FPT Education
FPT UNIVERSITY

Microsoft® .NET

Building Windows Presentation Foundation (WPF) Application

Objectives

- Overview Windows Presentation Foundation (WPF)
- Overview XAML(eXtensible Application Markup Language) in WPF
- Explain about Controls and Layouts in WPF
- Explain about Styles and Templates in WPF
- Explain about WPF Data-Binding Model
- Demo create WPF application by dotnet CLI and Visual Studio.NET
- Demo access to the database by WPF Application
- Explain about MVVM Pattern (Model-View-ViewModel)

Overview Windows Presentation Foundation (WPF)

WPF History

- Initial Development (Early 2000s) - Windows Presentation Foundation (WPF) was developed by Microsoft as a part of the .NET Framework. It was introduced during the development of Windows Vista and was released as a part of .NET Framework 3.0 in 2006.
- WPF in .NET Core and .NET 5/6/7/8
  - Microsoft started the effort to port WPF to .NET Core, aiming to provide a unified platform for building desktop applications across Windows, macOS, and Linux.
  - With the release of .NET 5, 6, 7, 8, WPF is now fully supported on .NET Core, providing improved performance, compatibility, and cross-platform capabilities.

What is Windows Presentation Foundation?

- Windows Presentation Foundation (WPF) is a UI framework that creates desktop client applications
- The WPF development platform supports a broad set of application development features, including an application model, resources, controls, graphics, layout, data binding, documents, and security
- The framework is part of .NET, so if we have previously built applications with .NET using ASP.NET or Windows Forms, the programming experience should be familiar

What is Windows Presentation Foundation?

- WPF uses the Extensible Application Markup Language (XAML) to provide a declarative model for application programming

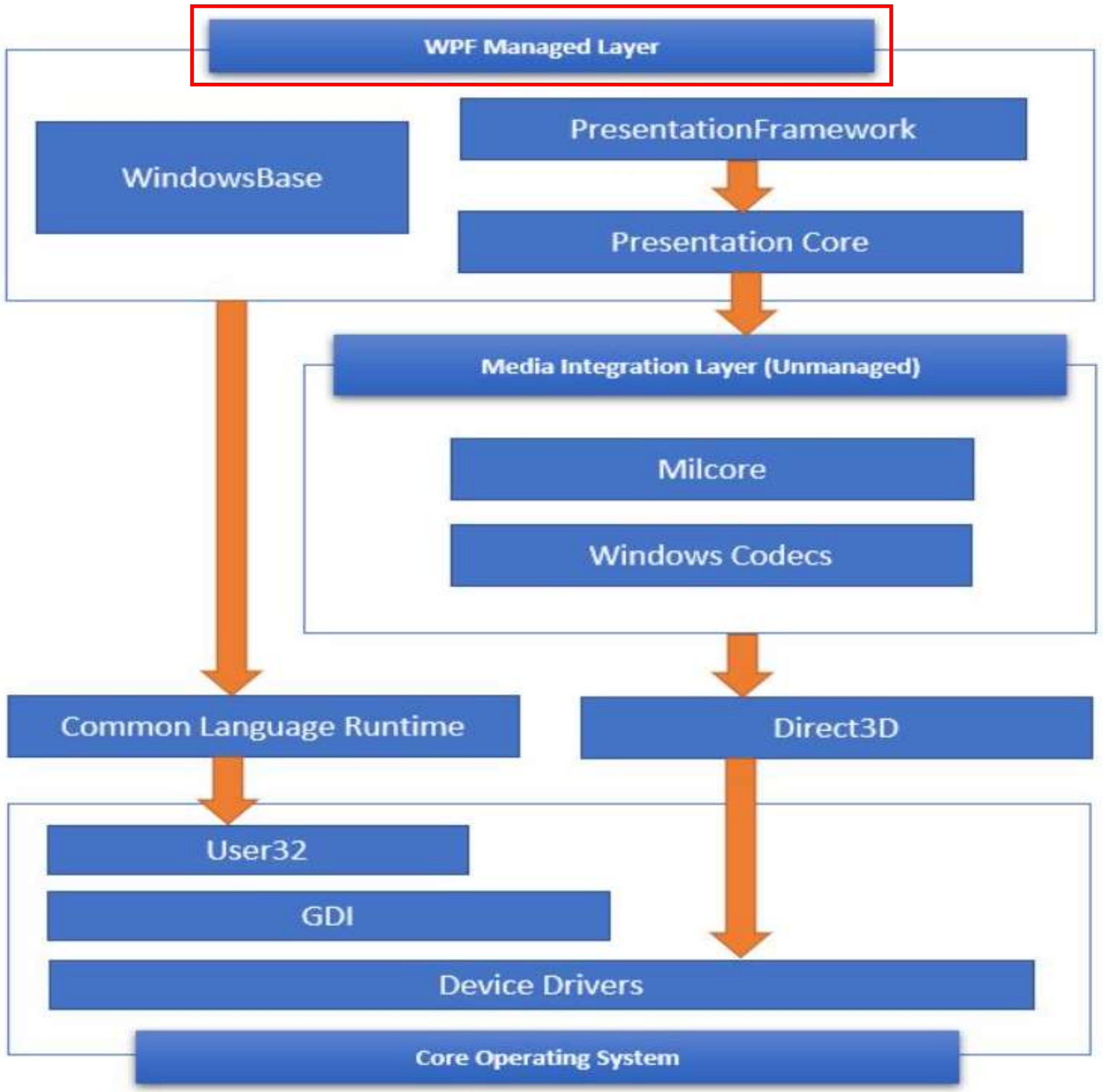
- WPF applications are based on a vector graphics architecture. This enables applications to look great on high DPI (Dots per inch) monitors, as they can be infinitely scaled

- WPF also includes a flexible hosting model, which makes it straightforward to host a video in a button, for example

- The visual designer provided in Visual Studio makes it easy to build WPF application, with drag-in-drop and/or direct editing of XAML markup

