



HÖGSKOLAN I BORÅS

Windows Presentation Foundation (WPF) 2

Troelsen Kapitel 25

Grundläggande applikationsutveckling med C#

Agenda

- WPF kontroller
- Layout Managers
- Layout Managers i Visual Studio
- WPF exempel 1 med kontroller & händelsehantering
- WPF Commands
- Routed Events
- WPF exempel 2 med kontroller & händelsehantering
- Databindning

Fundamentala WPF kontroller

- De fundamentala kontrollerna i WPF kategoriseras i kontroller för **användarinput (och output)**, **dekorationer**, **media** och **layout**.

WPF Control Category	Example Members	Meaning in Life
Core user input controls	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	WPF provides an entire family of controls you can use to build the crux of a user interface.
Window and control adornments	Menu, ToolBar, StatusBar, ToolTip, ProgressBar	You use these UI elements to decorate the frame of a Window object with input devices (such as the Menu) and user informational elements (e.g., StatusBar and ToolTip).
Media controls	Image, MediaElement, SoundPlayerAction	These controls provide support for audio/video playback and image display.
Layout controls	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	WPF provides numerous controls that allow you to group and organize other controls for the purpose of layout management.

WPF *Ink* kontroller

- WPF innehåller även kontroller för att fånga ***stylus*** input, t.ex. när man skapar applikationer för ***Tablet PC***, i det så kallade ***Ink*** API:t.
- Även icke ***touch*** skärmar kan använda applikationer skrivna med ***Ink*** API:t eftersom kontrollerna även kan fånga input från musen.
- Namespace **System.Windows.Ink** (i *PresentationCore.dll*) innehåller en del ***Ink*** typer såsom **Stroke** och **StrokeCollection**, men de flesta ***Ink*** typerna finns i namespace **System.Windows.Controls** (i *PresentationFramework.dll*), såsom **InkCanvas** och **InkPresenter**.

WPF *Dokument* kontroller

- WPF innehåller kontroller för **avancerad dokument processering** i namespace **System.Windows.Documents** (i *PresentationFramework.dll*).
 - Med typerna i namespace **System.Windows.Documents** kan man skapa applikationer med Adobe PDF-liknande funktionalitet, med bl.a. support för utskrifter, zooming, sökning, *user annotations (sticky notes)*
 - Dock används inte Adobe PDF API:t “under huven”, utan Microsofts **XML Paper Specification (XPS)** API.
- Det finns många gratis addins som kan konvertera mellan PDF och XPS formaten, samt andra PDF-bibliotek man kan använda istället, t.ex:
 - iText 7 (kommerciell eller AGLP licens, dvs restriktioner finns när man använder produkten)
<https://www.nuget.org/packages/itext7>
<https://itextpdf.com/en/resources>
<https://itextpdf.com/en/how-buy>
 - iTextSharp LGPL 4.1.6 (gammal version av iText med LGPL licens, dvs kan använda produkten gratis)
<https://www.nuget.org/packages/iTextSharp-LGPL>
 - PDFSharp (open source med MIT licens, dvs kan använda produkten gratis)
<https://www.nuget.org/packages/PdfSharp/1.51.5185-beta>
<http://www.pdfsharp.net>

WPF *Dialog* kontroller

- WPF innehåller även en del **dialoger** som ofta används i applikationer, såsom **OpenFileDialog** och **SaveFileDialog**. Dialogerna är definierade i namespace **Microsoft.Win32** (i *PresentationFramework.dll*) assembly. I princip, skapar man en instans av en **dialog** klass och anropar dess **ShowDialog()** metod.

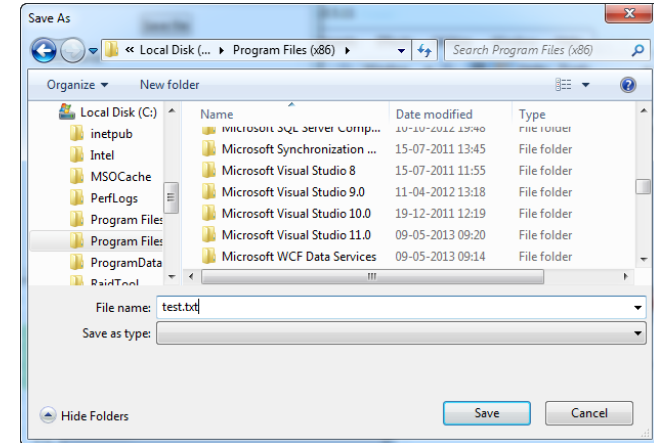
<https://www.wpf-tutorial.com/dialogs/the-openfiledialog>

<https://www.wpf-tutorial.com/dialogs/the-savefiledialog>

```
using Microsoft.Win32;

//omitted for brevity

private void btnShowDlg_Click(object sender, RoutedEventArgs e)
{
    // Show a file save dialog.
    SaveFileDialog saveDlg = new SaveFileDialog();
    saveDlg.ShowDialog();
}
```

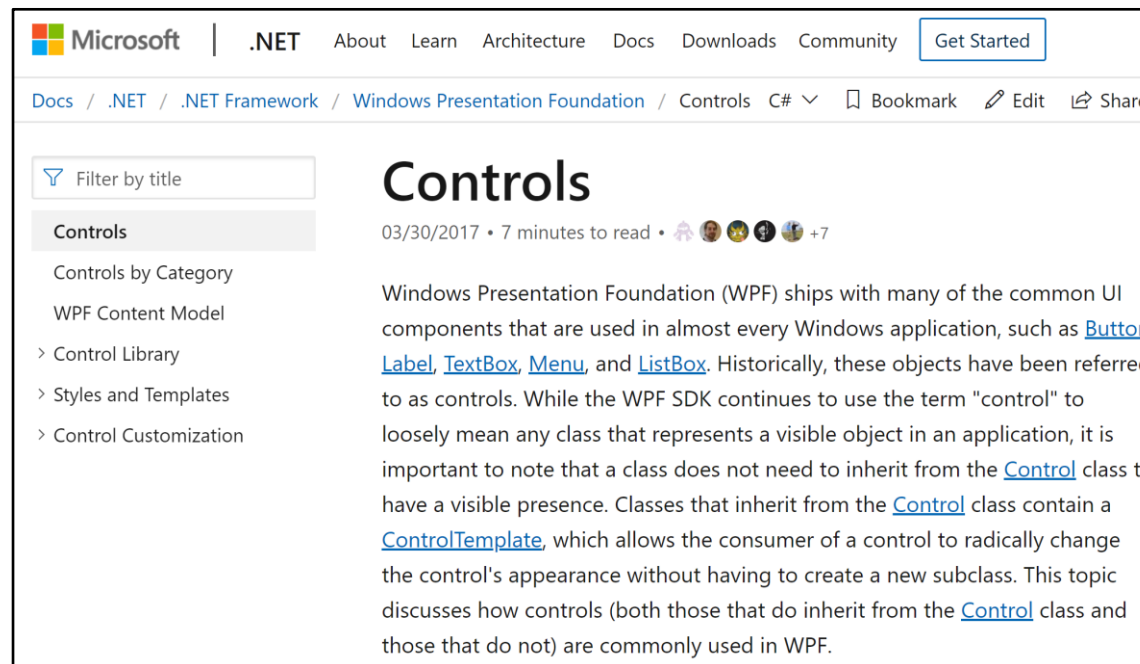


- Dialogklasserna innehåller medlemmar som kan användas för att bl.a. filtrera vilka typer av filer som skall visas i dialogen samt för att ställa in sökvägen till en folder.
- Man kan också skapa egendefinierade dialoger för att inhämta användarinput:
<https://www.wpf-tutorial.com/dialogs/creating-a-custom-input-dialog>

Dokumentationen för WPF kontroller

- Det finns många olika typer av WPF kontroller, där varje kontroll har många medlemmar. För att lära sig en WPF kontroll, är Microsofts WPF dokumentation en bra resurs, som även innehåller många exempel (i både C# och XAML) samt arvshierarkin för alla WPF kontroller.

<https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/index>



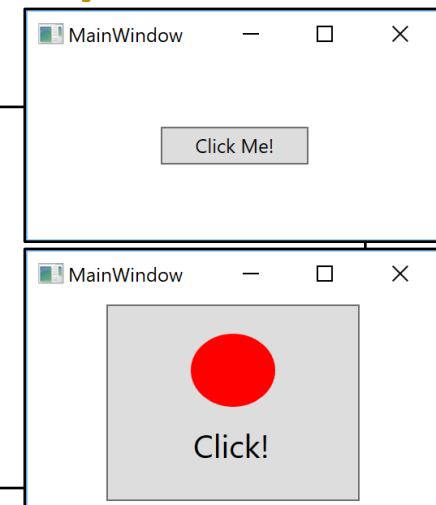
Att hantera WPF kontroller i Visual Studio

- När man placerar en WPF kontroll på Visual Studios designyta är det alltid bra att namnge kontrollen med **x:Name** propertyn (via **Properties** fönstret eller direkt i **XAML** dokumentet) vilket gör kontrollen tillgänglig i tillhörande C# **code behind** fil. Dessutom kan **Events** fliken i **Properties** fönstret användas för att snabbt skapa **händelsehanterare** för vald kontroll.

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140" Click="btnMyButton_Click" />
```

- I ovanstående kod har en knappns (**Button**) **Content** property tilldelats en simpel sträng **Click Me!**, men WPFs **content modell** möjliggör tilldelning av **komplex content** till en kontrolls **Content** property (*property-element syntax*).

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
    <Button.Content>
        <StackPanel Height="95" Width="128" Orientation="Vertical">
            <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
            <Label Width="59" FontSize="20" Content="Click!" Height="36" />
        </StackPanel>
    </Button.Content>
</Button>
```



Att hantera WPF kontroller i Visual Studio

- Barnelementet (i XAML) av en klass som ärver från **ContentControl** utgör, implicit, klassens **Content**, varför man inte, explicit, behöver ange ett **Content scope** (**Button.Content** i detta fallet) när man definierar **komplex content**.

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
    <StackPanel Height="95" Width="128" Orientation="Vertical">
        <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
        <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
</Button>
```

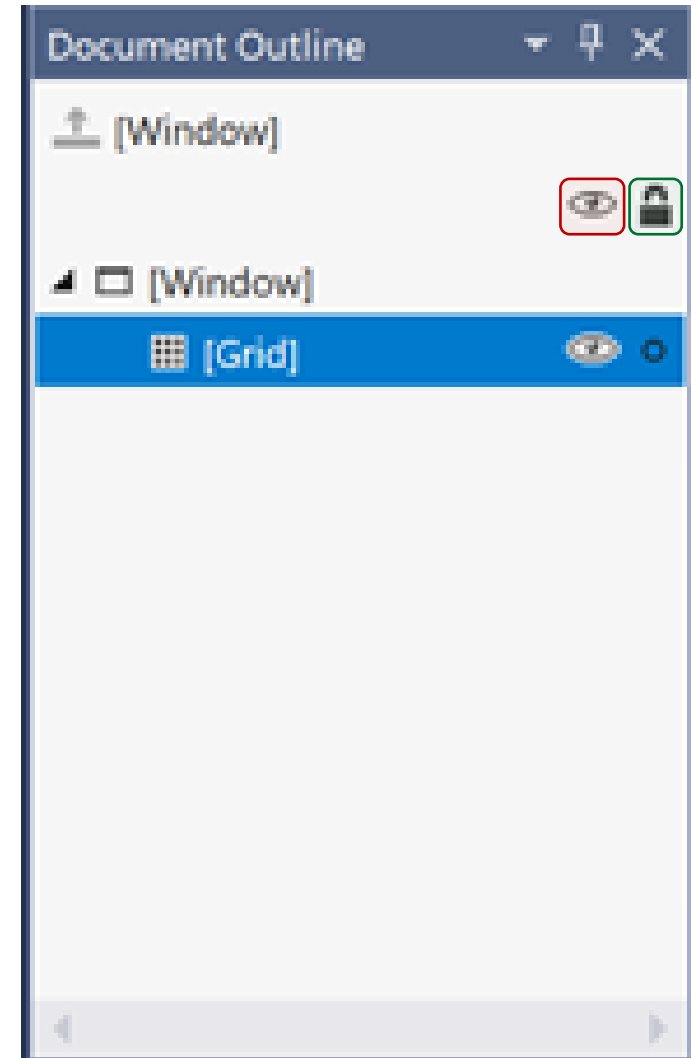
- I ovanstående kod utgörs det komplexa *content* av en **StackPanel** (en **layout manager**) med dess **kontroller** (komplex *content* kan även skapas med hjälp av Visual Studios designer).
- När man har skapat en **layout manager** för en kontrolls **Content**, kan man drag-and-drop:a andra kontroller till den i designern. Varje kontroll kan konfigureras med hjälp av **Properties** fönstret (**properties** och **events**). Om man exempelvis hanterar **Click** händelsen för **Button** kontrollen kommer IDEt att generera en tom **händelsehanterare** enligt nedan:

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
}
}
```

Document Outline editorn i Visual Studio

- **Document Outline** fönstret är användbart när man designar en kontroll som har komplex *content*. Det logiska XAML trädet visas i fönstret, och om man **klickar på en nod, selekteras aktuell kontroll i designern och i XAML editor** som då kan editeras.
- Till höger om en nod finns en **ikon som liknar ett öga**. Om man klickar på ikonen kan man **gömma eller visa aktuell kontroll i designern**, som gör det lättare att jobba med det grafiska gränssnittet när det innehåller många kontroller.
- Till höger om “ögat” finn en **ikon som liknar ett lås**. Om man klickar på ikonen kan man **låsa/låsa upp en kontroll för editering**, som kan användas om man inte vill ändra XAML för en kontroll av misstag.

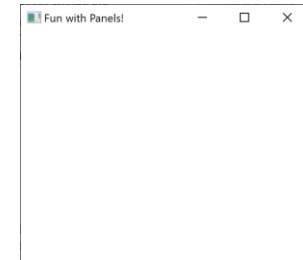
*OBS! Dessa knapparna/ikonerna för att gömma/visa och låsa/låsa upp en kontroll är **design-time restriktioner** och påverkar inte kontrollerna run-time.*



Layout paneler (*layout managers*)

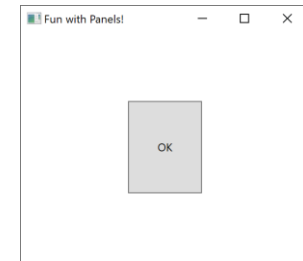
- En WPF applikation brukar innehålla många UI element som måste organiseras inom olika fönster. Dessutom måste kontrollerna bete sig på ett korrekt sätt när en användare ändrar storleken på ett fönster. Denna funktionalitet tillhandahålls av olika **panel typer** (kallas också **layout managers**).
- Som default, när man skapar ett **WPF Window** i Visual Studio, används en **Grid** som *layout manager* för fönstret. Om man tar bort **Grid layout managern**, ser fönstret ut enligt nedan.

```
<Window x:Class="MyWPFApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fun with Panels!" Height="285" Width="325">
    </Window>
```



- Om man placerar en kontroll direkt inuti ett fönster (dvs som fönstrets **Content** utan någon layout manager) kommer kontrollen att visas i fönstrets mitt. Exempelvis kommer nedanstående kod att placera en knapp (**Button**) mitt i fönstret. Knappens höjd och bredd sätts med **Height** och **Width**.

```
<!-- This button is in the center of the window at all times -->
<Window x:Class="MyWPFApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fun with Panels!" Height="285" Width="325">
    <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```



Layout paneler (*layout managers*)

- Om man försöker placera flera kontroller direkt i en kontrolls **Content**, fås ett kompileringsfel eftersom man endast kan tilldela **ett enda objekt** till en kontroll (som äver från **ContentControl**) **Content** property.

