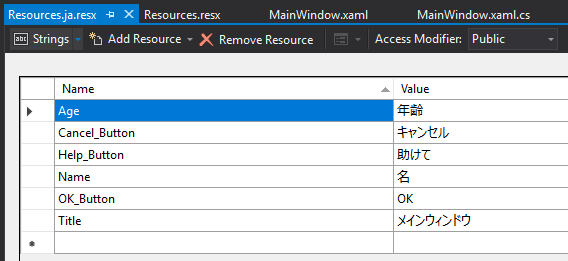


1. Make two copies (Ctrl+C and Ctrl+V) of Resources.resx file with the names Resources.en.resx and Resources.ja.resx.

**Notes:**

* These are naming conventions specific to language and country/region name, and it can be found on [National Language Support (NLS)](https://msdn.microsoft.com/en-us/goglobal/bb896001.aspx) API Reference.
* Although the default Resources.resx file may define strings in English, having a separate English resource file is necessary, which makes it much easier later to switch to the English language and back.

1. Change control’s names in Resources.ja.resx to Japanese:



1. Let’s go to the XAML file and create some controls.

First, add the namespace declaration to use localize resources:

xmlns:p="clr-namespace:WPFLocalization.Properties"

Then, set the properties of all the controls as shown below. In this example, we will not hard-code strings for the content of controls in the window. Instead, we will use strings defined in .resx files. For example, we define the window’s title in the XAML file like this: Title="{x:Static p:Resources.Title}"

Here is the XAML file in which controls are created and initialized with different properties:

<Window x:Class="WPFLocalization.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:WPFDemo"

xmlns:p="clr-namespace:WPFLocalization.Properties"

Title="{x:Static p:Resources.Title}" Height="350" Width="525">

<Grid>

<Label x:Name="label" Content="{x:Static p:Resources.Name}"

HorizontalAlignment="Left" VerticalAlignment="Top"

Margin="52,45,0,0" Width="86"/>

<TextBox x:Name="textBox" Margin="128,45,0,0"

HorizontalAlignment="Left" VerticalAlignment="Top"

Width="304" Height="23" />

<Label x:Name="label2" Content="{x:Static p:Resources.Age}"

HorizontalAlignment="Left" VerticalAlignment="Top"

Margin="52,95,0,0" Width="86"/>

<TextBox x:Name="textBox2" Margin="128,95,0,0"

HorizontalAlignment="Left" VerticalAlignment="Top"

Width="80" Height="23" />

<Button x:Name="button" Content="{x:Static p:Resources.OK\_Button}"

HorizontalAlignment="Left" VerticalAlignment="Top"

Margin="128,152,0,0" Width="75"/>

<Button x:Name="button1" Content="{x:Static p:Resources.Cancel\_Button}"

HorizontalAlignment="Left" VerticalAlignment="Top"

Margin="242,152,0,0" Width="75" />

<Button x:Name="button2" Content="{x:Static p:Resources.Help\_Button}"

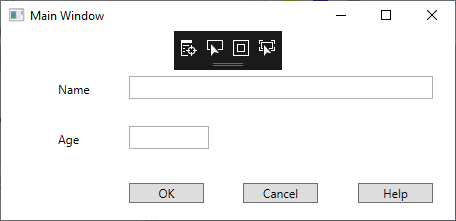
HorizontalAlignment="Left" VerticalAlignment="Top"

Margin="357,152,0,0" Width="75"/>

</Grid>

</Window>

Output:



By default, the program uses the default Resources.resx. If you want to show the text in Japanese language which are defined in Resources.ja.resx file, then you will need to set the culture explicitly when the program starts in App.xaml file as shown below.

using System;

using System.Windows;

using System.Globalization;

using System.Threading;

namespace WPFLocalization

{

public partial class App : Application

{

App()

{

Thread.CurrentThread.CurrentUICulture = new CultureInfo("ja");

// Thread.CurrentThread.CurrentUICulture = new CultureInfo("en");

}

}

}

After compile the application. The compiler and linker will create a separate satellite assembly for each culture. The satellite assemblies will be placed in sub-directories under the directory holding your main assembly. The sub-directories will be named by culture, allowing the .NET runtime to locate the resources appropriate to the culture in which the application runs. The main (default) resources file will be embedded in the main assembly.

├─bin

│ ├─Debug

│ │ ├─en

│ │ └─ja

Read more: [Dynamic Localization In WPF](https://www.c-sharpcorner.com/article/dynamic-localization-in-wpf/)

# WPF Commands

Talking about how to handle events (e.g. when the user clicks on a button or a menu item), it's typical for a function to be reachable from several places though, invoked by different user actions.