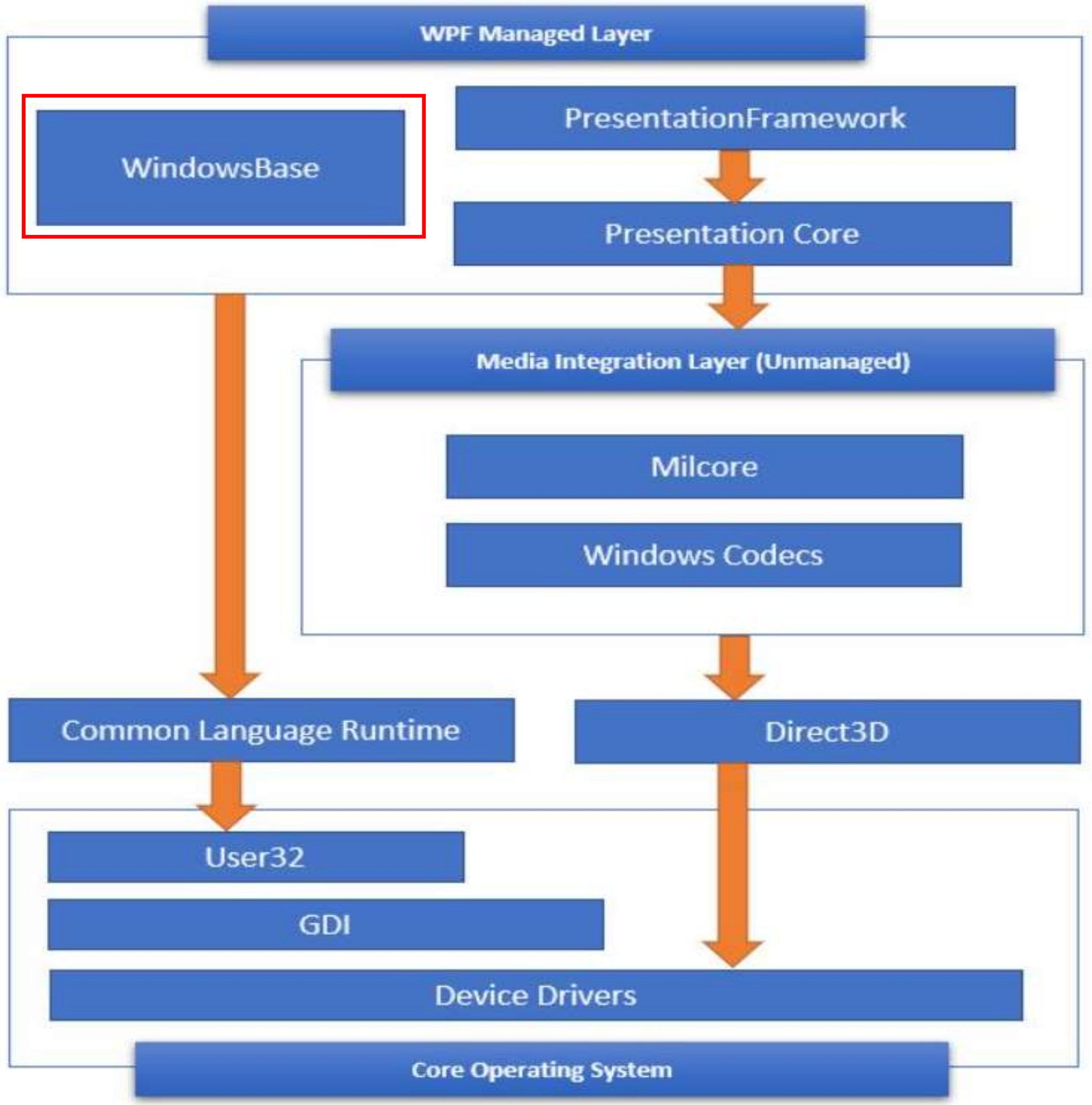
WPF Architecture

- Managed Layer: The layer is composed of three different services:
  - **PresentationFramework.dll**: This DLL provides the basic types to build a WPF application, such as windows, controls, shapes, media, documents, animation, data bindings, style and many more
  - **PresentationCore.dll**: This DLL provides basic types like UIElement and Visual. The UIElement defines the actions and element layout properties and provides classes to override them if required