```
<!-- Error! Content property is implicitly set more than once! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <!-- Error! Two direct child elements of the <Window>! -->
  <Label x:Name="lblInstructions" Width="328" Height="27" FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>

</Window>
```

- När ett fönster (eller andra kontroller som ärver från **ContentControl**) behöver innehålla **flera kontroller**, måste kontrollerna placeras i ett eller flera **paneler** (*layout managers*). Därefter tilldelas **panelen** som det enda objektet till kontrollens **Content** property (layout managers kan innehålla godtyckligt många kontroller).

Layout paneler (*layout managers*)

- Namespace **System.Windows.Controls** innehåller flera olika typer av paneler, där **varje panel bestämmer hur dess inneslutna kontroller skall organiseras, och hur de skall bete sig om t.ex. användaren ändrar storleken på fönstret.**
- En panel kan även innehålla andra paneler (t.ex. en **DockPanel** som innehåller en **StackPanel**, osv.)

*Vi kikar
på dessa.*



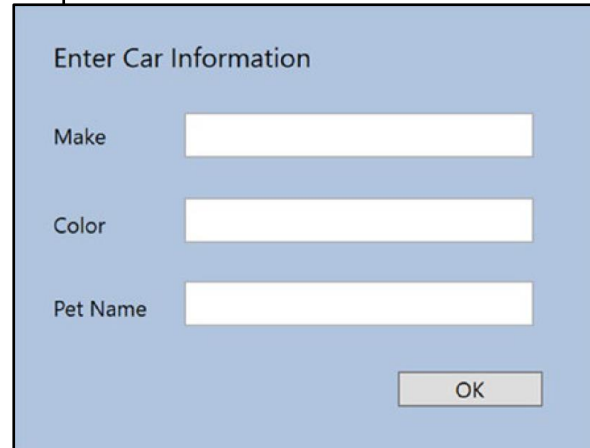
Panel Control	Meaning in Life
Canvas	Provides a classic mode of content placement. Items stay exactly where you put them at design time.
DockPanel	Locks content to a specified side of the panel (Top, Bottom, Left, or Right).
Grid	Arranges content within a series of cells, maintained within a tabular grid.
StackPanel	Stacks content in a vertical or horizontal manner, as dictated by the Orientation property.
WrapPanel	Positions content from left to right, breaking the content to the next line at the edge of the containing box. Subsequent ordering happens sequentially from top to bottom or from right to left, depending on the value of the Orientation property.

<https://www.wpf-tutorial.com/panels/introduction-to-wpf-panels>

Canvas panelen

- **Canvas** panelen möjliggör **absolut positionering av kontroller** (som i **Windows Forms**). Dock, om användaren minskar fönstrets storlek så att arean blir mindre än **Canvas** panelens storlek, kommer **Canvas** panelens kontroller inte att synas (men fönsterstorleken kan låsas av programmeraren).
- Man lägger till kontroller till en **Canvas** genom att ange dem inom **Canvas** kontrollens start- och sluttagg (i XAML). Därefter anger man det övre vänstra hörnet för varje kontroll med *attached properties* **Canvas.Top** och **Canvas.Left** (detta är var kontrollen skall renderas inom **Canvas** panelen). Man kan ange det nedre högra hörnet indirekt genom att sätta kontrollens **Height** och **Width** properties, eller via *attached properties* **Canvas.Right** och **Canvas.Bottom**.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203" Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
      Width="328" Height="27" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60" Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60" Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109" Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107" Width="193" Height="25"/>
    <Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155" Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153" Width="193" Height="25"/>
  </Canvas>
</ Window >
```



Enter Car Information

Make

Color

Pet Name

OK

Canvas panelen

- Ordningen i vilket kontrollerna läggs till i en **Canvas** spelar ingen roll för respektive kontrolls placering, utan detta bestäms av kontrollens storlek samt *attached properties* **Canvas.Top**, **Canvas.Bottom**, **Canvas.Left** och **Canvas.Right**.
- Om en kontroll inte specificerar en position med *attached property syntax* (dvs med **Canvas.Left** och **Canvas.Top**), kommer kontrollen att positioneras längst upp till vänster i **Canvas** panelen.
- Nackdelen med en **Canvas** är att dess kontroller inte dynamiskt ändrar storlek när *styles* (eller *templates*) appliceras till dem (t.ex. fontstorlek) samt att dess kontroller kan bli osynliga när användaren ändrar storleken på fönstret.
- En **Canvas** lämpar sig dock för att positionera *grafisk* innehåll, t.ex. om man skapar en egen bild med XAML (där man kanske vill att linjer, figurer och text skall stanna på samma position när användaren ändrar storleken på fönstret).

WrapPanel panelen

- En **WrapPanel** kommer att flytta omkring sina kontroller när fönstrets storlek ändras så att kontrollerna fortfarande är synliga. När man lägger till kontroller i en **WrapPanel** anger man inte **top**, **bottom**, **left** och **right** som med en **Canvas**, men man kan t.ex. ange kontrollens **Height** och **Width**. **Ordningen i vilket man lägger till kontroller till en WrapPanel spelar roll**, eftersom kontrollerna kommer att renderas från *första till sista* elementet.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <WrapPanel Background="LightSteelBlue">
    <Label x:Name="lblInstruction" Width="328" Height="27" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Content="Make"/>
    <TextBox x:Name="txtMake" Width="193" Height="25"/>
    <Label x:Name="lblColor" Content="Color"/>
    <TextBox x:Name="txtColor" Width="193" Height="25"/>
    <Label x:Name="lblPetName" Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Width="193" Height="25"/>
    <Button x:Name="btnOK" Width="80" Content="OK"/>
  </WrapPanel>
</Window>
```

Om en kontroll inte får plats på en rad,
"hoppas den ner" till nästa rad.

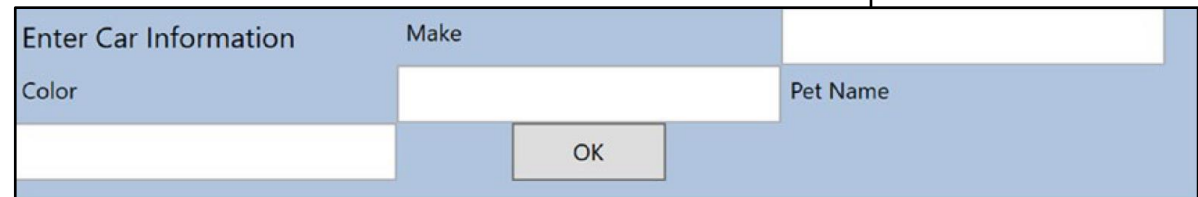
- Som *default* kommer kontrollerna att renderas från *vänster till höger*, men om man ändrar värdet av **Orientation** propertyn till **Vertical**, så renderas kontrollerna *uppfifrån och ner*.

```
<WrapPanel Background="LightSteelBlue" Orientation="Vertical">
```

WrapPanel panelen

- Man kan ange en **ItemWidth** och **ItemHeight** för en **WrapPanel** (och vissa andra panel typer) som bestämmer defaultstorleken på dess kontroller. Om en kontroll inte anger en egen **Height** och/eller **Width**, kommer den att positioneras utefter den relativa storleken som bestäms av **WrapPanel** panelens **ItemWidth** och **ItemHeight**.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="100" Width="650">
  <WrapPanel Background="LightSteelBlue" Orientation="Horizontal" ItemWidth="200" ItemHeight="30">
    <Label x:Name="lblInstruction" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Content="Make"/>
    <TextBox x:Name="txtMake"/>
    <Label x:Name="lblColor" Content="Color"/>
    <TextBox x:Name="txtColor"/>
    <Label x:Name="lblPetName" Content="Pet Name"/>
    <TextBox x:Name="txtPetName"/>
    <Button x:Name="btnOK" Width="80" Content="OK"/>
  </WrapPanel>
</Window>
```



- En **WrapPanel** är inte det bästa valet för att organisera kontroller direkt i fönstret eftersom kontrollerna flyttas omkring när användaren ändrar fönstrets storlek. Dock är en **WrapPanel** användbar som ett subelement till andra paneltyper, där en liten area i fönstret kan flytta omkring sina kontroller.

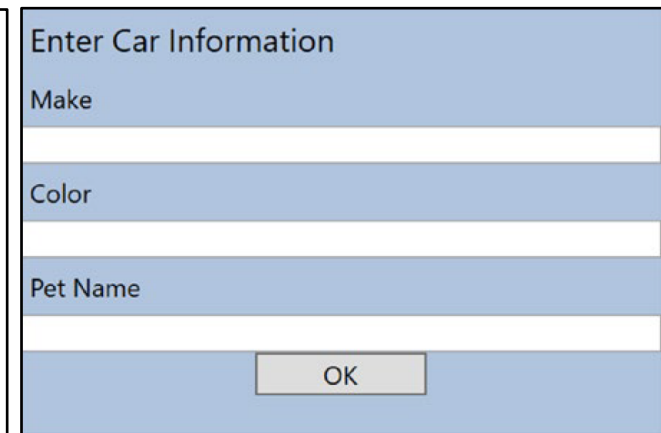
StackPanel panelen

- Som en **WrapPanel** kommer en **StackPanel** att organisera sina kontroller horisontellt från ***vänster till höger*** eller vertikalt ***uppfifrån och ner (default)*** beroende på vilket värde som tilldelas **Orientation** propriety. Dock kommer en **StackPanel** inte att flytta omkring sina kontroller när fönstrets storlek ändras, utan kontrollerna kommer istället att ***sträckas ut*** så att de fyller **StackPanelens** storlek.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="100" Width="650">

  <StackPanel Background="LightSteelBlue">
    <Label x:Name="lblInstruction" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Content="Make"/>
    <TextBox Name="txtMake"/>
    <Label x:Name="lblColor" Content="Color"/>
    <TextBox x:Name="txtColor"/>
    <Label x:Name="lblPetName" Content="Pet Name"/>
    <TextBox x:Name="txtPetName"/>
    <Button x:Name="btnOK" Width="80" Content="OK"/>
  </StackPanel>

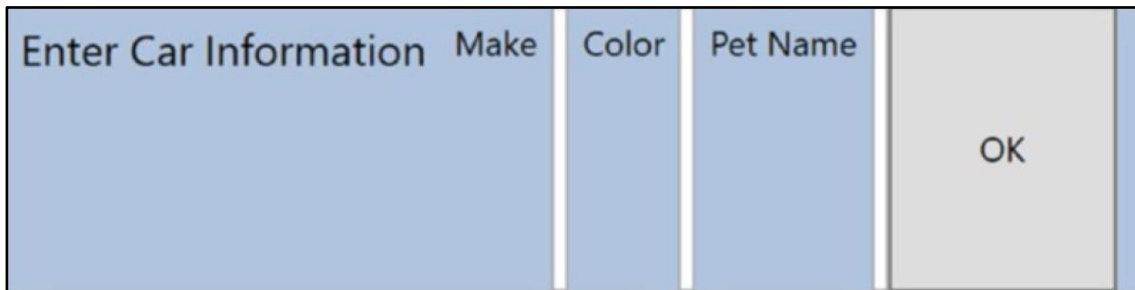
</Window>
```



StackPanel panelen

- Om **Orientation** propertyn istället sätts till **Horizontal**, kommer kontrollerna att organiseras från vänster till höger.

```
<StackPanel Background="LightSteelBlue" Orientation="Horizontal">
```



- Som med en **WrapPanel**, används sällan en **StackPanel** för att organisera kontroller direkt i et fönster, utan används som ett subelement i andra paneltyper.

Grid panelen

- **Grid** panelen är den mest flexibla paneltypen. Som en HTML tabell (eller ett *Excel* ark), kan en **Grid** fördelas i ett antal **celler**, där varje cell kan tilldelas kontroller.
- Man definierar en **Grid** i tre setg.
 1. Definiera och konfigurera varje **kolumn**.
 2. Definiera och konfigurera varje **rad**.
 3. **Tilldela kontroller till varje cell** genom att använda **attached property syntax**.
- Om man inte definierar några **kolumner** eller **rader**, kommer **Grid** panelen endast att innehålla **en enda cell** som fyller hela fönstrets area. Dessutom, om man inte tilldelar en kontroll till en cell (kolumn och rad), placeras den i **kolumn 0, rad 0**.
- De första två stegen (definiera kolumner och rader) görs genom att använda **Grid.ColumnDefinitions** och **Grid.RowDefinitions** elementen, vilka innehåller en samling **ColumnDefinition** respektive **RowDefinition** element.
- Varje cell i en **Grid** är ett .NET objekt, varför man kan konfigurera utseende och beteende av varje cell individuellt.

Grid panelen

```
<Window
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="100" Width="650">
```

```
    <Grid ShowGridLines="True" Background="LightSteelBlue">
```

```
        <!-- Define the rows/columns -->
```

```
        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition/>
```

```
            <ColumnDefinition/>
```

} 2 kolumner

```
        </Grid.ColumnDefinitions>
```

```
        <Grid.RowDefinitions>
```

```
            <RowDefinition/>
```

```
            <RowDefinition/>
```

} 2 rader

```
        </Grid.RowDefinitions>
```

```
        <!-- Now add the elements to the grid's cells -->
```

```
        <Label x:Name="lblInstruction" Grid.Column="0" Grid.Row="0" FontSize="15" Content="Enter Car Information"/>
```

```
        <Button x:Name="btnOK" Height="30" Grid.Column="0" Grid.Row="0" Content="OK"/>
```

```
        <Label x:Name="lblMake" Grid.Column="1" Grid.Row="0" Content="Make"/>
```

```
        <TextBox x:Name="txtMake" Grid.Column="1" Grid.Row="0" Width="193" Height="25"/>
```

```
        <Label x:Name="lblColor" Grid.Column="0" Grid.Row="1" Content="Color"/>
```

```
        <TextBox x:Name="txtColor" Width="193" Height="25" Grid.Column="0" Grid.Row="1" />
```

```
        <!-- Just to keep things interesting, add some color to the pet name cell -->
```

```
        <Rectangle Fill="LightGreen" Grid.Column="1" Grid.Row="1" />
```

```
        <Label x:Name="lblPetName" Grid.Column="1" Grid.Row="1" Content="Pet Name"/>
```

```
        <TextBox x:Name="txtPetName" Grid.Column="1" Grid.Row="1" Width="193" Height="25"/>
```

```
    </Grid>
```

```
</Window>
```

Varje element "kopplar sig" till en cell i **Grid** panelen genom att använda attached properties **Grid.Row** och **Grid.Column**. Den översta vänstra cellen specificeras med **Grid.Column="0"** och **Grid.Row="0"**, och eftersom denna **Grid** panelen har 2x2 celler, specificeras den nedersta högra cellen med **Grid.Column="1"** och **Grid.Row="1"**.

Grid panelen

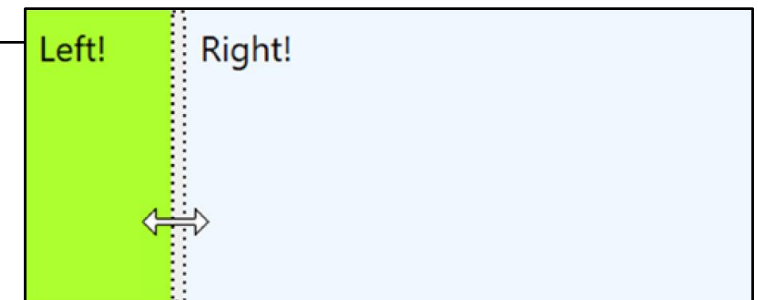
- Storleken för kolumner och rader i en **Grid** kan bestämmas på tre olika sätt.
 - Absolut storlek (t.ex. 100).
 - Automatisk storlek.
 - Relativ storlek (t.ex. 3x).
- En **absolut storlek** innebär att kolumnen eller raden får den angivna storleken i antal (*device-independent*) enheter.
- En **automatisk storlek** innebär att varje kolumn eller rad får en storlek beroende på kontrollernas storlek i varje kolumn eller rad.
- En **relativ storlek** innebär att den totala tillgängliga arean fördelas procentuellt på kolumnerna eller raderna (på samma sätt som en procentuell storlek i CSS). I exemplet nedan får den första kolumnen **25 procent** (eller **1/4**) av den totala arean medan den andra kolumnen får **75 procent** (eller **3/4**) av den totala arean.

```
<Grid.ColumnDefinitions>  
    <ColumnDefinition Width="1*" />  
    <ColumnDefinition Width="3*" />  
</Grid.ColumnDefinitions>
```


Grid panelen med GridSplitter

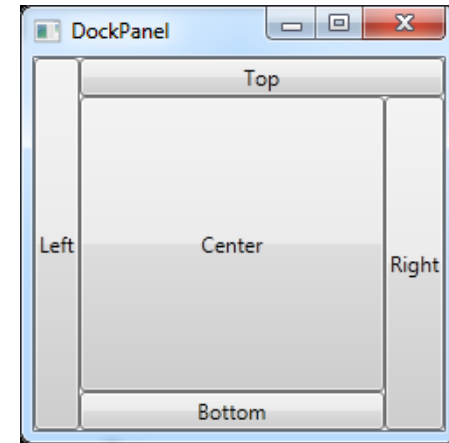
- En **Grid** har support för **GridSplitters**, som tillåter användaren att ändra storleken på kolumner och rader, varpå varje cell ändrar sin storlek beroende på hur dess innehåll har definierats.
- En **GridSplitter** läggs till en **Grid** genom att definiera en **GridSplitter** kontroll, och koppla den till en kolumn eller rad med **attached property** syntax. En **Width** eller **Height** (beroende på om vertikal eller horisontell *splitting* används) måste anges för **splittern** för att den skall synas på skärmen.
- I nedanstående exempel har en **GridSplitter** kopplats till en **Grid**s första kolumn (**Grid.Column="0"**). **GridSplittern** använder **attached property** syntax för att ange vilken kolumn den skall kopplas till. Kolumnen som splittern kopplas till har sin **Width** property satt till **Auto**. **GridSplitterns Width** har satts till **5 pixar**.

```
<Grid Background="LightSteelBlue">
  <!-- Define columns -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <!-- Add this label to cell 0 -->
  <Label x:Name="lblLeft" Background="GreenYellow" Grid.Column="0" Content="Left!"/>
  <!-- Define the splitter -->
  <GridSplitter Grid.Column="0" Width="5"/>
  <!-- Add this label to cell 1 -->
  <Label x:Name="lblRight" Grid.Column="1" Content="Right!"/>
</Grid>
```

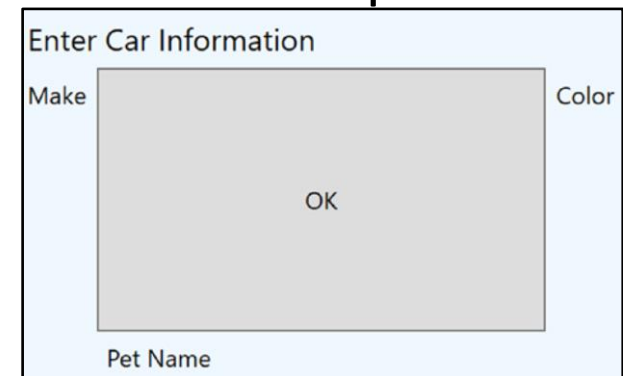


DockPanel panelen

- En **DockPanel** används typiskt som en “container” för andra paneltyper, och består av fem ytor; **Top**, **Left**, **Right**, **Bottom** och **Center**. En kontroll eller panel använder **attached property** syntax för att ange vilken yta som den skall koppla sig till via **DockPanel**ens **Dock** property (**Center** kan ej anges). <https://www.wpf-tutorial.com/panels/dockpanel>
- Om man lägger till flera element till samma yta i en **DockPanel**, kommer elementen att staplas utefter ytan i samma ordning som de läggs till.
- Fördelen med en **DockPanel** är att varje element kommer att stanna kvar i vald yta när användaren ändrar fönstrets storlek.
- Om man sätter en **DockPanel**s **LastChildFill** property till **true**, kommer sista elementet som läggs till **DockPanel**en att fylla återstående yta (som t.ex. med **Button** i exemplet nedan).