For example, if you have a typical interface with a main menu and a set of toolbars, **an action (e.g. New or Open) might be available in the menu, on the toolbar, in a context menu, and from a keyboard shortcut** (e.g. Ctrl+N and Ctrl+O). Each of these actions needs to perform what is typically the **exact same piece of code**.

In a WinForms app, you would have to define an event for each of them and then call a common function. With the above example, that would lead to at least three event handlers and some code to handle the keyboard shortcut. Not an ideal situation!

## Commands

With WPF, Microsoft is trying to remedy that with a concept called commands. Below are some great benefits of using commands in WPF:

* Commands allow to define actions in one place and then refer to them from all UI controls. That means **only one event handler is enough for all UI controls of the same function**.
* **Disabling/enabling UI controls of the same function becomes centralized among various UI controls**. With one method, you decide whether or not a given command can be executed, and then WPF toggles all the subscribing UI controls on or off automatically.
* You can [create your own commands](#_Event_Handler_(both). But to make it easier, MS has defined **over 100 commonly used commands** **for** you.  The command library consists of the following classes: [ApplicationCommands](https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.applicationcommands), [MediaCommands](https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.mediacommands), [EditingCommands](https://docs.microsoft.com/en-us/dotnet/api/system.windows.documents.editingcommands), [NavigationCommands](https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.navigationcommands) and [ComponentCommands](https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.componentcommands).

In a WinForms app, you would have to write code to disable UI elements when the action was not available. For instance, if the Cut function is only enabled when text was selected, you would have to manually enable/disable the main menu item, the toolbar button and the context menu item each time the text selection changed.

## Command Bindings

Commands don't actually do anything by themselves. At the root, they consist of the ICommand interface, which only defines an event and two methods:

* CanExecuteChanged event: raised when changes occur that affect whether or not the command should execute.
* Execute(): performs the actual action.
* CanExecute(): determines whether the action is available.

To perform the actual action of the command, you need a link between the command and your code and this is where the Window.CommandBindings comes into play.

You don't have to call these methods to have your UI elements updated - WPF does it automatically when the application has an idle moment, making sure that you interface remains updated all the time.

## Examples

### Self-Defined Event Handler

In XAML:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPF Commands"

Width="200"

Height="150">

<Window.CommandBindings>

<CommandBinding

CanExecute="Command\_Cut\_CanExecute"

Executed="Command\_Cut\_Executed"

Command="ApplicationCommands.Cut" />

<CommandBinding

CanExecute="Command\_Paste\_CanExecute"

Executed="Command\_Paste\_Executed"

Command="ApplicationCommands.Paste" />

</Window.CommandBindings>

<StackPanel >

<Menu DockPanel.Dock="Top" >

<MenuItem Header="\_File" >

<MenuItem Header="\_Cut" Command="ApplicationCommands.Cut" />

<MenuItem Header="\_Paste" Command="ApplicationCommands.Paste" />

</MenuItem>

</Menu>

<TextBox Name="txtEditor" Height="70" VerticalAlignment="Stretch"

AcceptsReturn="True" TextWrapping="Wrap" />

<DockPanel >

<Button Width="60" DockPanel.Dock="Left"

Command="ApplicationCommands.Cut"> \_Cut

</Button>

<Button Width="60" DockPanel.Dock="Right"

Command="ApplicationCommands.Paste"> \_Paste

</Button>

</DockPanel>

</StackPanel>

</Window>

In CS file:

using System.Windows;

using System.Windows.Input;

namespace WPFDemo

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

// one event handler for both ‘Cut’ button and ‘Cut’ menu item

private void Command\_Cut\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = (txtEditor != null) && (txtEditor.SelectionLength > 0);

}

private void Command\_Cut\_Executed(object sender, ExecutedRoutedEventArgs e)

{

txtEditor.Cut();

}

// one event handler for both ‘Paste’ button and ‘Paste’ menu item

private void Command\_Paste\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = Clipboard.ContainsText();

}

private void Command\_Paste\_Executed(object sender, ExecutedRoutedEventArgs e)

{

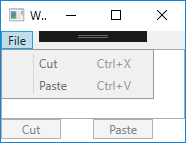
txtEditor.Paste();

}

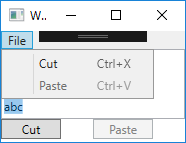
}

}

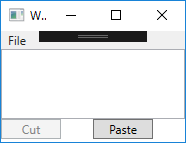
Output:



After inputing and selecting some text, the ‘Cut’ button and ‘Cut’ menu item are enabled:



After clicking ‘Cut’ button, the ‘Paste’ button and ‘Paste’ menu item are enabled:



Note: Because Cut and Paste are pre-defined commands from ApplicationCommands class, we can use shortcut **Ctrl+X** and **Ctrl+V**.