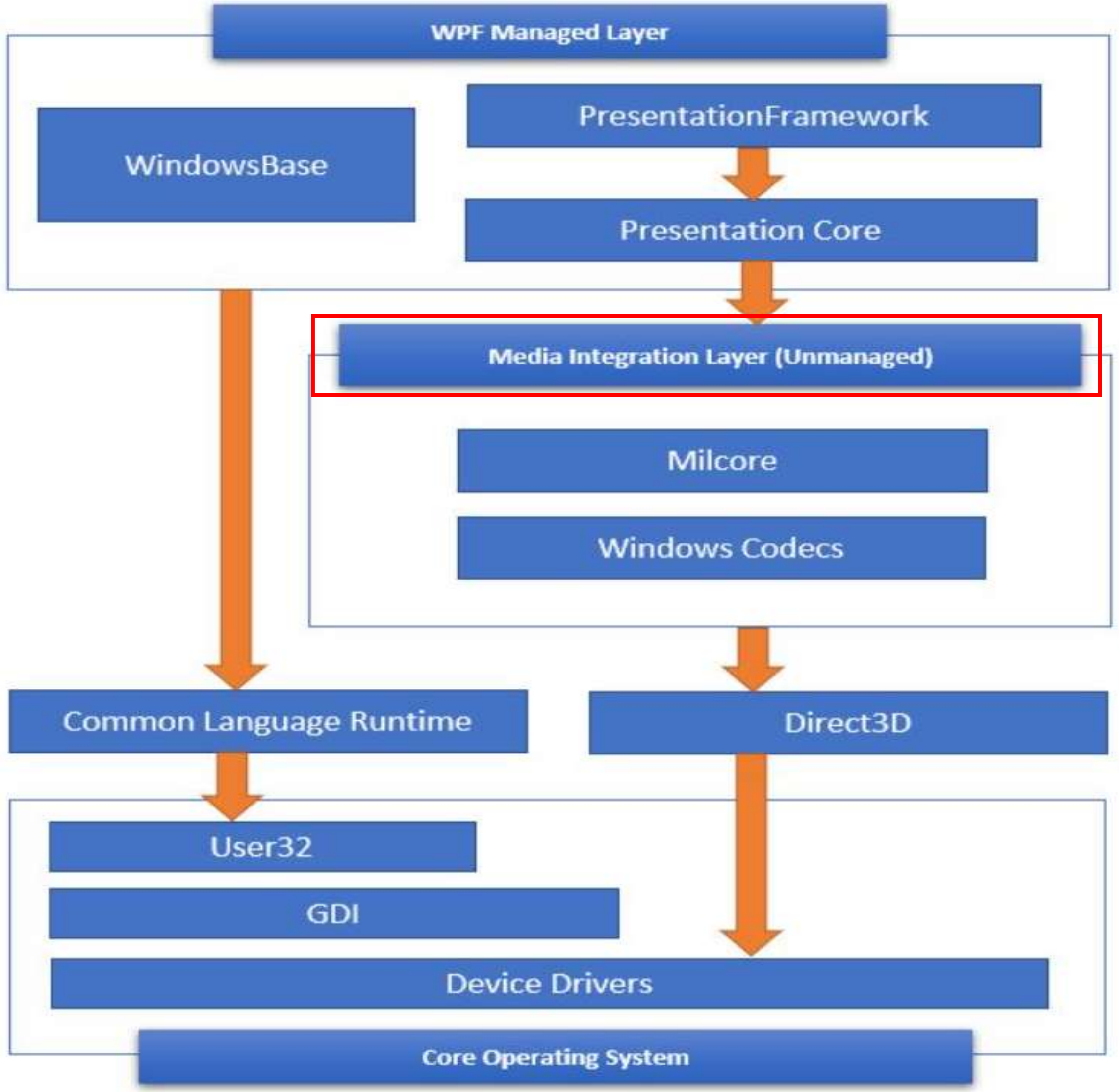
WPF Architecture

- **WindowBase.dll:** This DLL holds the WPF basic types like DependencyProperty, DependencyObject, DispatcherObject, and other types. The important one is given below:
  - DependencyProperty: provides a new property system that can enable or disable function like data binding, define attach properties, etc
  - DependencyObject: is the base of every WPF types and provides the function to enable property notification
  - DispatcherObject: This class provides a way for thread safety and threads other than Dispatcher created cannot directly access it



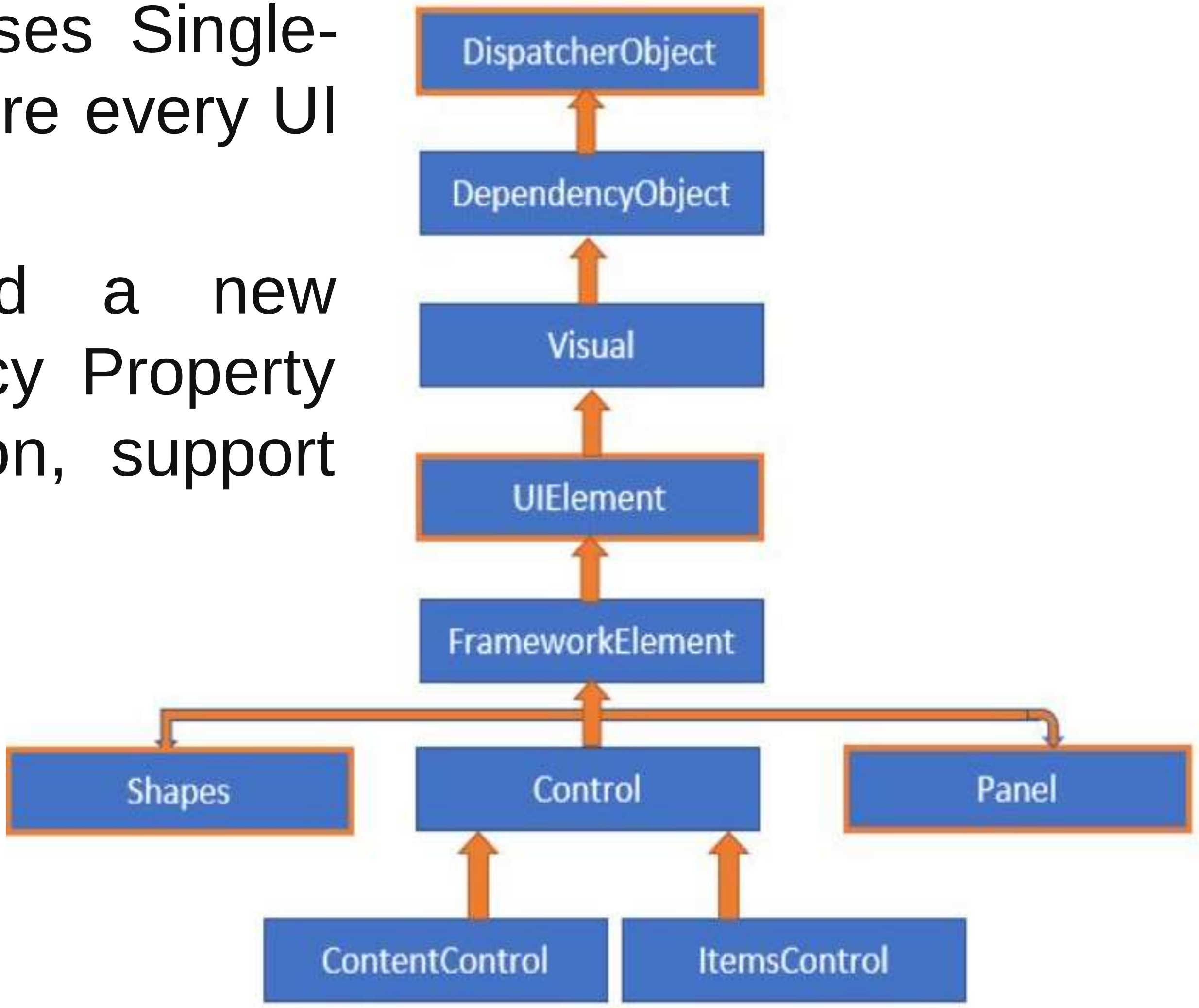
WPF Architecture

- Unmanaged Layer: This layer consist of two different services:
  - **Milcore.dll**: This is the media integration library or milcore that provides direct interaction with the DirectX and renders all the UI elements through this engine
  - **WindowsCodecs.dll**: This DLL provides the services for imaging like displaying, scaling, etc
  - **Direct3D**: This DLL provides access to low-level API which helps in rendering in WPF
  - **User32**: This is the basic core OS functionality that every application on Windows uses