```
<DockPanel LastChildFill="True" Background="AliceBlue">
  <!-- Dock items to the panel -->
  <Label DockPanel.Dock="Top" Name="lblInstruction" FontSize="15"
    Content="Enter Car Information"/>
  <Label DockPanel.Dock="Left" Name="lblMake" Content="Make"/>
  <Label DockPanel.Dock="Right" Name="lblColor" Content="Color"/>
  <Label DockPanel.Dock="Bottom" Name="lblPetName" Content="Pet Name"/>
  <Button Name="btnOK" Content="OK"/>
</DockPanel>
```



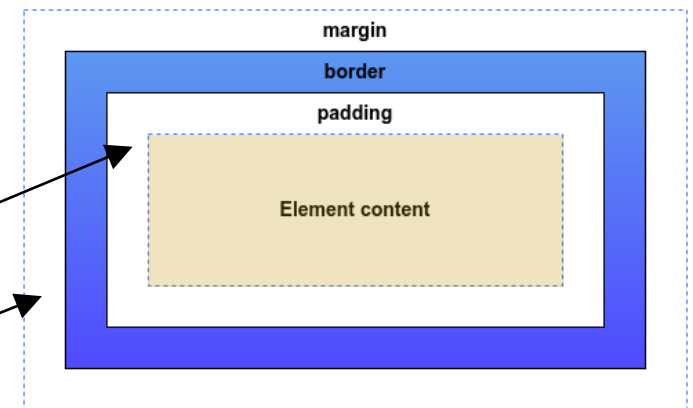
ScrollView

- **ScrollView** klassen kan användas för att få automatisk *skroll-beteende* för element i paneltyper. I exemplet visas en **scroll bar** till höger om **StackPanel** eftersom dess kontroller inte får plats i fönstret (vertikalt), och den finns innesluten i en **ScrollView**.

```
<ScrollView>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollView>
```



- Alla paneltyper innehåller många medlemmar som ger bättre kontroll över hur dess element skall placeras.
- Många WPF kontroller innehåller två properties, **Padding** och **Margin** som informerar den inneslutande panelen om hur mycket extra utrymme den vill ha runt omkring sig:
 - **Padding** property bestämmer hur mycket extra utrymme som skall användas **runt kontrollens inre** (mellan **border** och **content**).
 - **Margin** propertyn bestämmer hur mycket extra utrymme som skall användas **runt kontrollens yttre** (runt dess **border**).



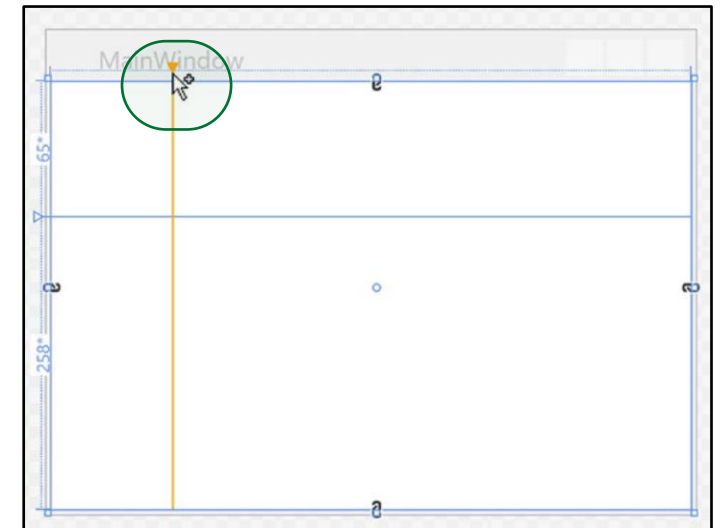
Att konfigurera paneler med Visual Studios designers

- Visual Studio har bra support för att skapa layouts genom att använda **Document Outline** fönstret.
- När man skapar ett WPF projekt i Visual Studio fås nedanstående XAML som default för applikationens huvudfönster, som innehåller en **Grid** panel.
- För att ändra storleken på **Grid** panelens celler eller skapa nya rader och kolumner, kan man först markera **Grid** elementet i **Document Outline** fönstret och sedan klicka på **Grid** panelens **ram** för att skapa nya rader och kolumner.

```
<Window x:Class="VisualLayoutTester.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"
  xmlns:local="clr-namespace:VisualLayoutTesterApp"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
```

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

```
</Window>
```

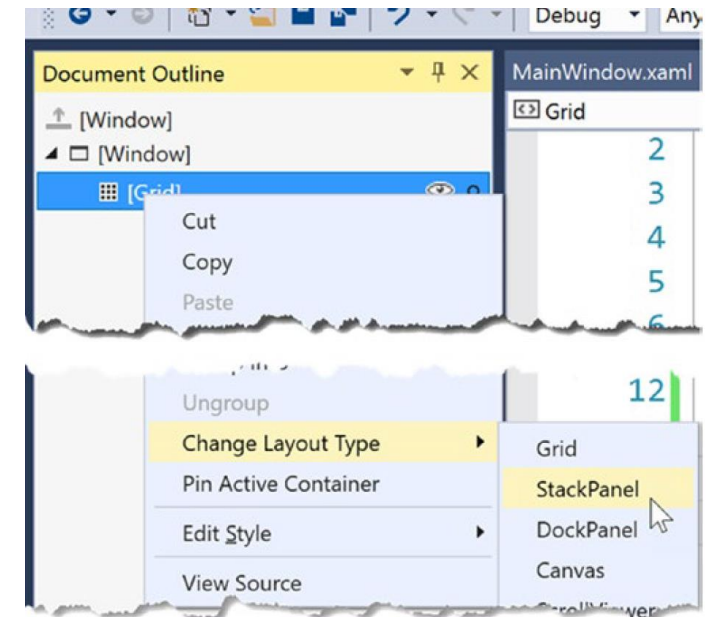


Att konfigurera paneler med Visual Studios designers

- Man kan *drag-and-drop:a* kontroller till en given cell på designytan, så kommer IDEt automatiskt att sätta rätt **Grid.Row** och **Grid.Column** properties för kontrollen, t.ex.:

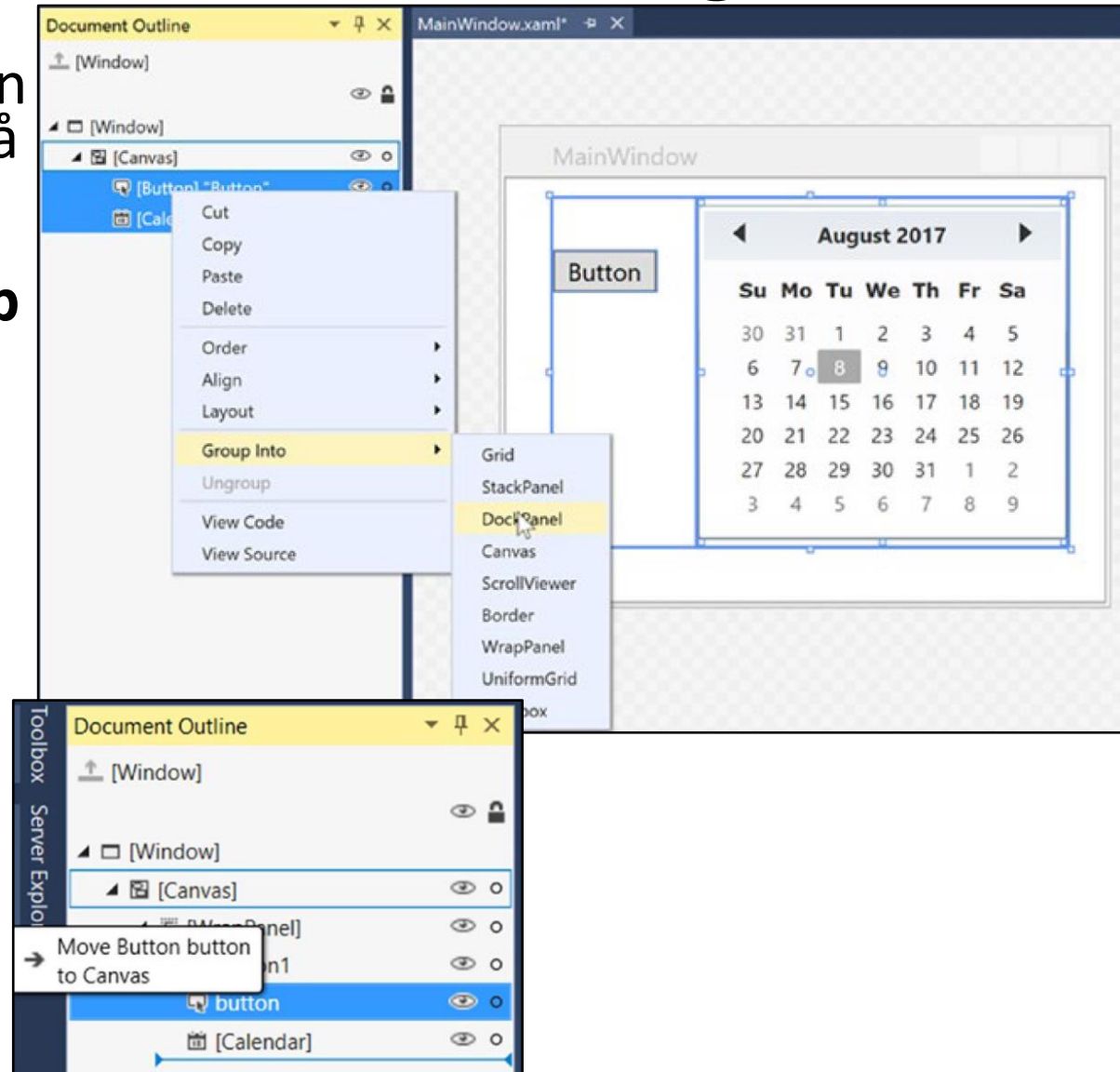
```
<Button x:Name="button" Content="Button" Grid.Column="1" HorizontalAlignment="Left"
        Margin="21,21.4,0,0" Grid.Row="1" VerticalAlignment="Top" Width="75"/>
```

- Man kan högerklicka på valfri *layout manager* i **Document Outline** fönstret, och *byta till en annan layout manager* via kontextfönstret som dyker upp.



Att konfigurera paneler med Visual Studios designers

- Man kan selektera kontroller på designytan (genom att hålla nere **<CTRL>** och klicka på kontrollerna) och gruppera dem i en ny nästlad **layout manager** (genom att högerklicka på selektionen och välja **Group Into** från *kontext* fönstret som dyker upp).
- **Document Outline** fönstret uppdateras med ändringarna. Alla noder i **Document Outline** fönstret kan *drag-and-drop:as* för att flytta omkring kontroller inom och mellan **layout managers**.



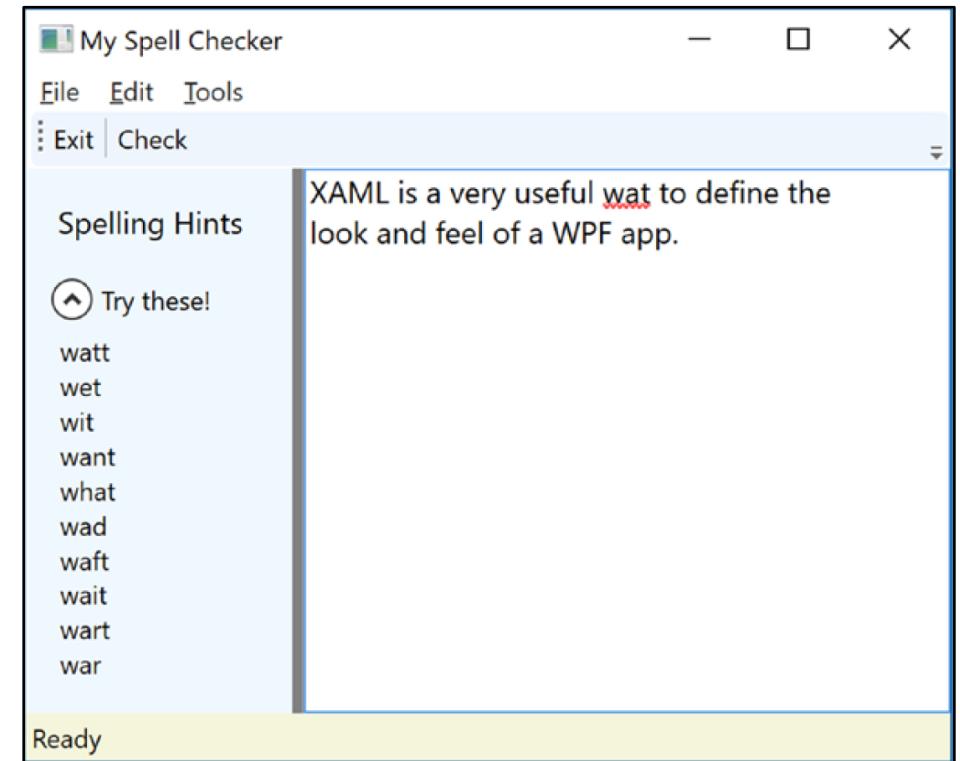
WPF applikation (exempel 1) - Ordbehandlare

- Vi vill bygga en ordbehandlare med stavningskontroll (WPF applikation).
- Målet är att konstruera en layout där huvudfönstret har ett huvudmeny (**meny**), en **toolbar** under huvudmenyn, och en **status bar** längst ner i huvudfönstret.
- **Status bar** kontrollen skall innehålla en **panel** som visar en beskrivande text när användaren väljer ett **menyalternativ** eller en **toolbar** knapp.
- **Menysystemet** och **toolbar** kontrollen skall ha **händelsehanterare** för att **avsluta applikationen** och för att visa **stavningsförslag** i en **Expander** kontroll.
- Vi börjar med att skapa ett nytt Visual Studio projekt av typ **WPF App (.NET Framework)**.

Utnyttja Microsofts documentation för att se vilka attribut och metoder som finns i olika kontroller, samt exempel på hur de används:

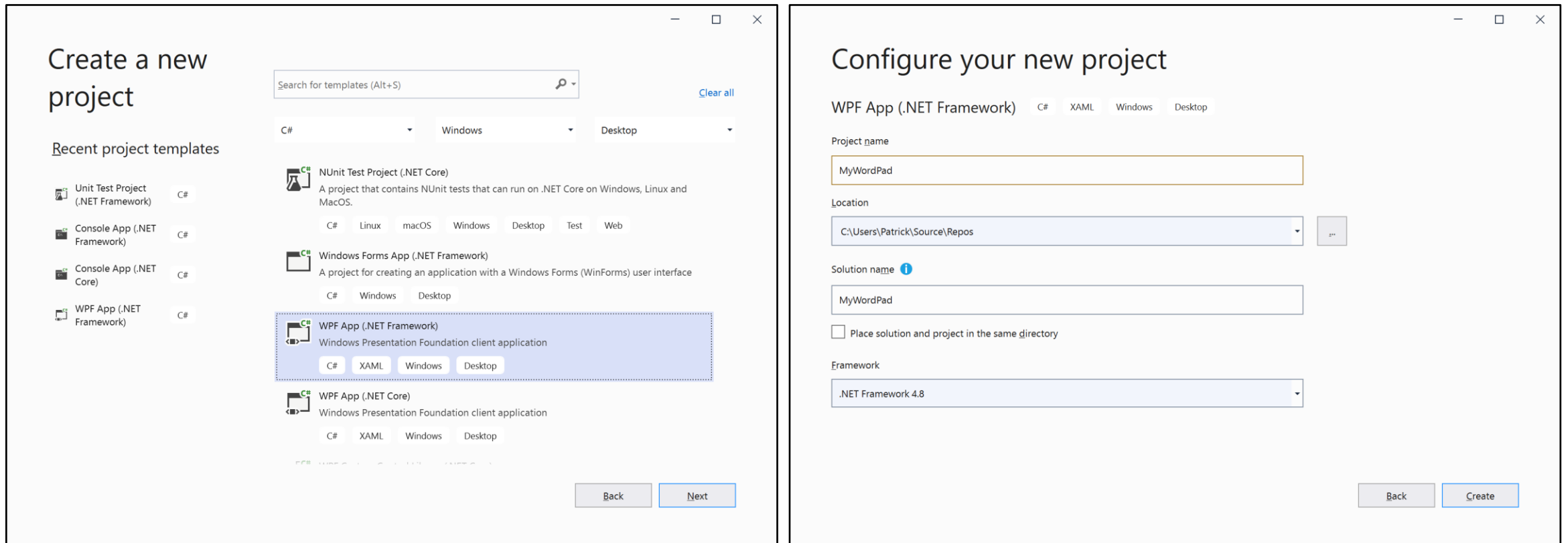
<https://docs.microsoft.com/en-us/dotnet/framework/wpf/index>

<https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/controls-by-category>



WPF applikation (exempel 1)

- Välj att skapa ett nytt projekt av typ **WPF App (.Net Framework)**.
- Döp sedan projektet till något lämpligt, t.ex. **MyWordPad**.



WPF applikation (exempel 1)

- I huvudfönstrets XAML-fil byter vi ut **Grid** panelen mot en **DockPanel**.