So, only one event handler is needed for 3 UI elements: button, menu item and shortcut.

### Pre-Defined Event Handler

We could **avoid all of the code-behind in the previous example**, because a WPF TextBox can automatically handle common commands like Cut, Copy, Paste, Undo and Redo by automatically handling the Executed() and CanExecute() methods for you

This is basically what we did in the previous example. But if you just want the basic behavior, you can let WPF **connect the commands and the TextBox control** and do the work for you using the CommandTarget.

For example:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="WPF Commands"

Width="200"

Height="150">

<StackPanel >

<Menu DockPanel.Dock="Top" >

<MenuItem Header="\_File" >

<MenuItem Header="\_Cut" Command="ApplicationCommands.Cut" />

<MenuItem Header="\_Paste" Command="ApplicationCommands.Paste" />

</MenuItem>

</Menu>

<TextBox Name="txtEditor" Height="70" VerticalAlignment="Stretch"

AcceptsReturn="True" TextWrapping="Wrap" />

<DockPanel >

<Button Width="60" DockPanel.Dock="Left"

Command="ApplicationCommands.Cut"

CommandTarget="{Binding ElementName=txtEditor}"> \_Cut

</Button>

<Button Width="60" DockPanel.Dock="Right"

Command="ApplicationCommands.Paste"

CommandTarget="{Binding ElementName=txtEditor}"> \_Paste

</Button>

</DockPanel>

</StackPanel>

</Window>

Note: No even hander implementation in cs file is needed.

Output:

Same as above example

Read more: <https://wpf.2000things.com/tag/commands/>

# WPF Shortcuts

<https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.inputgesture?view=netframework-4.8>

unlike with WinForms, WPF is not listening for keyboard shortcuts automatically if you assign them to e.g. a menu item - you will have to do that manually.

However, when using commands, WPF is all ears and will respond to keyboard shortcuts automatically.

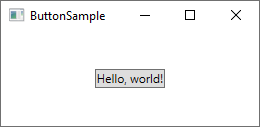
# WPF Controls

## Button

### A Simple Button

<Button>Hello, world!</Button>

Output:



Now let's make the button do something, by subscribing to its Click event

<Button Click="HelloWorldButton\_Click">Hello, World!</Button>

In Code-behind, you will need a matching method to handle the click:

private void HelloWorldButton\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Hello, world!");

}

Output:

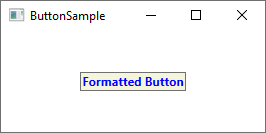
You now have a very basic button and when you click on it, a message will be displayed!

### Buttons with Formatted Content

You can find several properties on the Button control for doing formating content, such as Foreground, Background, FontWeight and so on. For example:

<Button Background="Beige" Foreground="Blue" FontWeight="Bold">Formatted Button</Button>

Output:



### Buttons with Advanced Formatted Content

One of the very cool things about WPF is the ability to replace simple text inside a control with other controls. This means that you don't have to limit your buttons to simple text, formatted in the same way - you can add several text controls with different formatting. For example:

<Button>

<StackPanel Orientation="Horizontal">

<TextBlock>Formatted</TextBlock>

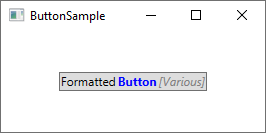
<TextBlock Foreground="Blue" FontWeight="Bold" Margin="2,0">Button</TextBlock>

<TextBlock Foreground="Gray" FontStyle="Italic">[Various]</TextBlock>

</StackPanel>

</Button>

Output:



### Buttons with Images

The ImageButton is a button which allows you to include an image before the text. In WPF, there's no need for a separate control to accomplish this - as you just saw, we **can put several controls inside a Button**. So you can easily add an Image control to it, like this:

<Button Padding="5">

<StackPanel Orientation="Horizontal">

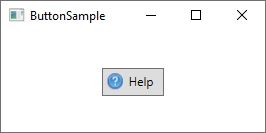
<Image Source="/WpfTutorialSamples;component/Images/help.png" />

<TextBlock Margin="5,0">Help</TextBlock>

</StackPanel>

</Button>

Output:



### Button Padding

You may have noticed that buttons in the WPF framework doesn't come with any padding by default. This means that the text is very close to the borders, which might look a little bit strange, because most buttons found elsewhere (web, other applications etc.) do have at least some padding in the sides. No worries, because the Button comes with a Padding property:

<Button Padding="5,2">Hello, World!</Button>

This will apply a padding of 5 pixels on the sides, and 2 pixels in the top and bottom.