Basic Class Hierarchy of WPF Types

- **DispatcherObject**: WPF application uses Single-Thread Affinity (STA) model and therefore every UI element is owned by a single thread
- **DependencyObject**: WPF introduced a new property system called the Dependency Property having features like change notification, support data bindings, attached properties, etc
- **Visual**: The Visual class defines all the properties required for rendering, clipping, transforming, bounding, and hit test. All the user interface controls like Button, ListBox derive from this class



Basic Class Hierarchy of WPF Types

- **UIElement**: This class adds the basic functionality of layout, input, focus, and events to UI elements and sets the basic foundation of the layout process
- **FrameworkElement**: This class extends the functionality provided by the UIElement, and override the layout for framework level implementations
- **Shapes**: This class is the base class for shape elements like Line, Ellipse, Polygon, Path, etc
- **Controls**: This namespace contains all the elements that help in interacting with the user. Few of the control like Textbox, Button, Listbox, Menu, etc are present in this namespace. Font, Background color, and control appearances support via templates support are added from this namespace

Basic Class Hierarchy of WPF Types

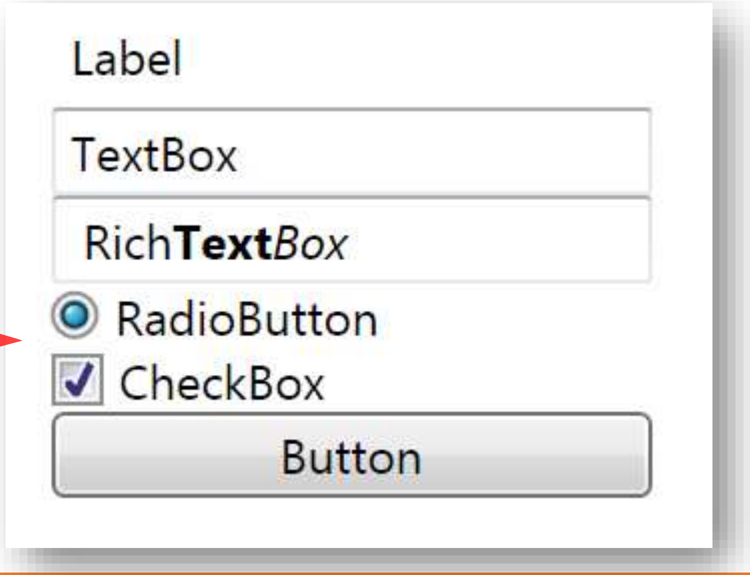
- **ContentControl**: This is the base class for all the control that supports only single content. Control from Label, Button, Windows, etc. The appearance of the control can be enhanced using a data template
- **ItemsControl**: This is the base class for all the control that displays a list of items and includes controls like, ListBox, TreeView, Menus, Toolbar, etc. ControlTemplate can be used to change the appearance of the control and ItemsTemplate can be applied to define how the objects will be displayed on the control
- **Panel**: This class is the base class of all the layout container elements. The class can host child objects and provides service to position and arrange child objects in the user interface. Control like Grid, Canvas, DockPanel, StackPanel, WrapPanel, etc derives from this class

WPF Capabilities and Features

- **Providing a Separation of Concerns via XAML**
  - One of the most compelling benefits is that WPF provides a way to cleanly separate the look and feel of a GUI application from the programming logic
  - Using XAML, it is possible to define the UI of an application via XML markup. This markup (ideally generated using tools such as Microsoft Visual Studio or Blend for Visual Studio) can then be connected to a related C# code file to provide the guts of the program’s functionality
  - XAML allows us to define not only simple UI elements (buttons, grids, list boxes, etc.) in markup but also interactive 2D and 3D graphics, animations, data-binding logic, and multimedia functionality (such as video playback)

XAML sample

```
<Window ... >
...
    <Label>Label</Label>
    <TextBox>TextBox</TextBox>
    <RichTextBox ... />
    <RadioButton>RadioButton</RadioButton>
    <CheckBox>CheckBox</CheckBox>
    <Button>Button</Button>

</Window>
```



WPF Capabilities and Features

- **Providing an Optimized Rendering Model**
  - The WPF programming model is quite different, in that GDI is not used when rendering graphical data. All rendering operations (e.g., 2D graphics, 3D graphics, animations, control rendering, etc.) now make use of the DirectX API
  - The first obvious benefit is that our WPF applications will automatically take advantage of hardware and software optimizations
  - As well, WPF applications can tap into very rich graphical services (blur effects,anti-aliasing, transparency, etc.) without the complexity of programming directly against the DirectX AP
  - *If we want to build a desktop application that requires the fastest possible execution speed (such as a 3D video game), unmanaged C++ and DirectX are still the best approach*

WPF Capabilities and Features

- **Simplifying Complex UI Programming**
  - A number of layout managers (far more than Windows Forms) to provide extremely flexible control over the placement and repositioning of content
  - Use of an enhanced data-binding engine to bind content to UI elements in a variety of ways and a built-in style engine, which allows us to define “themes” for a WPF application
  - Use of vector graphics, which allows content to be automatically resized to fit the size and resolution of the screen hosting the application
  - Support for 2D and 3D graphics, animations, and video and audio playback
  - A rich typography API, such as support for XML Paper Specification (XPS) documents, fixed documents (WYSIWYG), flow documents, and document annotations (e.g., a Sticky Notes API)
  - Support for interoperating with legacy GUI models (e.g., Windows Forms, ActiveX, and Win32 HWNDs)

The WPF Assemblies

- The following table will describe the key assemblies used to build WPF applications, each of which must be referenced when creating a new project:

Assembly	Description
PresentationCore	This assembly defines numerous namespaces that constitute the foundation of the WPF GUI layer. For example, this assembly contains support for the WPF Ink API, animation primitives, and numerous graphical rendering types
PresentationFramework	This assembly contains a majority of the WPF controls, the Application and Window classes, support for interactive 2D graphics, and numerous types used in data binding
System.Xaml.dll	This assembly provides namespaces that allow us to program against a XAML document at runtime. By and large, this library is useful only if we are authoring WPF support tools or need absolute control over XAML at Runtime
WindowsBase.dll	This assembly defines types that constitute the infrastructure of the WPF API, including those representing WPF threading types, security types, various type converters, and support for dependency properties and routed Events