```
<Window x:Class="MyWordPad.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"
  xmlns:local="clr-namespace:MyWordPad"
  mc:Ignorable="d"
  Title="My Spell Checker" Height="350" Width="525">

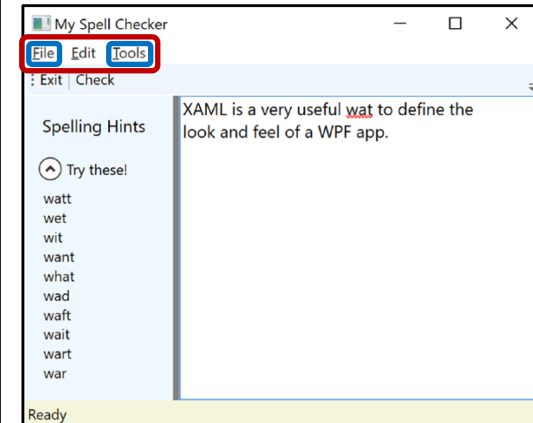
  <!-- This panel establishes the content for the window -->
  <DockPanel>
  </DockPanel>

</Window>
```

WPF applikation (exempel 1) - Menysystemet

- Menysystem i WPF representeras av **Menu** klassen, som innehåller en samling **MenuItem** objekt.
- En **MenuItem** kan hantera olika typer av **events**, där **Click** händelsen är mest användbar, och inträffar när användaren väljer ett menyalternativ.
- Vi börjar med att lägga till **File**, **Tools** och **Exit** menyvalen (**MenuItem**) i huvudmenyn (**Menu**), och skapar händelsehanterare för deras **Click**, **MouseEnter** och **MouseLeave** events.
- Menysystemet placeras i **DockPanel** panelens **Top** yta med **DockPanel.Dock="Top"**, och en **Separator** används för att infoga en tunn horisontell linje precis innan **Exit** menyvalet.
- Värdet för respektive menyvals **Header** property innehåller ett understrykningstecken (t.ex. **_Exit**) som bestämmer vilken bokstav som kommer att understrykas när användaren trycker ner **<Alt>** tangenten på tangentbordet. Detta är en **keyboard shortcut**, dvs användaren kan t.ex. välja menyvalet **Exit** med **<Alt> + <E>** eftersom understrykningstecknet anges innan bokstaven **E**.

```
<!-- Dock menu system on the top -->
<Menu DockPanel.Dock="Top" HorizontalAlignment="Left" Background="White" BorderBrush="Black">
  <MenuItem Header="_File">
    <Separator/>
    <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
      MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
  </MenuItem>
  <MenuItem Header="_Tools">
    <MenuItem Header="_Spelling Hints" MouseEnter="MouseEnterToolsHintsArea"
      MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```



WPF applikation (exempel 1) - Menysystemet

- Händelsehanterarna har skapats i *huvudfönstrets* tillhörande *code-behind* fil.
- Här har händelsehanteraren ***FileExit_Click()*** för menyvalet **File → Exit** implementeras, som ***stänger huvudfönstret*** (vilket kommer att avsluta applikationen eftersom huvudfönstret är applikationens översta fönster).
- Övriga händelsehanterare har inte implementerats ännu, men händelsehanterarna för **MouseEnter** och **MouseExit** kommer att uppdatera texten i **Statusbar** kontrollen, samt händelsehanteraren **ToolsSpellingHints_Click()** kommer att generera stavningsförslag.

```
using Microsoft.Win32;
using System.IO;
public partial class MainWindow : Window
{
    public MainWindow() {
        InitializeComponent();
    }
    protected void FileExit_Click(object sender, RoutedEventArgs args) {
        // Close this window.
        this.Close();
    }
    protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args) {}
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args) {}
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args) {}
    protected void MouseLeaveArea(object sender, RoutedEventArgs args) {}
}
```

Man kan också skriva:
`Environment.Exit(0);`

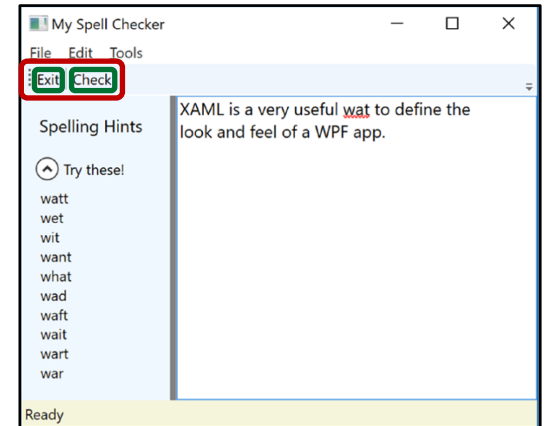
WPF applikation (exempel 1) - Menysystemet

- I Visual Studio kan man även bygga t.ex. menysystem, *toolbars* och *status bars* visuellt i designern.
- Exempelvis, om man högerklickar på **Menu** kontrollen visas ett **Add MenuItem** val som kan användas för att lägga till fler menyval till menyn.
- När man har lagt till de översta menyvalen i menyn, kan man lägga till submenyval och separatorer på samma sätt, expandera eller krympa menyn, osv. via valen som dyker upp när man högerklickar på meny-kontrollerna.

WPF applikation (exempel 1) – Toolbar kontrollen

- *Toolbars* (som representeras av **ToolBar** klassen) möjliggör typiskt ett alternativt sätt att aktivera ett menyalternativ på.
- I detta fallet innehåller **ToolBar** kontrollen två **Button** kontroller, vars events hanteras av samma händelsehanterare som för motsvarande menyval.
- Eftersom en **ToolBar** “är-en” **ContentControl** (via arv), kan man använda andra kontroller än knappar (**Button**), t.ex. *drop-down* listor, bilder och grafik.
- I XAML koden har **Check** knappen tilldelats en annan *cursor* typ för musen via **Cursor** propertyn.
- **ToolBar** kontrollen kan alternativt placeras i en **ToolBarTray** kontroll, som hanterar *layout*, *docking* och *drag-and-drop* operationer för en samling **ToolBar** objekt (*runtime*).

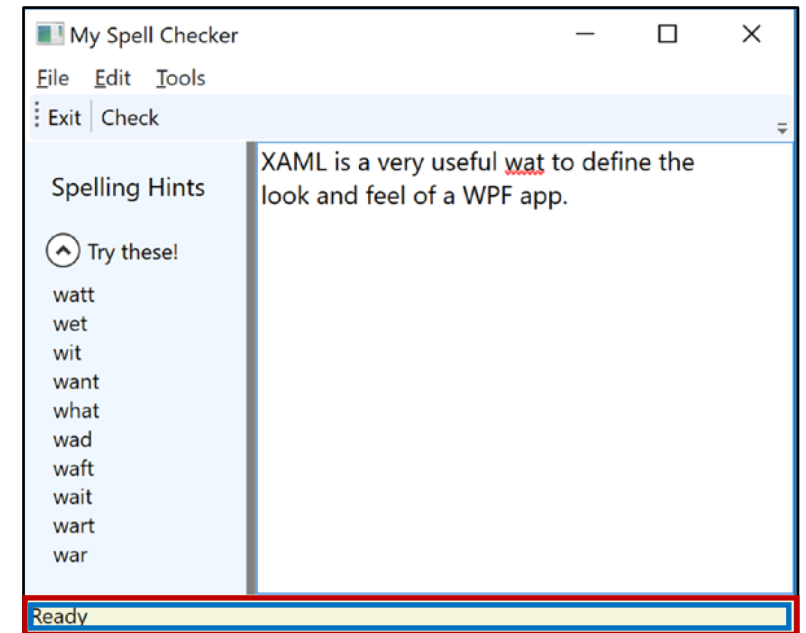
```
<!-- Put Toolbar under the Menu -->
<ToolBar DockPanel.Dock="Top">
    <Button Content="Exit" MouseEnter="MouseEnterExitArea"
            MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
    <Separator/>
    <Button Content="Check" MouseEnter="MouseEnterToolsHintsArea"
            MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click" Cursor="Help"/>
</ToolBar>
```



WPF applikation (exempel 1) – StatusBar kontrollen

- En **StatusBar** kontroll placeras i **DockPanelens** nedre yta (via **DockPanel.Dock="Bottom"**) och innehåller en **TextBlock** kontroll.
- En **TextBlock** kontroll kan innehålla text med support för flera textuella egenskaper såsom fel stil, understryken text, osv.

```
<!-- Put a StatusBar at the bottom -->
<StatusBar DockPanel.Dock="Bottom" Background="Beige">
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready" />
  </StatusBarItem>
</StatusBar>
```

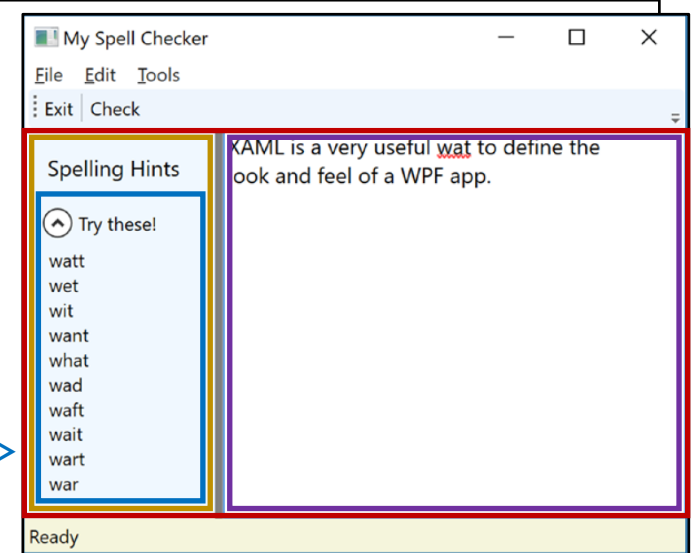


WPF applikation (exempel 1) – Övrig GUI design

- Avslutningsvis läggs en **Grid** med **två kolumner** och en **GridSplitter** till GUI:t.
- I vänstra kolumnen placeras en **Expander** kontroll (som kommer att visa stavningsförslag) innesluten i en **StackPanel** kontroll.
- I högra kolumnen placeras en **TextBox** kontroll som har support för flera rader, *scrollbars* och stavningskontroll. Hela **Grid** kontrollen placeras i **DockPanel** panelens **Left** yta.

```
<Grid DockPanel.Dock="Left" Background="AliceBlue">
  <!-- Define the rows and columns -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column="0" Width="5" Background="Gray"/>
  <StackPanel Grid.Column="0" VerticalAlignment="Stretch">
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling" Header="Try these!" Margin="10,10,10,10">
      <!-- This will be filled programmatically -->
      <Label Name="lblSpellingHints" FontSize="12"/>
    </Expander>
  </StackPanel>
  <!-- This will be the area to type within -->
  <TextBox Grid.Column="1" SpellCheck.IsEnabled="True" AcceptsReturn="True" Name="txtData" FontSize="14"
    BorderBrush="Blue" VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"/>
</Grid>
```

<https://www.wpf-tutorial.com/basic-controls/the-textbox-control>

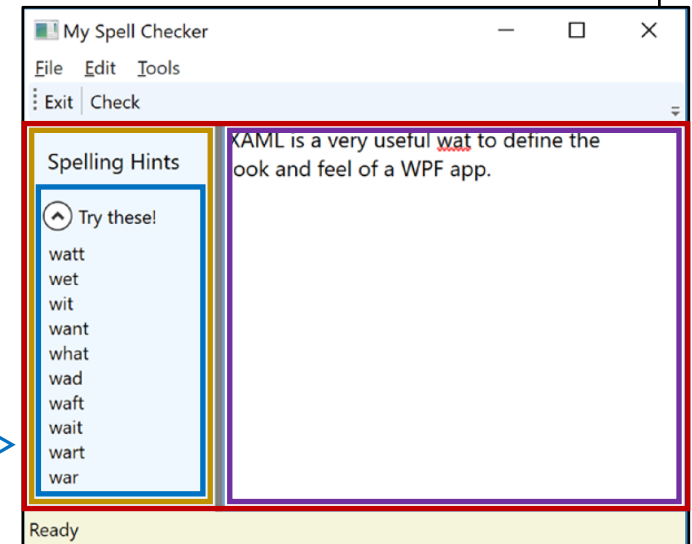


WPF applikation (exempel 1) – Övrig GUI design

- *SpellCheck.IsEnabled="True" aktiverar WPFs inbyggda support för stavningskontroll.*
- *Default stöds Engelska, Franska, Tyska samt Spanska, t.ex. Language="en-US".*
<https://www.wpf-tutorial.com/basic-controls/the-textbox-control>
- *För andra språk måste först aktuell Language pack installeras och konfigureras:*
<https://support.microsoft.com/en-us/help/14236/windows-language-packs>
<https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.inputlanguagemanager?view=netframework-4.8>

```
<Grid DockPanel.Dock="Left" Background="AliceBlue">
  <!-- Define the rows and columns -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column="0" Width="5" Background="Gray"/>
  <StackPanel Grid.Column="0" VerticalAlignment="Stretch">
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling" Header="Try these!" Margin="10,10,10,10">
      <!-- This will be filled programmatically -->
      <Label Name="lblSpellingHints" FontSize="12"/>
    </Expander>
  </StackPanel>
  <!-- This will be the area to type within -->
  <TextBox Grid.Column="1" SpellCheck.IsEnabled="True" AcceptsReturn="True" Name="txtData" FontSize="14"
    BorderBrush="Blue" VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"/>
</Grid>
```

<https://www.wpf-tutorial.com/basic-controls/the-textbox-control>



WPF applikation (exempel 1) – Övriga händelsehanterare

- Nu kan vi implementera händelsehanterarna **MouseEnterExitArea**, **MouseEnterToolsHintsArea** och **MouseLeaveArea** och så att de uppdaterar **StatusBar** kontrollens **TextBlock** med en lämplig text när musen flyttas mellan olika menyalternativ och *toolbar* knappar.

```
public partial class MainWindow : System.Windows.Window
{
    ...
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Exit the Application";
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Show Spelling Suggestions";
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Ready";
    }
}
```

WPF applikation (exempel 1) – Stavningsförslagen

- WPF innehåller inbyggd support för stavningskontroll. Genom att sätta **TextBox** kontrollens **SpellCheck.IsEnabled** property till **true**, kommer felstavade ord att understrykas med en röd linje. Dessutom kan man erhålla en lista med stavningsförslag via API:t.
- Händelsehanteraren **ToolsSpellingHints_Click()** implementeras enligt nedan, där nuvarande position av markören i text boxen först erhålls via **CaretIndex** propertyn, från vilket ett **SpellingError** objekt extraheras. Om det finns ett stavfel (dvs **SpellingError** objektet inte är **null**), loopas listan av stavningsförslag (som finns i property **Suggestions**) igenom och skrivs till **Label** kontrollen i **Expander** kontrollen.

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
    string spellingHints = string.Empty;
    // Try to get a spelling error at the current caret location.
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
        // Build a string of spelling suggestions.
        foreach (string s in error.Suggestions)
        {
            spellingHints += $"{s}\n";
        }
        // Show suggestions and expand the expander.
        lblSpellingHints.Content = spellingHints;
        expanderSpelling.IsExpanded = true;
    }
}
```

WPF Commands

- WPF har support för ***kontroll-agnostiska events*** via ***WPFs command arkitektur***.
- Typiskt definieras en ***event*** inom en viss basklass och kan endast användas av basklassen eller dess subklasser. Därför är sådana events ***starkt kopplade till klassen i vilken de är definierade***.
- ***WPF commands*** är event-liknande entiteter som är ***oberoende av en specific kontroll*** och kan, i många fall, appliceras till många kontrolltyper.
- Exempelvis har WPF support för ***copy, paste och cut commands***, som kan appliceras till många olika kontroller såsom menyalternativ, toolbar knappar, egendefinierade knappar och *keyboard shortcuts* (t.ex. ***<Ctrl> + <C>*** och ***<Ctrl> + <V>***).

WPF Commands

- WPF distribueras med flera inbyggda kontroll **commands** (som t.ex. kan användas ihop med *keyboard shortcuts* eller andra input gester).
- En **WPF command** är ett objekt med en property (oftast kallad **Command**) som returnerar ett objekt som implementerar **ICommand** interfacet (med två metoder **CanExecute()** och **Execute()** samt en händelse **CanExecuteChanged**).
- WPF innehåller ca. 100 färdigbyggda **command** klasser.

```
public interface ICommand
{
    // Occurs when changes occur that affect whether
    // or not the command should execute.
    event EventHandler CanExecuteChanged;

    // Defines the method that determines whether the command
    // can execute in its current state.
    bool CanExecute(object parameter);

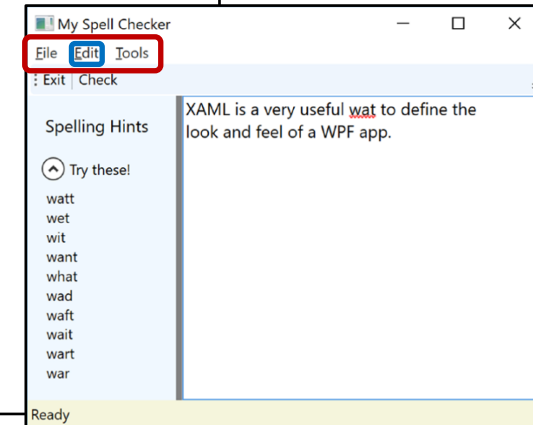
    // Defines the method to be called when the command is invoked.
    void Execute(object parameter);
}
```

WPF Class	Command Objects	Meaning in Life
ApplicationCommands	Close, Copy, Cut, Delete, Find, Open, Paste, Save, SaveAs, Redo, Undo	Various application-level commands
ComponentCommands	MoveDown, MoveFocusBack, MoveLeft, MoveRight, ScrollToEnd, ScrollToHome	Various commands common to UI components
MediaCommands	BoostBase, ChannelUp, ChannelDown, FastForward, NextTrack, Play, Rewind, Select, Stop	Various media-centric commands
NavigationCommands	BrowseBack, BrowseForward, Favorites, LastPage, NextPage, Zoom	Various commands relating to the WPF navigation model
EditingCommands	AlignCenter, CorrectSpellingError, DecreaseFontSize, EnterLineBreak, EnterParagraphBreak, MoveDownByLine, MoveRightByWord	Various commands relating to the WPF Documents API

Koppla *commands* till *Command* en property (exempel 1)

- Att koppla en **WPF command** till en kontroll som har support för **Command** propertyn (t.ex. en **Button** eller **MenuItem**) är ganska simpelt. I nedanstående exempel har ett nytt **Edit** menyalternativ (**MenuItem**) lagts till i huvudmenyn (**Menu**). **Edit** menyalternativet innehåller, i sin tur, tre **submenyalternativ** (**MenuItem**) för att kopiera, klistra in och klippa ut textuell data.
- För varje submenyalternativ i **Edit** menyn har ett **command** objekt tilldelats **Command** propertyn. Detta innebär att submenyalternativen automatiskt erhåller korrekt namn och *shortcut key* (t.ex. **<Ctrl> + <C>** för en "klipp ut" operation) samt att applikationen nu kan hantera *copy*, *cut* och *paste* utan att skriva någon extra kod!