But having to apply padding to all of your buttons might get a bit tiresome at a certain point, so here's a small tip: You can apply the padding globally, either across the entire application or just this specific Window, using a Style. Here's an example where we apply it to the Window, using the Window.Resources:

<Window.Resources>

<Style TargetType="{x:Type Button}">

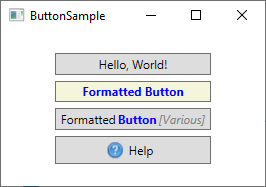
<Setter Property="Padding" Value="5,2"/>

</Style>

</Window.Resources>

Output:

This padding will now be applied to all your buttons, but you can of course override it by specifically defining the Padding property on a Button. Here's how all the buttons of this example look with the common padding:



## Menu

### A Simple Menu

<Window x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MenuSample" Height="200" Width="200">

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="\_File">

<MenuItem Header="\_New" />

<MenuItem Header="\_Open" />

<MenuItem Header="\_Save" />

<Separator />

<MenuItem Header="\_Exit" />

</MenuItem>

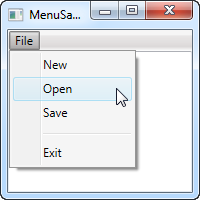
</Menu>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>

Output:



The above example creates a single top-level menu which is placed in the top of the window, with 4 child items and a separator. But we can place it wherever we like, and in any width or height.

You should notice the **underscore before the first character of each label, which tells WPF to use that character as the accelerator key**, which means that the user can press the *Alt* key followed by the given character, to activate the menu item. This works all the way from the top-level item and down the hierarchy, meaning that in this example I could press Alt, then F and then N, to activate the New item.

### Icons and Checkboxes

Two common features of a menu are the icon and the ability to have checkable menu items. For example:

<Window x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MenuIconCheckableSample" Height="150" Width="300">

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="\_File">

<MenuItem Header="\_Exit" />

</MenuItem>

<MenuItem Header="\_Tools">

<MenuItem Header="\_Manage users">

<MenuItem.Icon>

<Image Source="/WpfTutorialSamples;component/Images/user.png" />

</MenuItem.Icon>

</MenuItem>

<MenuItem Header="\_Show groups" IsCheckable="True" IsChecked="True" />

</MenuItem>

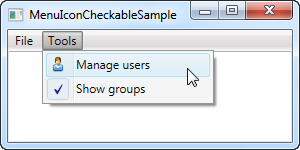
</Menu>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>

Output:



### Event Handler

The easiest way is to simply add a click event handler to the MenuItem, like this:

<MenuItem Header="\_New" Click="mnuNew\_Click" />

In code-behind, you will implement the mnuNew\_Click method, like this:

private void mnuNew\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("New");

}

This suffices for more simple applications, but **the WPF way is to use a** [**Command**](#_WPF_Commands) **for this**.

The below example demonstrates how to use pre-defined commands and how to **create your own commands**:

In XAML:

<Window

x:Class="WPFDemo.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:self="clr-namespace:WPFDemo"

Title="MenuWithCommandsSample"

Width="300" Height="200">

<Window.CommandBindings>

<!-- Without Command\_New\_CanExecute(), the commands are always available -->

<CommandBinding Command="self:CustomCommands.Exit" Executed="Command\_Exit\_Executed" />

</Window.CommandBindings>

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="File">

<!-- There is NO ApplicationCommands.Exit, so we have create it -->

<MenuItem Command="self:CustomCommands.Exit" />

</MenuItem>

<MenuItem Header="Edit">

<MenuItem Command="ApplicationCommands.Cut" />

<MenuItem Command="ApplicationCommands.Paste" />

</MenuItem>

</Menu>

<TextBox Name="txtEditor" AcceptsReturn="True" />

</DockPanel>

</Window>

In CS:

using System.Windows;

using System.Windows.Input;

namespace WPFDemo

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void Command\_Exit\_Executed(object sender, ExecutedRoutedEventArgs e)

{

Application.Current.Shutdown();

}

}

public static class CustomCommands

{

public static readonly RoutedUICommand Exit = new RoutedUICommand

(

"Exit", // Name of the menu item. With it, no need <MenuItem Header="Exit" />

"Exit",

typeof(CustomCommands),

new InputGestureCollection()

{

new KeyGesture(Key.F4, ModifierKeys.Alt)

}

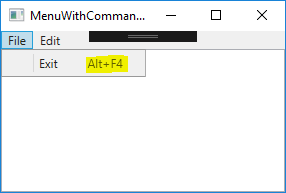
);

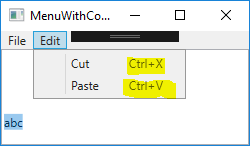
// Define more commands here, just like the one above

}

}

Output:





As you see, by using commands for menu items, **commonly used commands (e.g. Cut/Paste) in combination with their shortcuts and states (enable/disable) are handled automatically**. We only need to handle commands that WPF has no way of guessing what we want them to do (e.g. Exit).