The WPF Namespaces

- The following table describes the role of some of the important namespaces in WPF:

Assembly	Description
System.Windows	This is the root namespace of WPF. Here, we will find core classes (such as Application and Window) that are required by any WPF desktop project
System.Windows.Controls	This contains all of the expected WPF widgets, including types to build menu systems, tooltips, and numerous layout managers
System.Windows.Documents	This contains types to work with the documents API, which allows us to integrate PDF-style functionality into our WPF applications, via the XML Paper Specification (XPS) protocol
System.Windows.Ink	This provides support for the Ink API, which allows us to capture input from a stylus or mouse, respond to input gestures, and so forth. This is useful for Tablet PC programming; however, any WPF can make use of this API

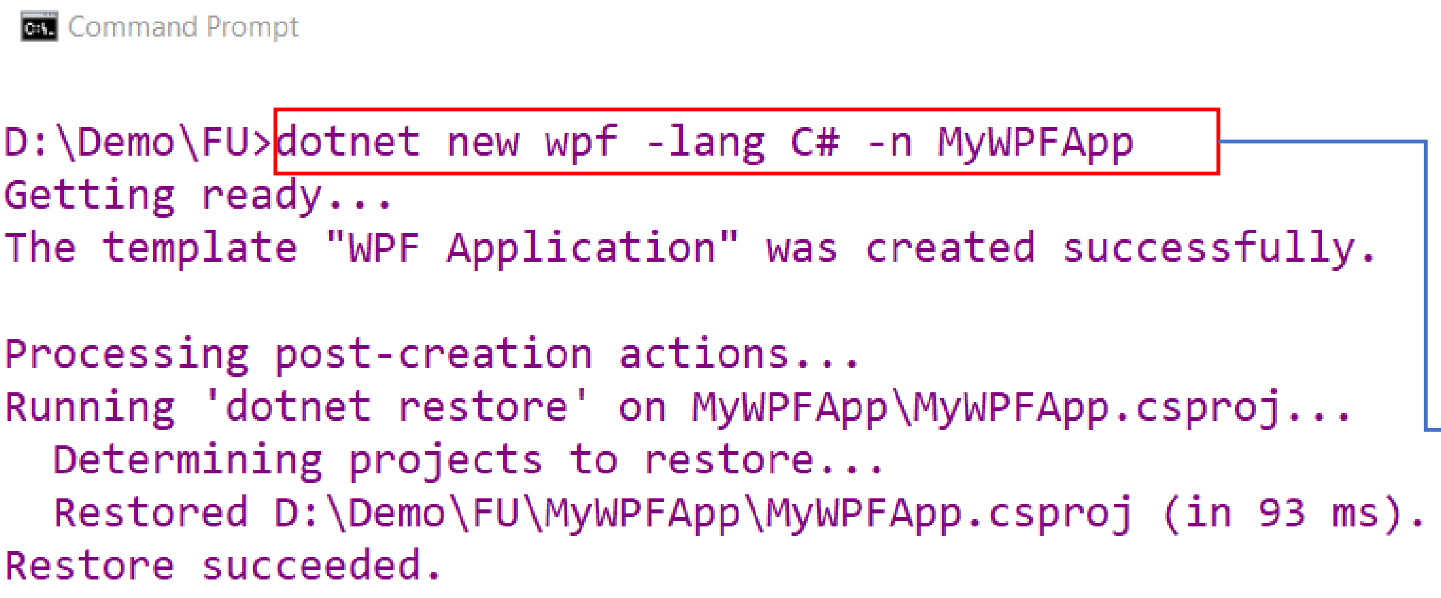
The WPF Namespaces

Namespace	Description
System.Windows.Markup	This namespace defines a number of types that allow XAML markup (and the equivalent binary format, BAML) to be parsed and processed programmatically
System.Windows.Media	This is the root namespace to several media-centric namespaces. Within these namespaces we will find types to work with animations, 3D rendering, text rendering, and other multimedia primitives
System.Windows.Navigation	This namespace provides types to account for the navigation logic employed by XAML browser applications (XBAPs) as well as standard desktop applications that require a navigational page model
System.Windows.Shapes	This defines classes that allow us to render interactive 2D graphics that automatically respond to mouse input
System.Windows.Data	This contains types to work with the WPF data-binding engine, as well as support for data-binding templates

Demo 01: Create a WPF Application using dotnet CLI

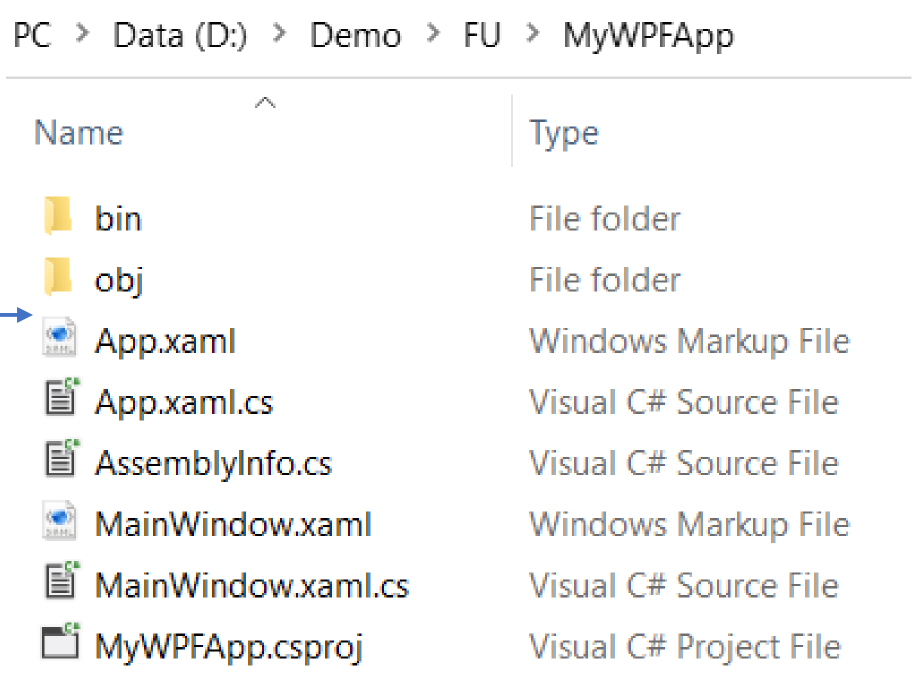
1. Install package: **dotnet-sdk-5.0.102-win-x64.exe** and open Command Prompt dialog