```
<Menu DockPanel.Dock="Top" HorizontalAlignment="Left" Background="White" BorderBrush="Black">
  <MenuItem Header="_File">
    <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
      MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
  </MenuItem>
  <!-- New menu item with commands! -->
  <MenuItem Header="_Edit">
    <MenuItem Command="ApplicationCommands.Copy"/>
    <MenuItem Command="ApplicationCommands.Cut"/>
    <MenuItem Command="ApplicationCommands.Paste"/>
  </MenuItem>
  <MenuItem Header="_Tools">
    <MenuItem Header="_Spelling Hints" MouseEnter="MouseEnterToolsHintsArea"
      MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```



Koppla *commands* till godtyckliga händelser (exempel 1)

- Om man vill koppla ett **command** objekt till en godtycklig (applikationsspecifik) händelse, måste man skriva kod istället för XAML.
- Exempelvis, om man vill att huvudfönstret skall aktivera ett hjälpsystem när <F1> tangenten trycks på tangentbordet, kan man implementera koden i **SetF1CommandBinding()** metoden nedan.
- Denna metoden skapar ett **CommandBinding** objekt, som kan användas när man vill koppla (binda) ett **command** objekt till en händelsehanterare i applikationen. I detta fallet konfigureras **CommandBinding** objektet genom att använda **command ApplicationCommands.Help**, som automatiskt är <F1>-medveten.
- Slutligen läggs **CommandBinding** objektet till fönstrets **CommandBindings** samling.

```
public MainWindow()
{
    InitializeComponent();
    SetF1CommandBinding();
}

private void SetF1CommandBinding()
{
    CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
    helpBinding.CanExecute += CanHelpExecute;
    helpBinding.Executed += HelpExecuted;
    CommandBindings.Add(helpBinding);
}
```

Koppla *commands* till godtyckliga händelser (exempel 1)

- Ofta vill man att ett **CommandBinding** objekt skall hantera events **CanExecute** (som används för att bestämma om kommandot skall exekvera beroende på logiken i applikationen) och **Executed** (som innehåller koden som exekveras när kommandot körs).
- I detta fallet implementeras **CanHelpExecute()** så att <F1> hjälpen alltid tillåts att exekvera, genom att alltid sätta **CanExecute** till **true**. Dock, om <F1> hjälpen inte skall exekvera i vissa situationer i applikationen, kan **CanExecute** istället sättas till **false** när detta är nödvändigt.
- **HelpExecuted()** har i detta fallet implementerats genom att visa en **MessageBox** med ett meddelande till användaren.

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
    // Here, you can set CanExecute to false if you want to prevent the command from executing.
    e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Look, it is not that difficult. Just type something!", "Help!");
}
```

Open och Save Commands (exempel 1)

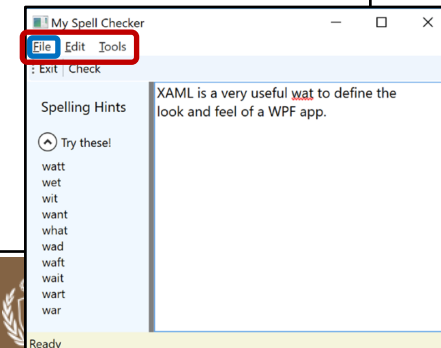
- I detta exempel kopplas två menyval till **ApplicationCommands** objekten **Open** och **Save** så att text i en **TextBox** kan läsas/skrivas från/till textfiler (*.txt). Återigen behövs två händelsehanterare för respektive **ApplicationCommands CanExecute** och **Executed** events. Här används *property-element* syntaxen **Window.CommandBindings** för att lägga till respektive **CommandBinding** till fönstrets **CommandBindings**.

```
<MenuItem Header="_File">
  <MenuItem Command="ApplicationCommands.Open"/>
  <MenuItem Command="ApplicationCommands.Save"/>
  <Separator/>
  <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea" MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
</MenuItem>
```

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation="CenterScreen">

  <!-- This will inform the Window which handlers to call, when testing for the Open and Save commands. -->
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open" Executed="OpenCmdExecuted" CanExecute="OpenCmdCanExecute"/>
    <CommandBinding Command="ApplicationCommands.Save" Executed="SaveCmdExecuted" CanExecute="SaveCmdCanExecute"/>
  </Window.CommandBindings>

  <!-- This panel establishes the content for the window -->
  <DockPanel>
    ...
  </DockPanel>
</Window>
```

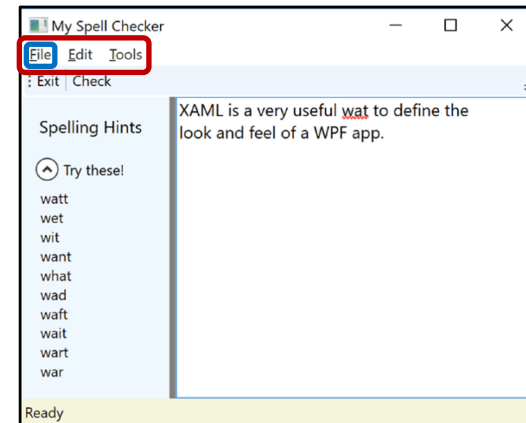


Open och Save Commands (exempel 1)

- Händelsehanterarna implementeras i motsvarande *code-behind* enligt nedan, som kommer att visa respektive *OpenFileDialog* och *SaveFileDialog*.

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e) {  
    e.CanExecute = true;  
}  
private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e) {  
    e.CanExecute = true;  
}
```

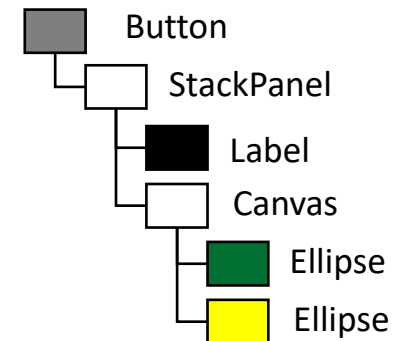
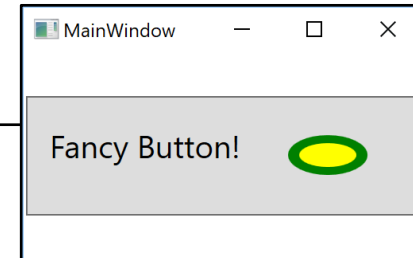
```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e) {  
    // Create an open file dialog box and only show XAML files.  
    var openDlg = new OpenFileDialog { Filter = "Text Files (*.txt)";  
    // Did they click on the OK button?  
    if (true == openDlg.ShowDialog())  
    {  
        // Load all text of selected file.  
        string dataFromFile = File.ReadAllText(openDlg.FileName);  
        // Show string in TextBox.  
        txtData.Text = dataFromFile;  
    }  
}  
private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e) {  
    var saveDlg = new SaveFileDialog { Filter = "Text Files (*.txt)";  
    // Did they click on the OK button?  
    if (true == saveDlg.ShowDialog())  
    {  
        // Save data in the TextBox to the named file.  
        File.WriteAllText(saveDlg.FileName, txtData.Text);  
    }  
}
```



Routed Events

- De flesta events i .NETs standardbibliotek skickar två parametrar till sina händelsehanterare, där första parametern är av typ **object** och andra parameter är av typ **EventArgs** eller en av dess subklasser.
- I WPF har denna **standard event modellen** förfinats till en **routed events modell**, där den andra parameter är av typ **RoutedEventArgs** eller en av dess subklasser.
- **Routed events modellen** ser till så att events kan processas på ett sätt som bättre passar XAMLs trädmodell.
- Anta att en knapp (**Button**) har definierats med komplex *content* samt en händelsehanterare för knappens **Click** event.

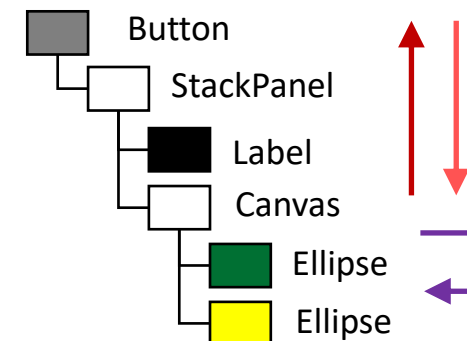
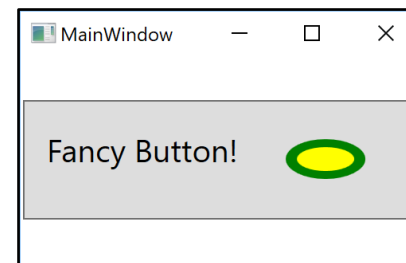
```
<Button Name="btnClickMe" Height="75" Width="250" Click="btnClickMe_Clicked">
  <StackPanel Orientation="Horizontal">
    <Label Height="50" FontSize="20">Fancy Button!</Label>
    <Canvas Height="50" Width="100">
      <Ellipse Name="outerEllipse" Fill="Green" Height="25" Width="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name="innerEllipse" Fill="Yellow" Height="15" Width="36" Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```



Routed Events

- Knappens **Click** event använder **RoutedEventHandler** (en *delegate*) som kräver en händelsehanterare där första parametern är av typ **object** och andra parametern är av typ **System.Windows.RoutedEventArgs**.

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
    // Do something when button is clicked.
    MessageBox.Show("Clicked the button");
}
```

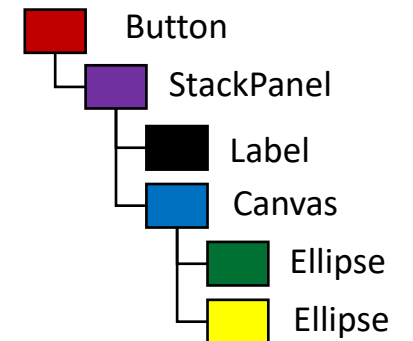
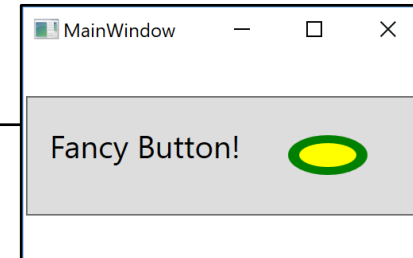


- Händelsehanteraren kommer att anropas oberoende av vilken del av knappens **Content** man klickar på (den **gröna elipsen**, den **gula elipsen**, **labeln** eller **knappens yta**), tack vare **routed events modellen** som automatiskt propagerar en **event** uppför (eller nerför) XAML-trädet av element tills en lämplig händelsehanterare hittas.
- Mer specifikt kan ett **routed event** använda sig av tre **routing strategier**.
 - Om ett **event** rör sig från elementet som skapade **eventet** och uppför trädet till andra inneslutande element sägs **eventet** vara ett **bubbling event**.
 - Om ett event rör sig från det yttersta **elementet** (t.ex. en **Window**) nerför trädet till ett element sägs **eventet** vara ett **tunneling event**.
 - Om ett **event** genereras och hanteras direkt av elementet som skapade **eventet** sägs **eventet** vara ett **direct event**.

Routed Bubbling Events

- I nuvarande exempel, om man klickar på:
 - Den gula **Ellipse**n, kommer **Click** eventet att “bubbla upp” till dess inneslutande scope (**Canvas**), och vidare till **StackPanel**, samt slutligen till **Button** där det finns en **händelsehanterare**.
 - På **Label** kontrollen, kommer **Click** eventet att “bubbla upp” till **StackPanel**, och sedan **Button**.
- På grund av **Bubbling Events mönstret** behöver man alltså inte skapa händelsehanterare för alla element i en *komplex* kontroll.

```
<Button Name="btnClickMe" Height="75" Width="250" Click="btnClickMe_Clicked">  
  <StackPanel Orientation="Horizontal">  
    <Label Height="50" FontSize="20">Fancy Button!</Label>  
    <Canvas Height="50" Width="100">  
      <Ellipse Name="outerEllipse" Fill="Green" Height="25" Width="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>  
      <Ellipse Name="innerEllipse" Fill="Yellow" Height="15" Width="36" Canvas.Top="17" Canvas.Left="32"/>  
    </Canvas>  
  </StackPanel>  
</Button>
```

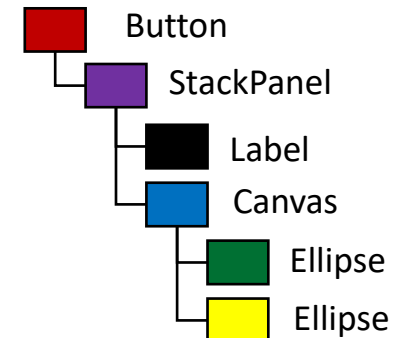
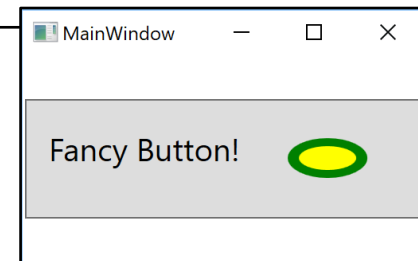


Routed Bubbling Events

- Dock kan man skapa egna händelsehanterare för olika element i en *komplex* kontroll.
- I detta fallet har en egen händelsehanterare skapats för event **MouseDown** på den gröna **Ellipse**.
- Nu kommer den gröna **Ellipse**s **MouseDown** **händelsehanterare** att anropas om musknappen trycks ner på den gröna **Ellipse** (och vidare till **Buttons Click** händelsehanterare), samt **Buttons** händelsehanterare om man klickar på övriga delar av **Buttons** scope (**Content**).
- **Routed Bubbling events** rör sig alltid från elementet som skapade eventet till nästa inneslutande *scope*. Därför, om man klickar på den gula **Ellipse**, kommer eventet att “bubbla upp” till **Canvas** (inte till den gröna **Ellipse**).

```
<Button Name="btnClickMe" Height="75" Width = "250" Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100">
      <Ellipse Name="outerEllipse" Fill="Green" Height="25" MouseDown="outerEllipse_MouseDown"
        Width="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name="innerEllipse" Fill ="Yellow" Height="15" Width="36" Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Change title of window.
    this.Title = "You clicked the outer ellipse!";
}
```

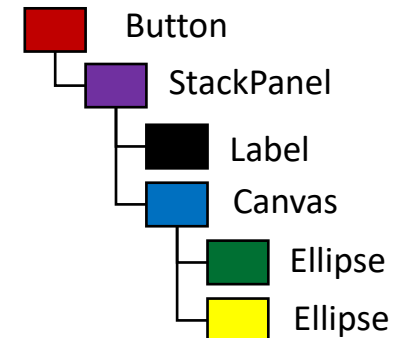
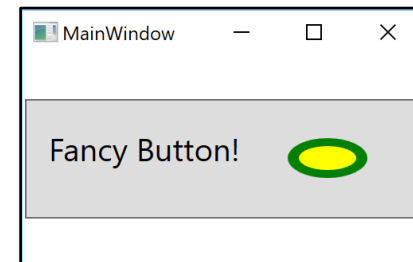


Att avbryta *Routed Bubbling Events*

- Om man klickar på den gröna *Ellipse* kommer händelsehanteraren för ***MouseDown*** att triggas, men dessutom kommer eventet att “bubbla vidare” till ***Click*** händelsehanteraren för *Button*.
- Om man vill informera WPF att ett event inte skall “bubbla vidare” uppför element-trädet, kan man sätta propriety *Handled* (i parametern av typ ***EventArgs*** eller en av dess subtyper, t.ex. ***MouseButtonEventArgs***) till ***true***.
- ***Routed Bubbling events*** gör det alltså möjligt att betrakta en *komplex* grupp av kontroller (***Content***) som ett enda logiskt element (t.ex. en *Button*) eller som diskreta element (t.ex. en *Ellipse* i en *Button*).

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Change title of window.
    this.Title = "You clicked the outer ellipse!";

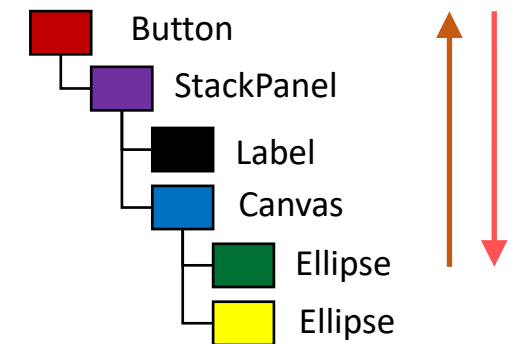
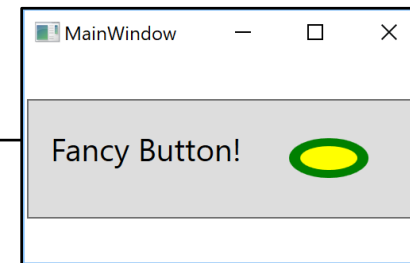
    // Stop bubbling!
    e.Handled = true;
}
```