2. Create WPF App named **MyWPFApp** with C# language



Command Prompt

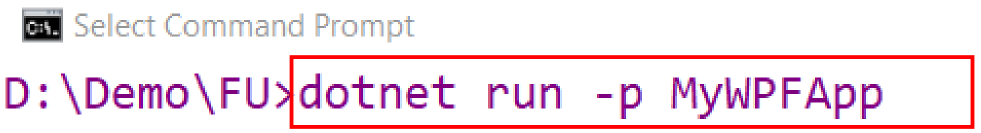
```
D:\Demo\FU>dotnet new wpf -lang C# -n MyWPFApp
Getting ready...
The template "WPF Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on MyWPFApp\MyWPFApp.csproj...
  Determining projects to restore...
  Restored D:\Demo\FU\MyWPFApp\MyWPFApp.csproj (in 93 ms).
Restore succeeded.
```



PC > Data (D:) > Demo > FU > MyWPFApp

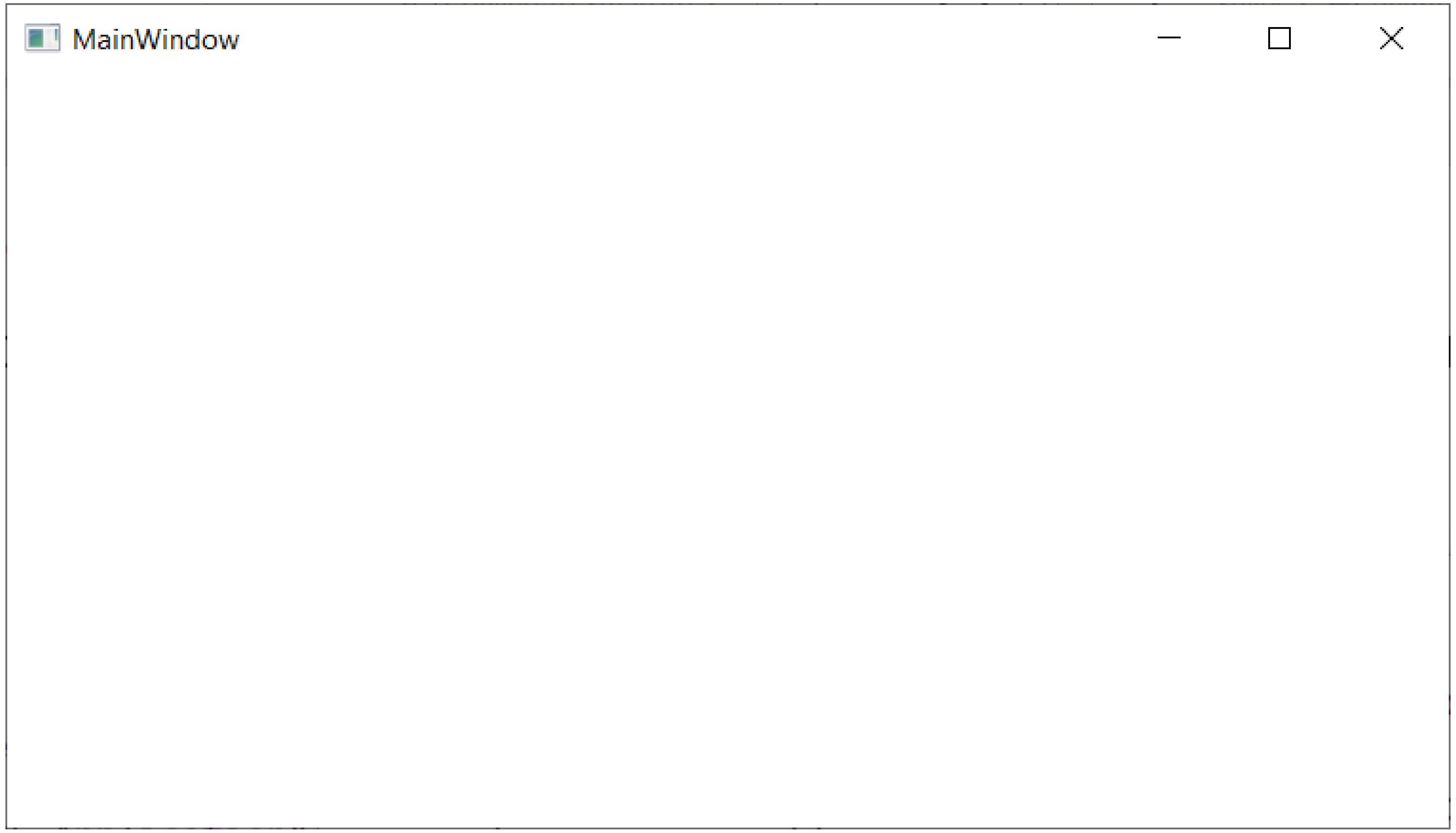
Name	Type
bin	File folder
obj	File folder
App.xaml	Windows Markup File
App.xaml.cs	Visual C# Source File
AssemblyInfo.cs	Visual C# Source File
MainWindow.xaml	Windows Markup File
MainWindow.xaml.cs	Visual C# Source File
MyWPFApp.csproj	Visual C# Project File

3. Run ***MyWPFApp*** application



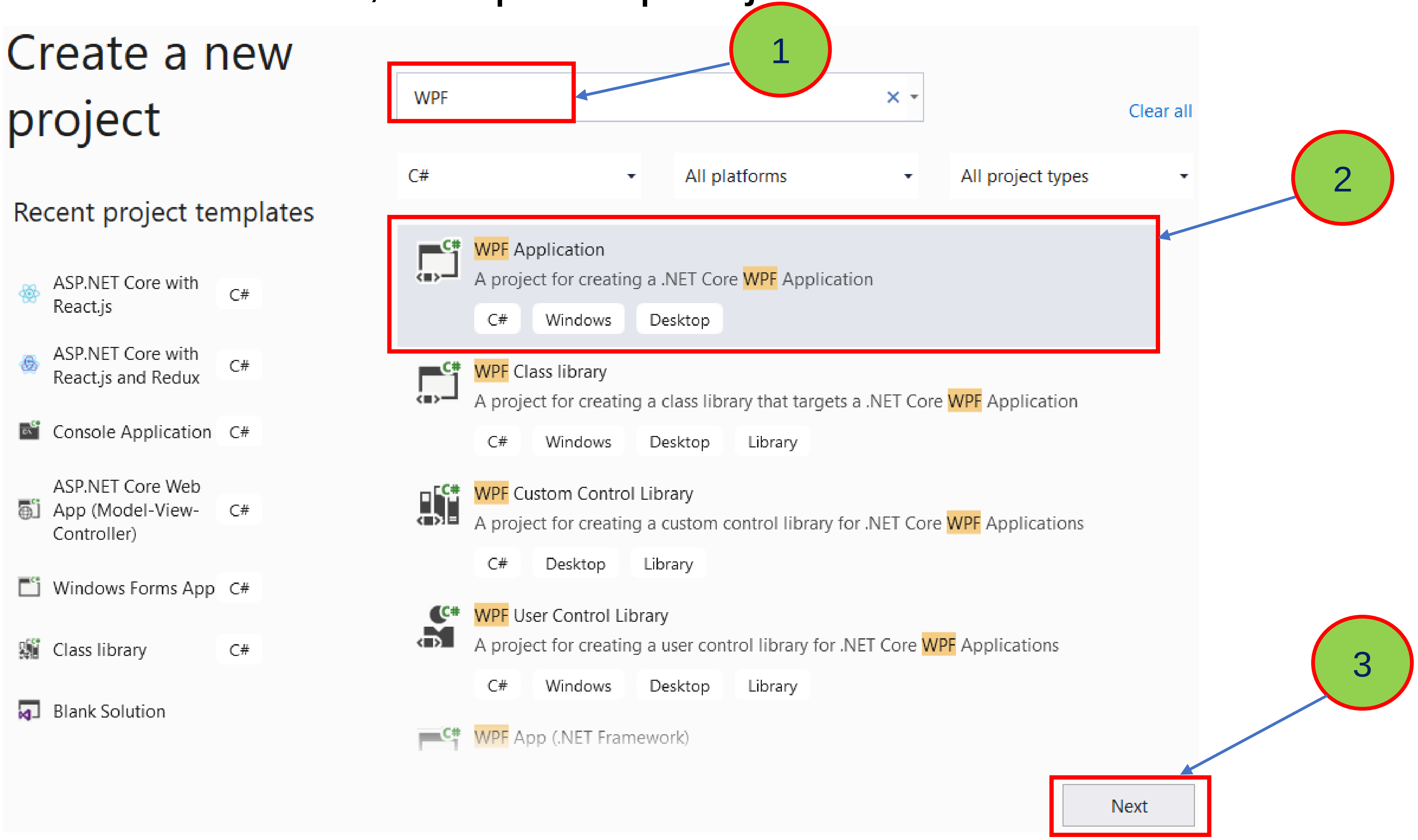
Select Command Prompt

```
D:\Demo\FU>dotnet run -p MyWPFApp
```



Demo 02: Create a WPF Application using Visual Studio.NET

1. Open Visual Studio.NET , File | New | Project



Create a new project

WPF

Clear all

C# | All platforms | All project types

Recent project templates

ASP.NET Core with React.js	C#
ASP.NET Core with React.js and Redux	C#
Console Application	C#
ASP.NET Core Web App (Model-View-Controller)	C#
Windows Forms App	C#
Class library	C#
Blank Solution	

WPF Application
A project for creating a .NET Core WPF Application
C# Windows Desktop

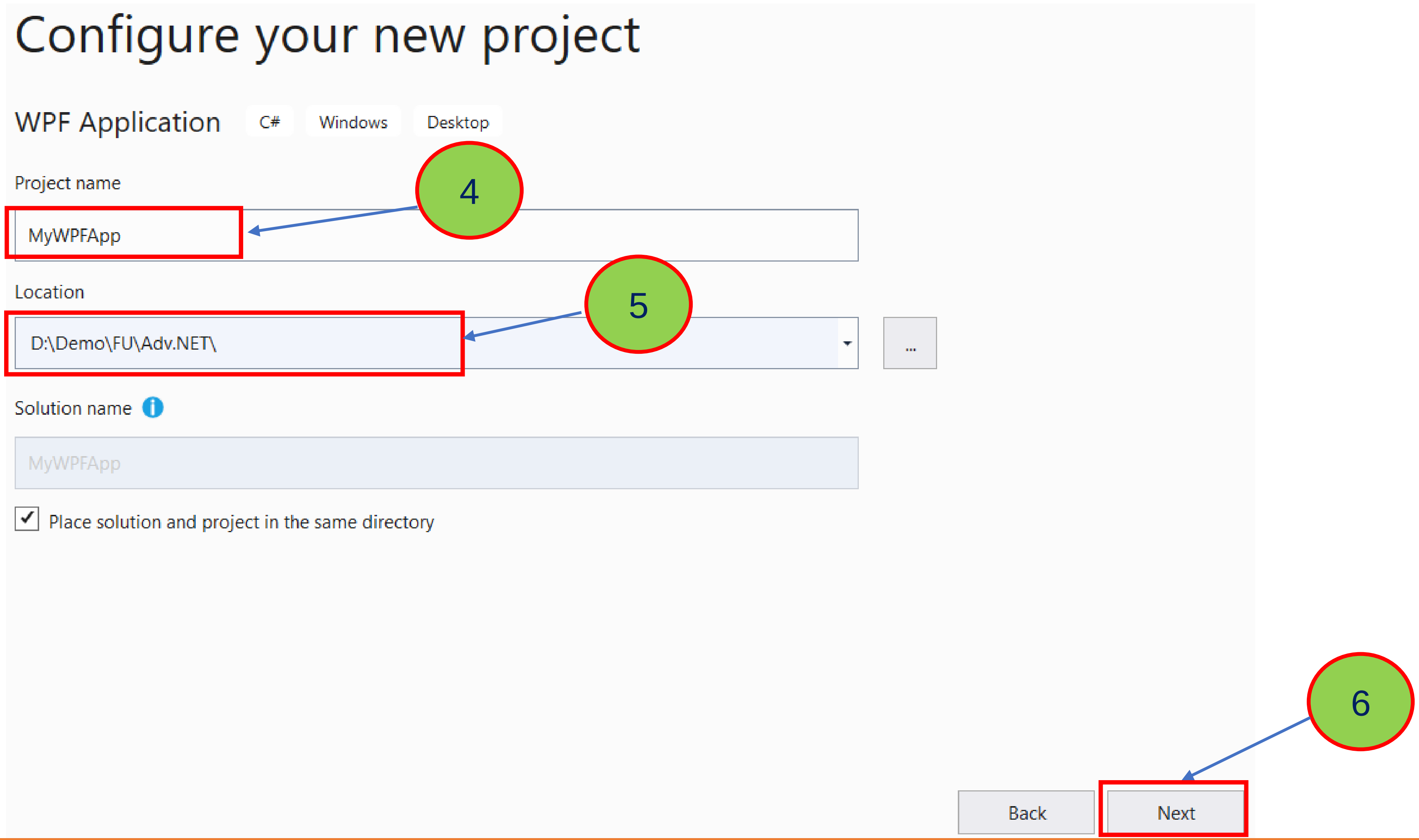
WPF Class library
A project for creating a class library that targets a .NET Core WPF Application
C# Windows Desktop Library