Routed Tunneling Events

- **Tunneling events** börjar alltid från det yttersta (översta) elementet i element-trädet och rör sig nedför trädet till ett **specifikt element**.
- Varje **Bubbling event** har ett tillhörande **Tunneling event** (de kommer alltid i par), där ett **Tunneling event** alltid triggas innan tillhörande **Bubbling event**.
- Ett **Bubbling events** motsvarande **Tunneling event** börjar alltid med ordet **Preview**, t.ex. **PreviewMouseDown** (tunneling) respektive **MouseDown** (bubbling).
- Ett **Tunneling event** hanteras på samma sätt som ett **Bubbling event**.

```
<Ellipse Name="outerEllipse" Fill="Green" Height="25"
  MouseDown="outerEllipse_MouseDown"
  PreviewMouseDown="outerEllipse_PreviewMouseDown"
  Width="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

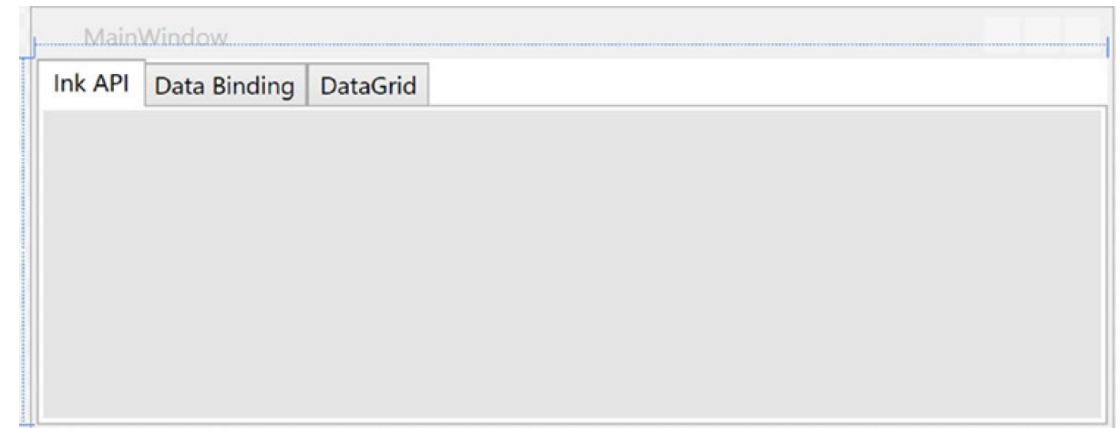


Routed Tunneling Events

- Varför har alla WPF events en tillhörande ***Tunneling event*** till varje ***Bubbling event***?
- Genom att “preview:a” events (***Tunneling event***), kan logik implementeras för t.ex. datavalidering eller för att avbryta ***bubbling*** innan tillhörande ***Bubbling event*** triggas.
- Exempelvis, om man har en ***TextBox*** som endast skall innehålla numerisk data, kan man hantera eventet ***PreviewKeyDown***, och om användaren har matat in icke-numerisk data, kan tillhörande ***Bubbling event*** avbrytas genom att sätta propertyn ***Handled*** till ***true*** i händelsehanteraren för ***PreviewKeyDown***.

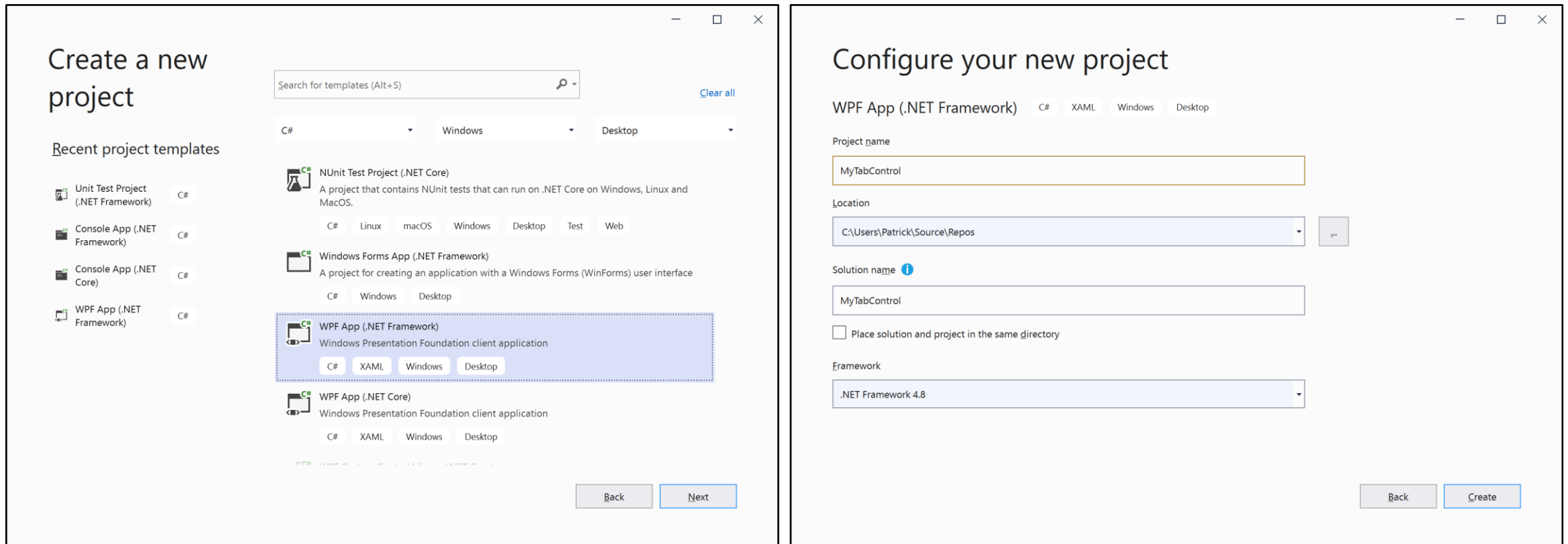
WPF applikation (exempel 2)

- Vi vill bygga en WPF applikation som innehåller en **TabControl** med tre flikar.
 - I första fliken demonstreras kontrollerna:
 - **RadioButton** (**CheckBox** fungerar på ett liknande sätt).
 - **ComboBox** (**ListBox** fungerar på ett liknande sätt).
 - **InkCanvas**.
 - I andra fliken demonstreras:
 - WPFs **databindningsmodell till kontroller**.
 - I tredje fliken demonstreras:
 - WPFs **databindningsmodell till objekt**.
 - **DataGrid** kontrollen.
- Vi börjar med att skapa ett nytt Visual Studio projekt av typ *WPF App (.Net Framework)*.



WPF applikation (exempel 2)

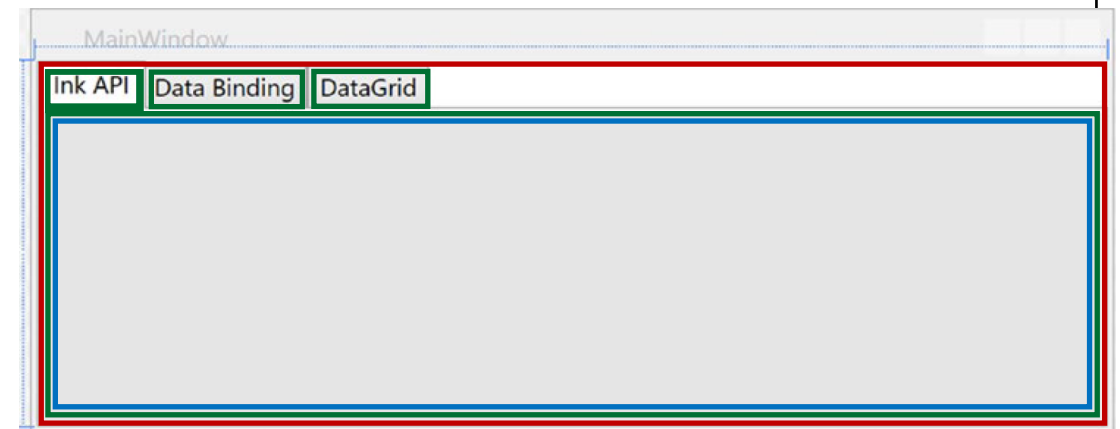
- Välj att skapa ett nytt projekt av typ **WPF App (.Net Framework)**.
- Döp sedan projektet till något lämpligt, t.ex. **MyTabControl**.



WPF applikation (exempel 2) - *TabControl*

- Vi börjar med att sätta fönstrets (**Window**) Width="800" och Height="350".
- Därefter "drag-and-drop:ar" vi en **TabControl** kontroll (från Visual Studios **toolbox** i vänstermarginalen) till designytan och uppdaterar XAMLn enligt nedan (lägg märke till att en "flik" skapas med en **TabItem** och att varje "flik" innehåller en **StackPanel**).
- För att lägga till fler flikar kan man högerklicka på **TabControl** noden i **Document Outline** (eller på **TabControl** kontrollen på designytan) och välja **Add TabItem** (alternativt lägga till fler **TabItems** direkt i XAML).
- När man selekterar en flik, blir fliken "aktiv" för editering och kan designas genom att drag-and-drop:a kontroller från **Toolboxen**.

```
<TabControl Name="MyTabControl" HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
    <TabItem Header="InkAPI">
        <StackPanel Background="#FFE5E5E5">
        </StackPanel>
    </TabItem>
    <TabItem Header="Data Binding">
        <StackPanel Width="250">
        </StackPanel>
    </TabItem>
    <TabItem Header="DataGrid">
        <StackPanel>
        </StackPanel>
    </TabItem>
</TabControl>
```



WPF applikation (exempel 2) – Första fliken

- Vi börjar med att designa första fliken (*Ink API*) som bl.a. innehåller:
 - En **ToolBar** med:
 - Tre **RadioButton**s (som kan *markeras/avmarkeras* ... oftast uteslutande).
 - En **ComboBox** med 3 element (där ett element kan *selekteras* från en drop-down lista).
 - Tre **Buttons**.
 - En **InkCanvas** (kan "*ritas på*" med musen, eller *stylus* penna på en tryckkänslig skärm).



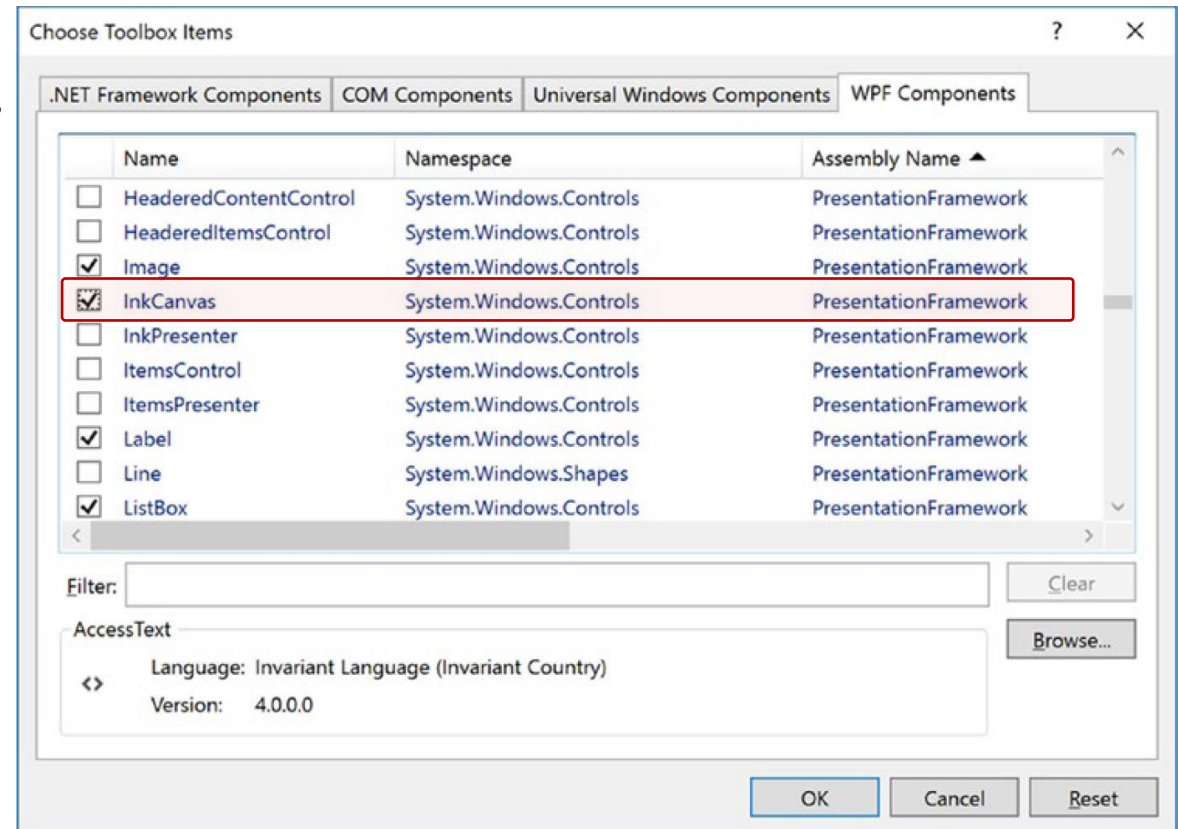
WPF applikation (exempel 2) – Första fliken

```
<TabItem Header="InkAPI">
  <StackPanel Background="#FFE5E5E5">
    <ToolBar Name="InkToolBar" Height="60">
      <Border Margin="0,2,0,2.4" Width="280" VerticalAlignment="Center">
        <WrapPanel>
          <RadioButton x:Name="inkRadio" Margin="5,10" GroupName="editingMode" Content="Ink Mode!" IsChecked="True"/>
          <RadioButton x:Name="eraseRadio" Margin="5,10" GroupName="editingMode" Content="Erase Mode!"/>
          <RadioButton x:Name="selectRadio" Margin="5,10" GroupName="editingMode" Content="Select Mode!"/>
        </WrapPanel>
      </Border>
      <Separator/>
      <ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0">
        <ComboBoxItem Content="Red"/>
        <ComboBoxItem Content="Green"/>
        <ComboBoxItem Content="Blue"/>
      </ComboBox>
      <Separator/>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <Button Grid.Column="0" x:Name="btnSave" Margin="10,10" Width="70" Content="Save Data"/>
        <Button Grid.Column="1" x:Name="btnLoad" Margin="10,10" Width="70" Content="Load Data"/>
        <Button Grid.Column="2" x:Name="btnClear" Margin="10,10" Width="70" Content="Clear"/>
      </Grid>
    </ToolBar>
    <InkCanvas x:Name="MyInkCanvas" Background="#FFB6F4F1"/>
  </StackPanel>
</TabItem>
```

Alla **RadioButtons** med samma **GroupName** är ömsesidigt uteslutande, dvs endast en **RadioButton** i gruppen kan vara vald åt gången.

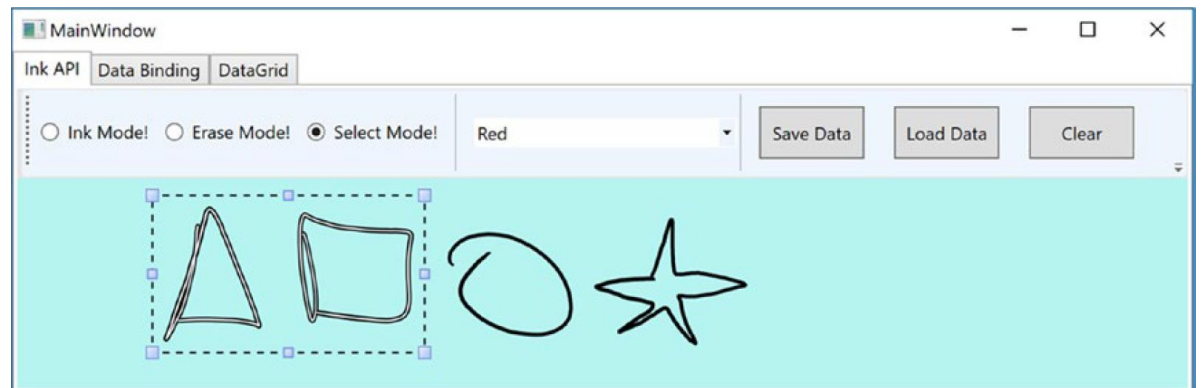
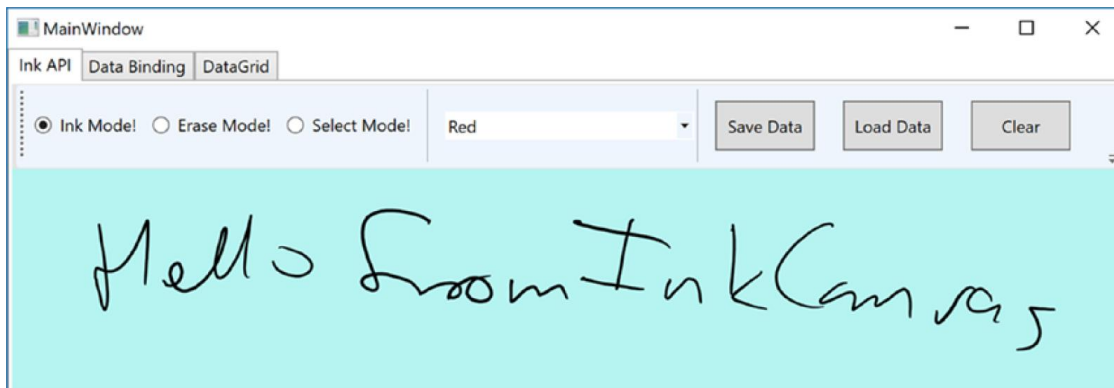
WPF applikation (exempel 2) - Toolboxen

- I detta fallet lades en **InkCanvas** kontroll till direkt i XAML.
- Om man vill lägga till en **InkCanvas** kontroll via **Toolboxen** (*drag-and-drop*), så måste man först lägga till kontrollen i **Toolboxen** eftersom kontrollen inte visas där som default (alla kontroller visas inte i **Toolboxen** som default).
- För att lägga till fler kontroller i **Toolboxen**, högerklicka i **Toolboxen** och välj **Choose Items**. Därefter visas ett fönster med tillgängliga kontroller som kan markeras för att lägga till dem i **Toolboxen**.
- I detta fallet markerar vi **InkCanvas** kontrollen och klickar på **OK** knappen.



WPF applikation (exempel 2) - *InkCanvas*

- Man kan “rita på” en ***InkCanvas*** kontroll med musen, en *stylus* penna eller fingrarna (på en tryckkänslig skärm).
- ***InkCanvas*** har support för ett antal ***editing modes*** som bestäms av property ***EditingMode***, och vars värde väljs från ***enumen InkCanvasEditingMode***, t.ex.:
 - ***Ink***, som tillåter användaren att rita på ***InkCanvas***.
 - ***Select***, som tillåter användaren att använda musen för att selektera en region, som sedan kan flyttas eller förstoras/förminskas.
 - ***EraseByStoke***, som tillåter användaren att ta bort det senaste ***mouse stroke***.
- En ***stroke*** är den rendering som sker vid en mus ner/mus upp operation (eller sylus/fingrar)
- ***InkCanvas*** lagrar alla ***strokes*** i en ***StrokeCollection***, som kan accessas via ***Strokes*** proprietytyn.



WPF applikation (exempel 2) - *RadioButton*

- ***RadioButton*** innehåller en ***IsChecked*** property, som “togglar” mellan ***true*** och ***false*** när användaren klickar på den.
- ***RadioButton*** har två events (***Checked*** och ***Unchecked***) som kan hanteras för att hålla reda på tillståndsförändringar.
- För att en grupp ***RadioButton*** kontroller skall vara ömsesidigt uteslutande, tilldelar man samma värde till propertyn ***GroupName*** (för varje ***RadioButton***).



- En ***CheckBox*** har samma property (utan ***GroupName***) och events som en ***RadioButton***, men visas som en ruta som kan avbockas/bockas (ej ömsesidigt uteslutande).

WPF applikation (exempel 2) - *ComboBox*

- En **ComboBox** visas som en *drop-down* lista och innehåller noll till flera element.
- Det finns tre sätt att ta reda på vilket element som är valt i en **ComboBox**:
 - Propertyn **SelectedIndex** returnerar **indexet** (noll-baserat) av valt element (värdet **-1** innebär att inget element är valt).
 - Propertyn **SelectedItem** returnerar **objektet** för valt element.
 - Propertyn **SelectedValue** returnerar **värdet** för valt element (som typiskt erhålls genom ett anrop till **ToString()**).



- En **ListBox** har samma propriety och events som en **ComboBox**, men visas som en expanderad lista (inte som en *drop-down* lista), och där flera element kan vara selekterade samtidigt.

WPF applikation (exempel 2) - Händelsehanterare

- För att skapa händelsehanterare för alla **RadioButtons** och **ComboBoxen** kan respektive kontroll väljas i **designern**, i **Document Outline** eller i **XAML** filen, varpå events fliken (med “blixtsymbolen”) i **Properties** fönstret kan användas för att namnge och skapa händelsehanterarna i **code-behind** filen
- I detta fallet anger vi samma namn **RadioButtonClicked** på händelsehanteraren för varje **RadioButtons Click** event. Därför kommer alla tre **RadioButtons Click** event att gå till samma händelsehanterare.
- Vi skapar också en händelsehanterare med namn **ColorChanged** för **ComboBoxens SelectionChanged** event (som triggas när valt element byts).

```
public partial class MainWindow : Window {  
    public MainWindow() {  
        this.InitializeComponent();  
    }  
  
    private void RadioButtonClicked(object sender, RoutedEventArgs e)  
    {  
        // TODO: Add event handler implementation here.  
    }  
  
    private void ColorChanged(object sender, SelectionChangedEventArgs e)  
    {  
        // TODO: Add event handler implementation here.  
    }  
}
```

WPF applikation (exempel 2) - Händelsehanterare

- Vi uppdaterar fönstrets konstruktor i **code-behind** filen genom att sätta **InkCanvas** i **Ink** mode, motsvarande **RadioButton** som **IsChecked = true**, samt selekterar det första elementet i **ComboBoxen**.
- Vi uppdaterar även händelsehanteraren för samtliga **RadioButtons Click** event så att rätt **InkCanvas mode** väljs beroende på vilken **RadioButton** är vald.

```
public MainWindow()
{
    this.InitializeComponent();
    // Be in Ink mode by default.
    this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    this.inkRadio.IsChecked = true;
    this.comboColors.SelectedIndex = 0;
}

private void RadioButtonClicked(object sender, RoutedEventArgs e)
{
    // Place the InkCanvas in a mode based on which button sent the event
    switch((sender as RadioButton)?.Content.ToString())
    {
        // These strings must be the same as the Content values for each
        // RadioButton.
        case "Ink Mode!":
            this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
            break;
        case "Erase Mode!":
            this.MyInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
            break;
        case "Select Mode!":
            this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Select;
            break;
    }
}
```

WPF applikation (exempel 2) - Händelsehanterare

- Vi uppdaterar även händelsehanteraren för **ComboBox**ens **SelectionChanged** så att vi kan byta färg på "pennan" vi använder för att rita på vår **InkCanvas**.
- **InkCanvas** propertyn **DefaultDrawingAttributes** innehåller ett **DrawingAttributes** objekt som kan användas för att konfigurera flera egenskaper för "pennan", inklusive storlek och färg.

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Get the selected value in the combo box.
    string colorToUse = (this.comboColors.SelectedItem as ComboBoxItem)?.Content.ToString();

    // Change the color used to render the strokes.
    this.MyInkCanvas.DefaultDrawingAttributes.Color = (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

- **ComboBox**en har en samling **ComboBoxItems**. När **SelectedItem** anropas (ovan), returneras valt **ComboBoxItem** (som lagras som en **Object**). Därefter typkonverteras denna till en **ComboBoxItem**, från vilket värdet av dess **Content** hämtas ut. Detta värdet är en sträng som innehåller önskad färg (*Red*, *Green*, eller *Blue*). Denna strängen konverteras sedan till ett **Color** objekt med hjälp av **ColorConverter** klassen.

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
    <ComboBoxItem Content="Red"/>
    <ComboBoxItem Content="Green"/>
    <ComboBoxItem Content="Blue"/>
</ComboBox>
```

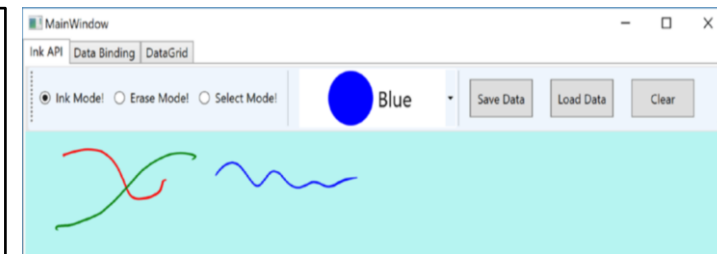
WPF applikation (exempel 2) - Händelsehanterare

- **ComboBox**ar (och **ListBox**ar) kan även innehålla **komplex Content** (istället för endast strängar).

```
<ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0" SelectionChanged="ColorChanged">
    <StackPanel Orientation="Horizontal" Tag="Red">
        <Ellipse Fill="Red" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center" VerticalAlignment="Center" Content="Red"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Tag="Green">
        <Ellipse Fill="Green" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center" VerticalAlignment="Center" Content="Green"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Tag="Blue">
        <Ellipse Fill="Blue" Height="50" Width="50"/>
        <Label FontSize="20" HorizontalAlignment="Center" VerticalAlignment="Center" Content="Blue"/>
    </StackPanel>
</ComboBox>
```

- Här tilldelas ett värde (en sträng) till varje **StackPanel**s **Tag** property, så att man kan identifiera vilken **StackPanel** har valts av användaren (detta är en *quick-and-dirty* lösning som används i kursboken, men bör inte användas i "riktig" kod).

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Get the Tag of the selected StackPanel.
    string colorToUse = (this.comboColors.SelectedItem as StackPanel).Tag.ToString();
    ...
}
```



WPF applikation (exempel 2) - Händelsehanterare

- Slutligen lägger vi till händelsehanterare för knapparnas (**Button**) **Click** event så att vi kan **rensa** vår **InkCanvas**, samt **spara/läsa** dess innehåll till/från en fil.

```
<Button Grid.Column="0" x:Name="btnSave" Margin="10,10" Width="70" Content="Save Data" Click="SaveData"/>
<Button Grid.Column="1" x:Name="btnLoad" Margin="10,10" Width="70" Content="Load Data" Click="LoadData"/>
<Button Grid.Column="2" x:Name="btnClear" Margin="10,10" Width="70" Content="Clear" Click="Clear"/>
```

```
private void SaveData(object sender, RoutedEventArgs e) {
    // Save all data on the InkCanvas to a local file.
    using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
    {
        this.MyInkCanvas.Strokes.Save(fs);
        fs.Close();
    }
}

private void LoadData(object sender, RoutedEventArgs e) {
    // Fill StrokeCollection from file.
    using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Open, FileAccess.Read))
    {
        StrokeCollection strokes = new StrokeCollection(fs);
        this.MyInkCanvas.Strokes = strokes;
    }
}

private void Clear(object sender, RoutedEventArgs e) {
    // Clear all strokes.
    this.MyInkCanvas.Strokes.Clear();
}
```

WPF applikation (exempel 2) - Andra fliken

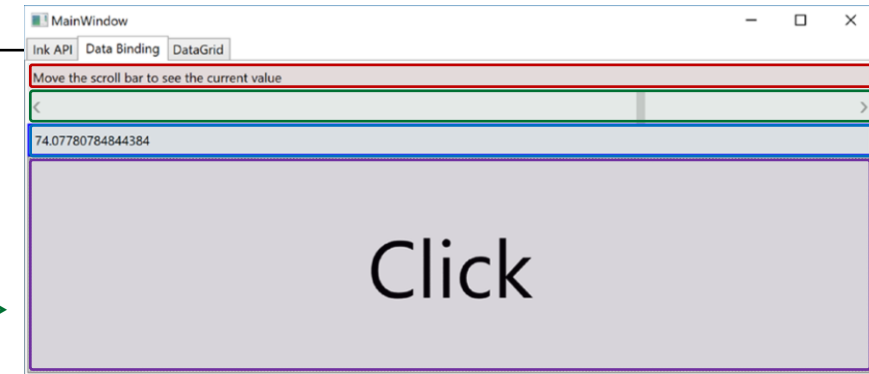
- I den andra fliken implementerar vi ett exempel på **databindning** mellan element i XAML filen.
- Vi börjar med nedanstående XAML som innehåller en **Label** med instruktioner till användaren samt en **ScrollBar**, ytterligare en **Label** och en **Button** (i en **StackPanel**).
- En **ScrollBar** kan anta värden mellan ett valt intervall (1-100 nedan) och innehåller ett **reglage** (och **vänster/högerpilar**) som kan skjutas fram och tillbaka för att ändra värdet inom intervallet.
- Vi vill använda **databindning** så att vår **Label** automatiskt visar vår **ScrollBars** nuvarande värde samt så att vår **ScrollBars** värde automatiskt bestämmer vår **Buttons** fontstorlek.

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>

    <!-- The scrollbar's value is the source of this data bind. -->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
      Minimum="1" Maximum="100" LargeChange="1" SmallChange="1"/>

    <!-- The label's content will be bound to the scroll bar. -->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2" Content="0"/>

    <!-- The button's font size will be bound to the scroll bar. -->
    <Button Content="Click" Height="200" FontSize="12"/>
  </StackPanel>
</TabItem>
```



Databindning

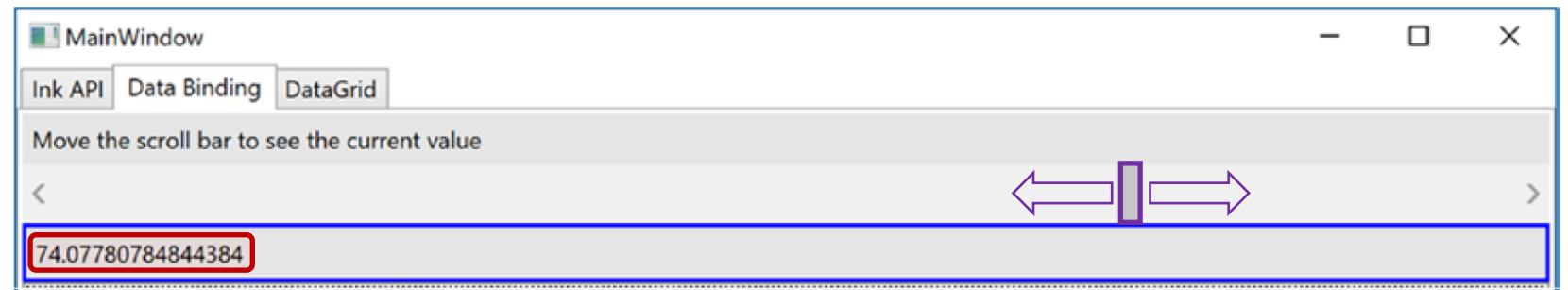
- Kontroller används ofta i **databindnings**-operationer.
- **Databindning (data binding)** innebär att man “kopplar” (binder) properties i en kontroll till datavärden, som kan ändras under tiden en applikation exekverar. På så sätt kan en kontroll i användargränssnittet visa tillståndet av en variabel i koden (om variabelvärdet ändras, så ändras automatiskt den “bundna” kontrollens property-värde i användargränssnittet).
- Exempelvis kan man använda databindning i följande fall:
 - Bocka/avbocka en **CheckBox** baserat på en boolesk property i ett visst objekt (klassinstans).
 - Binda en **Label** till ett heltal (integer) som representerar antalet filer i en folder.
 - Visa data från en relationell databas i en **DataGrid**.
- När man använder WPFs databindningsmotor, måste man vara medveten om skillnaden mellan **källan (source)** och **målet (destination)** av bindningsoperationen.
 - **Källan (source)** är själva datan (t.ex. en boolesk property eller relationell data).
 - **Målet (destination)** är en UI kontroll property (t.ex. **IsChecked** för en **CheckBox** eller **Text** för en **TextBox**).
- WPF har också support för **element bindning**, dvs att man kan binda en property av en UI kontroll till en property i en annan UI kontroll, t.ex. synligheten av en kontroll baserat på en **CheckBoxs IsChecked** property.
- Utöver **envägs-bindning (one-way binding)** som ovan, har WPF support för **tvåvägs-bindning (two-way binding)**, dvs så att både **källan (source)** uppdaterar **målet (destination)** och **målet (destination)** uppdaterar **källan (source)**. Detta innebär att **källan (source)** och **målet (destination)** kan hållas synkroniserade oberoende av var ändringen sker (**källan** eller **målet**).

{Binding} markup extension (exempel 2) - Andra fliken

- I XAML används en **{Binding} markup extension** för att skapa bindningar.

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
      Content="{Binding ElementName=mySB, Path=Value}" />
```

- I ovanstående XAML kod binds **ScrollBar**ns **Value** property till **Label**ns **Content** property:
 - Värdet för **Label**ns **Content** innehåller en **{Binding} markup extension** som indikerar en bindingsoperation.
 - I **ElementName** attributets värde anges **källan (source)**, som är namet på **ScrollBar**en.
 - I **Path** attributets värde anges **källans** property som skall bindas till, som är **ScrollBar**ns **Value**.
- Nu kommer **Label**ns **Content** alltid att visa **ScrollBar**ns värde (**Value**) då reglaget skjuts fram och tillbaka.



DataContext propertyn (exempel 2) - Andra fliken

- Man kan även separera bindningen till **källan** i **ElementName** och till dess property **Path värde** i en **{Binding}** markup extension genom att explicit sätta **DataContext** propertyn för kontrollen (målet) till **källan**.

```
<!-- Breaking object/value apart via DataContext -->  
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"  
      DataContext="{Binding ElementName=mySB}" Content="{Binding Path=Value}"/>
```

- Fördelen med att bryta ut bindningen till källan i **ElementName** genom att använda **DataContext** propertyn, är att alla subelement i element-trädet ärver dess **DataContext** värde.
- Genom denna konstruktionen kan man sätta samma källa i en element-samling, istället för att återupprepa **"{Binding ElementName=X, Path=Y}"** i flera kontroller.
- I nedanstående XAML sätts t.ex. **DataContext="{Binding ElementName=mySB}"** i en **StackPanel**, varför alla dess inneslutna element endast behöver sätta **Content="{Binding Path=Value}"**.

```
<!-- Note the StackPanel sets the DataContext property. -->  
<StackPanel Background="#FFE5E5" DataContext="{Binding ElementName=mySB}">  
  <Label Content="Move the scroll bar to see the current value"/>  
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB" Maximum="100" LargeChange="1" SmallChange="1"/>  
  <!-- Now both UI elements use the scrollbar's value in unique ways. -->  
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2" Content="{Binding Path=Value}"/>  
  <Button Content="Click" Height="200" FontSize="{Binding Path=Value}"/>  
</StackPanel>
```

TwoWay bindning (exempel 2) - Andra fliken

- ***Tvåvägsbindning (two-way binding)*** erhålls genom att sätta ***Mode=TwoWay*** i XAML koden.
- Här har ***Label***n bytts ut mot en ***TextBox***, där ***ScrollBar***ens ***Value*** property och ***TextBox***ens ***Text*** property är bundna till varandra med ***tvåvägs-bindning***.

```
<!-- Note the StackPanel sets the DataContext property. -->
<StackPanel Background="#FFE5E5E5" DataContext="{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB" Maximum="100" LargeChange="1" SmallChange="1"/>
  <!-- Now both UI elements use the scrollbar's value in unique ways. -->
  <TextBox x:Name="textboxSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
    Text="{Binding Path=Value, Mode=TwoWay}"/>
  <Button Content="Click" Height="200"
    FontSize="{Binding Path=Value}"/>
</StackPanel>
```

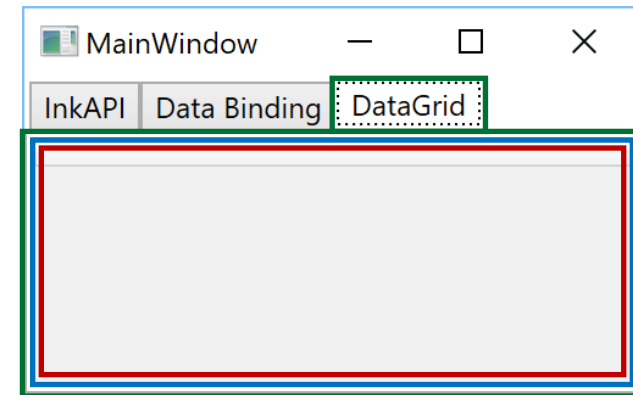
Databindning i kod

- Databinding kan också göras via Visual Studios property editor och i kod (istället för i XAML).
- Man kan också formatera den bundna datan.
- Se kursboken s. 1052 – 1054 för ett exempel.

WPF applikation (exempel 2) - Tredje fliken

- I den tredje fliken implementerar vi ett exempel på ***databindning*** mellan en ***DataGrid*** kontroll i XAML filen och en klass i ***code-behind*** filen.
- Vi börjar med att lägga till en ***DataGrid*** kontroll i en ***StackPanel***.

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
    <StackPanel>
        <DataGrid x:Name="gridCars" Height="288"/>
    </StackPanel>
</TabItem>
```



- ***DataGrid*** kontrollen är en ganska avancerad kontroll med mycket funktionalitet. Bland annat innehåller den en property ***ItemSource*** som kan tilldelas ***källan (source)***, där källan t.ex. kan vara en samlingsklass som innehåller objekt av någon klass med publika properties.

WPF applikation (exempel 2) - Bilar

- Vi skapar en bil klass (**Car**) som innehåller tre publika properties **Make**, **Color** och **PetName**.
- Därefter skapar vi en samlingsklass **List<Car>** som innehåller ett antal instaser av bilklassen.

```
public class Car
{
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }

    public Car() {}
    public Car(string make, string color, string petName)
    {
        Make = make;
        Color = color;
        PetName = petName;
    }
}
```

```
List<Car> cars = new List<Car>
{
    new Car {Make = "VW", Color = "Black", PetName = "Zippy"},
    new Car {Make = "Ford", Color = "Rust", PetName = "Rusty"},
    new Car {Make = "Saab", Color = "Black", PetName = "Mel"},
    new Car {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
    new Car {Make = "BMW", Color = "Black", PetName = "Bimmer"},
    new Car {Make = "BMW", Color = "Green", PetName = "Hank"},
    new Car {Make = "BMW", Color = "Pink", PetName = "Pinky"},
    new Car {Make = "Pinto", Color = "Black", PetName = "Pete"},
    new Car {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
};
```

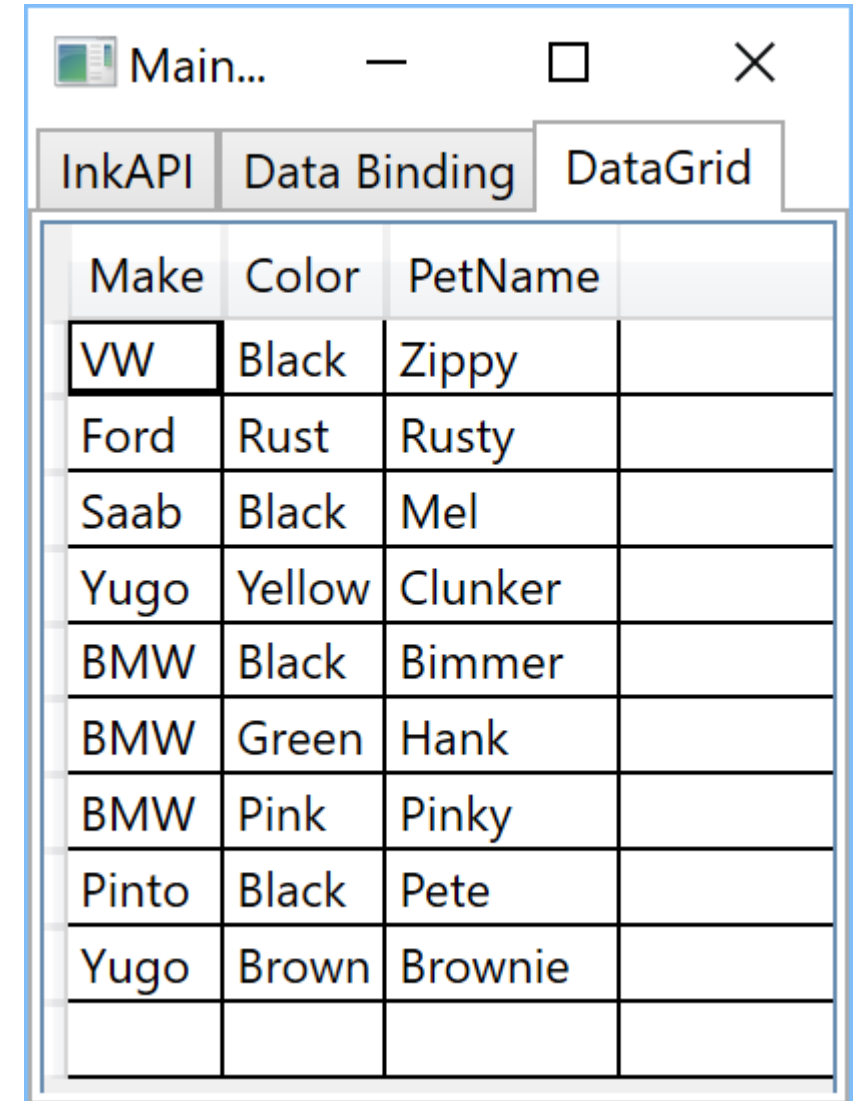
WPF applikation (exempel 2) - Bilar

- Slutligen tilldelar vi vårans instans *cars* av *List<Car>* till vårans *DataGrids ItemSource* property i fönstrets *code-behind* fil.

```
public MainWindow()  
{  
    this.InitializeComponent();  
    gridCars.ItemsSource = cars;  
}
```

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">  
    <StackPanel>  
        <DataGrid x:Name="gridCars" Height="288"/>  
    </StackPanel>  
</TabItem>
```

- Nu visas bilarnas tre property i *DataGrid* kontrollen.



The screenshot shows a WPF application window titled "Main..." with three tabs: "InkAPI", "Data Binding", and "DataGrid". The "DataGrid" tab is active, displaying a table with four columns: "Make", "Color", "PetName", and an empty column. The table contains the following data:

Make	Color	PetName	
VW	Black	Zippy	
Ford	Rust	Rusty	
Saab	Black	Mel	
Yugo	Yellow	Clunker	
BMW	Black	Bimmer	
BMW	Green	Hank	
BMW	Pink	Pinky	
Pinto	Black	Pete	
Yugo	Brown	Brownie	

WPF tutorial länkar

Några länkar som kan vara till hjälp för att lära sig olika WPF kontroller:

- <https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls>
- <https://www.wpf-tutorial.com>
- <https://www.wpftutorial.net>
- Den andra länken innehåller ett exempel på hur man skapar skräddarsydda dialoger (<https://www.wpf-tutorial.com/dialogs/creating-a-custom-input-dialog>).

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF kontroller

- Kategorier:
 - Input/output, t.ex. Button
 - Dekorationer, t.ex. ProgressBar
 - Media, t.ex. Image
 - Layout, t.ex. Grid
- En kontrolls **Content** kan vara:
 - **Simple** (sträng).
 - **Komplex** (nästlade kontroller i en layout manager).

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF kontroller

- Window
- DockPanel
- StackPanel
- WrapPanel
- Grid
 - ColumnDefinition
 - RowDefinition
 - GridSplitter
- ScrollBar
- Menu
 - MenuItem
- Separator
- ToolBar
- StatusBar
 - StatusBarItem
- TabControl
 - TabItem
- Expander
- InkCanvas
- Border
- Label
- Button
- TextBlock
- TextBox
- RadioButton
- ComboBox
 - ComboBoxItem
- DataGrid
- Ellipse
- CheckBox*
- ListBox*

Sammanfattning Windows Presentation Foundation (WPF) 2

Dialoger

- OpenFileDialog, SaveFileDialog, Egendefinierade dialoger.

Layout Managers (paneler)

- En kontrolls Content kan endast innehålla en kontroll/värde.
- En layout manager kan innehålla flera kontroller.
- En layout manager organiserar sina kontroller enligt en layoutalgoritm.
 - **Canvas**: använder **absolut positionering** av kontroller.
 - **WrapPanel**: kontroller placeras från **vänster till höger**, och **hoppas ner till nästa rad** då en rad är full (**Orientation=Horizontal** default).
 - **StackPanel**: kontroller placeras **som i en WrapPanel** (**Orientation=Vertical** default), **men sträcker ut kontroller** som inte fyller en rad/kolumn.
 - **Grid**: organiserar kontroller i ett **rutnät** som består av **rader, kolumner** och celler.
 - **Storleken** av rader/kolumner kan vara **absolut, automatisk** eller **procentuell**.
 - En **GridSplitter** kan användas för att **ändra storleken på en rad/kolumn runtime**.
 - **DockPanel**: placerar kontroller i **ytan** (**Top, Bottom, Left, Right**), där propertyn **LastChildFill** ofta används för placering i **Center** ytan.
- **Attached properties** används för att **positionera kontroller i** (vissa) **layout managers**.
- Properties **Margin** och **Padding** används för att bestämma **utrymmet mellan respektive inom kontroller**.
- En layout manager kan placeras i en **ScrollViewer** för att **automatiskt få ScrollBars** då **allt innehåll inte får plats**.

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Commands

- En typ som **implementerar ICommand** interfacet.

```
event EventHandler CanExecuteChanged
bool CanExecute(object parameter)
void Execute(object parameter)
```

- WPF innehåller många **inbyggda Command klasser** kategoriserade i de statiska klasserna:
 - ApplicationCommands, ComponentsCommands, MediaCommands, NavigationCommands, EditingCommands.
- Kan **definiera egna Command klasser** som **implementerar ICommand**.
- **WPF kontroller** (t.ex. MenuItem) **har en Command property** som kan **tilldelas ett Command objekt** (t.ex. ApplicationCommands.Copy).

Routed Events

- **Bubbling events propageras uppåt** det logiska GUI trädets.
- **Tunneling events propageras nedåt** det logiska GUI trädets.
- **Direkt events hanteras direkt** av elementet som skapade ett event.
- En **RoutedEvent** hanteras av ett element med en **lämplig händelsehanterare**.
- Om property **Handled=true** i en händelsehanterare, **avbryts propageringen** av ett event.
- Varje Bubbling event har ett tillhörande Tunneling event.
 - Motsvarande **Tunneling event börjar alltid med ordet Preview** (t.ex. MouseDown/PreviewMouseDown).
 - Ett **Tunneling event alltid triggas innan tillhörande Bubbling event**.

Sammanfattning Windows Presentation Foundation (WPF) 2

Databindning

- Man **binder en property (målet)** i en kontroll till **data (källan)**, där **propertyn uppdateras** när **datavärdet ändras**.
- Både **en- och tvåvägsbindning** är möjlig:
 - **Envägsbindning** innebär att **källan uppdaterar målet** (default).
 - **Tvåvägsbindning** innebär att **källan och målet uppdaterar varandra**.
- Man **kan binda**:
 - **Mellan element** i en XAML fil.
 - **Mellan element** i XAML filen **och objekt** i Code-Behind filen.
- **I XAML används en {Binding} markup extension** för att binda källan till en property (målet) i ett element, där:
 - **ElementName** anger **objektet** som skall bindas till.
 - **Path** anger **propertyn** i **objektet** som skall bindas till.
 - **Mode** anger envägsbindning (Mode=**OneWay**) eller tvåvägsbindning (Mode=**TwoWay**).
 - Exempel: `Content="{Binding ElementName=myStatusBar, Path=Value, Mode=OneWay}"`
- I XAML har varje element en property **DataContext** som kan **tilldelas en referens till ett objekt**.
 - Alla **subelement** ärver samma **DataContext** tilldelning, varför **endast propertyn** på den bundna objektet **behöver tilldelas** i subelement.
 - Exempel: `DataContext="{Binding ElementName=myStatusBar}" Content="{Binding Path=Value, Mode=OneWay}"`
- En del kontroller har en **ItemsSource** property som kan **tilldelas objektet** som skall bindas till (t.ex. i Code-Behind filen):
 - Exempel: `myDataGrid.ItemsSource = cars; // cars av typ List<Car>`

Sammanfattning Windows Presentation Foundation (WPF) 2

Arbeta med WPF kontroller i Visual Studio

- **Drag-and-drop:** a kontroller från **ToolBoxen** till **Designytan** eller **XAML editorn**.
 - För att **lägga till fler kontroller** i **ToolBoxen**, högerklicka på **ToolBoxen**, välj **Choose Items**, markera fler kontroller, och klicka på **OK**.
- Namnge en kontroll i XAML med **x>Name** om du vill **komma åt kontrollen** i **Code-Behind** filen.
- Använd **Properties och Events editorn** för att **editera** egenskaper (**properties**) och händelser (**events**) för en kontroll.
- Använd **Document Outline editorn** för att:
 - **Dölja/visa, låsa/låsa upp kontroller** (design time).
 - **Flytta kontroller** i trädstrukturen.
 - **Organisera** och byta **layout managers**.
- Använd den grafiska **Designytan** för att t.ex. **skapa nya rader/kolumner** för en **Grid** panel.

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Exempel 1 (Ordbehandlare med stavningskontroll)

• Kontroller

- Window (Title, Height, Width, CommandBindings, WindowStartupLocation, Close()).
- DockPanel (Dock, LastChildFill, HorizontalAlignment, Background, BorderBrush)
- Menu (HorizontalAlignment, Background, BorderBrush).
- MenuItem (Header, Command, MouseEnter, MouseLeave, Click)
- Separator
- ToolBar
- Button (Content, Cursor, MouseEnter, MouseLeave, Click)
- StatusBar (Background)
- StatusBarItem
- TextBlock (Name, Text)
- Grid (Background, ColumnDefinitions, Column, Row)
- ColumnDefinition
- GridSplitter (Width, Background)
- StackPanel (VerticalAlignment, Orientation)
- Label (Name, Content, FontSize, Margin)
- Expander (Name, Header, Margin, IsExpanded)
- TextBox (Name, SpellCheck.IsEnabled, AcceptsReturn, FontSize, BorderBrush, VerticalScrollBarVisibility, HorizontalScrollBarVisibility, CaretIndex, GetSpellingError())

■ = property
■ = event
■ = method

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Exempel 1 (Ordbehandlare med stavningskontroll)

- Andra klasser

- SpellingError (**Suggestions**)
- ApplicationCommands.*
- CommandBinding (**CanExecute**, **Executed**)
 - Konstruktorn tar en **ICommand** som in-parameter (t.ex. ApplicationCommands.Help).
- CommandBindings (**Add()**)
 - Innehåller **CommandBinding** objekt.
- MessageBox (**Show()**)
- OpenFileDialog (**Filter**, **FileName**, **ShowDialog()**)
- SaveFileDialog (**Filter**, **FileName**, **ShowDialog()**)

 = property
 = event
 = method

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Exempel 2 (TabControl med kontroller, events och databindning)

- **Flikar**

- Flik 1: RadioButton, ComboBox och InkCanvas.
- Flik 2: Databindning mellan kontroller (Label och ScrollBar, TextBox och ScrollBar).
- Flik 3: Databindning mellan kontroller (DataGrid) och objekt (List<Car>).

- **Kontroller (huvudfönstret)**

- Window (**Title, Height, Width, WindowStartupLocation**, **Close()**).
- TabControl (**Name, HorizontalAlignment, VerticalAlignment**)
- TabItem (**Header**)
- StackPanel (**Background**)



Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Exempel 2 (TabControl med kontroller, events och databindning)

• Kontroller (flik 1)

- ToolBar (**Name**, **Height**)
- Separator
- Border (**Margin**, **Width**, **VerticalAlignment**)
- WrapPanel
- StackPanel (**Orientation**, **Tag**)
- RadioButton (**Name**, **Margin**, **GroupName**, **Content**, **IsChecked**, **Click**, **Checked**, **Unchecked**)
- ComboBox (**Name**, **Width**, **Margin**, **SelectedIndex**, **SelectedItem**, **SelectedValue**, **SelectionChanged**)
- ComboBoxItem (**Content**)
- Grid (**ColumnDefinitions**)
- ColumnDefinition (**Width**)
- Button (**Name**, **Margin**, **Width**, **Content**, **Click**)
- InkCanvas (**Name**, **Background**, **DefaultDrawingAttributes**, **EditingMode**, **Strokes**)
- Label (**FontSize**, **HorizontalAlignment**, **VerticalAlignment**, **Content**)
- Ellipse (**Fill**, **Height**, **Width**)

CheckBox*

ListBox*

• Andra klasser (flik 1)

- DefaultDrawingAttributes (**Color**)
- InkCanvasEditingMode
- StrokeCollection (**Save()**, **Clear()**)
- Color
- ColorConverter (**ConvertFromString()**)
- FileStream (**Close()**)

 = property
 = event
 = method

Sammanfattning Windows Presentation Foundation (WPF) 2

WPF Exempel 2 (TabControl med kontroller, events och databindning)

- **Kontroller (flik 2)**

- StackPanel (**Width**, **Background**, **DataContext**)
- Label (**Name**, **Height**, **BorderBrush**, **BorderThickness**, **Content**, **DataContext**)
- TextBox (**Name**, **Height**, **BorderBrush**, **BorderThickness**, **Text**)
- Button (**Content**, **Height**, **FontSize**)
- ScrollBar (**Name**, **Orientation**, **Height**, **Minimum**, **Maximum**, **LargeChange**, **SmallChange**, **Value**)

- **Andra klasser (flik 2)**

- Binding (**ElementName**, **Path**, **Mode**)

- **Kontroller (flik 3)**

- StackPanel
- DataGrid (**Name**, **Height**, **ItemSource**)

- **Andra klasser (flik 3)**

- Binding (**ElementName**, **Path**, **Mode**)
- Car (**Make**, **Color**, **PetName**) – egendefinierad klass
- List<Car>