WPF Custom Control Library
A project for creating a custom control library for .NET Core WPF Applications
C# Desktop Library

WPF User Control Library
A project for creating a user control library for .NET Core WPF Applications
C# Windows Desktop Library

WPF App (.NET Framework)

2. Fill out **Project name**: MyWPFApp and **Location** then click **Next**



Configure your new project

WPF Application C# Windows Desktop

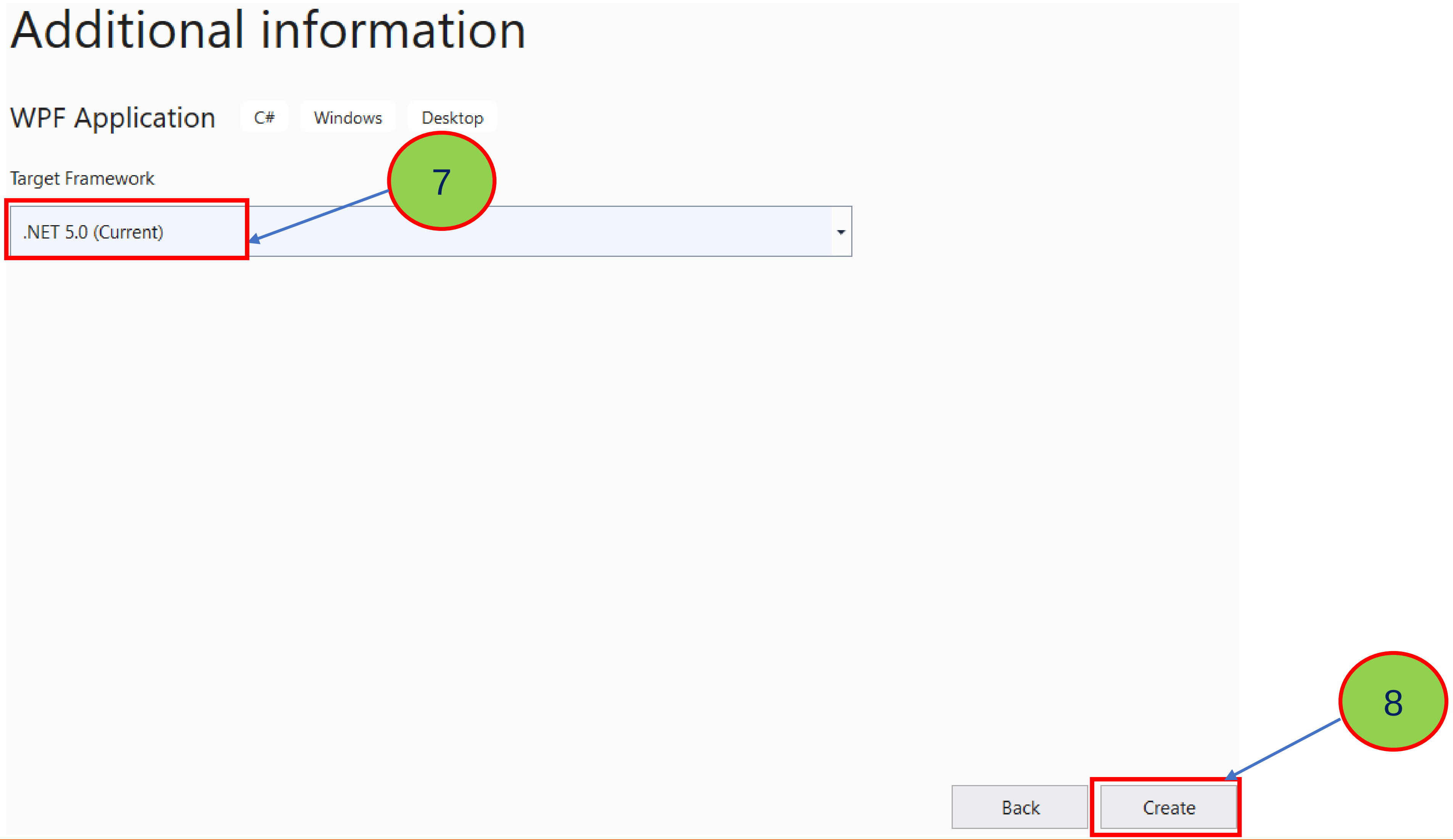
Project name: MyWPFApp

Location: D:\Demo\FU\Adv.NET\

Solution name: MyWPFApp

☑ Place solution and project in the same directory

3. Choose **Target Framework**: .NET 5.0 (Current) then click **Create**

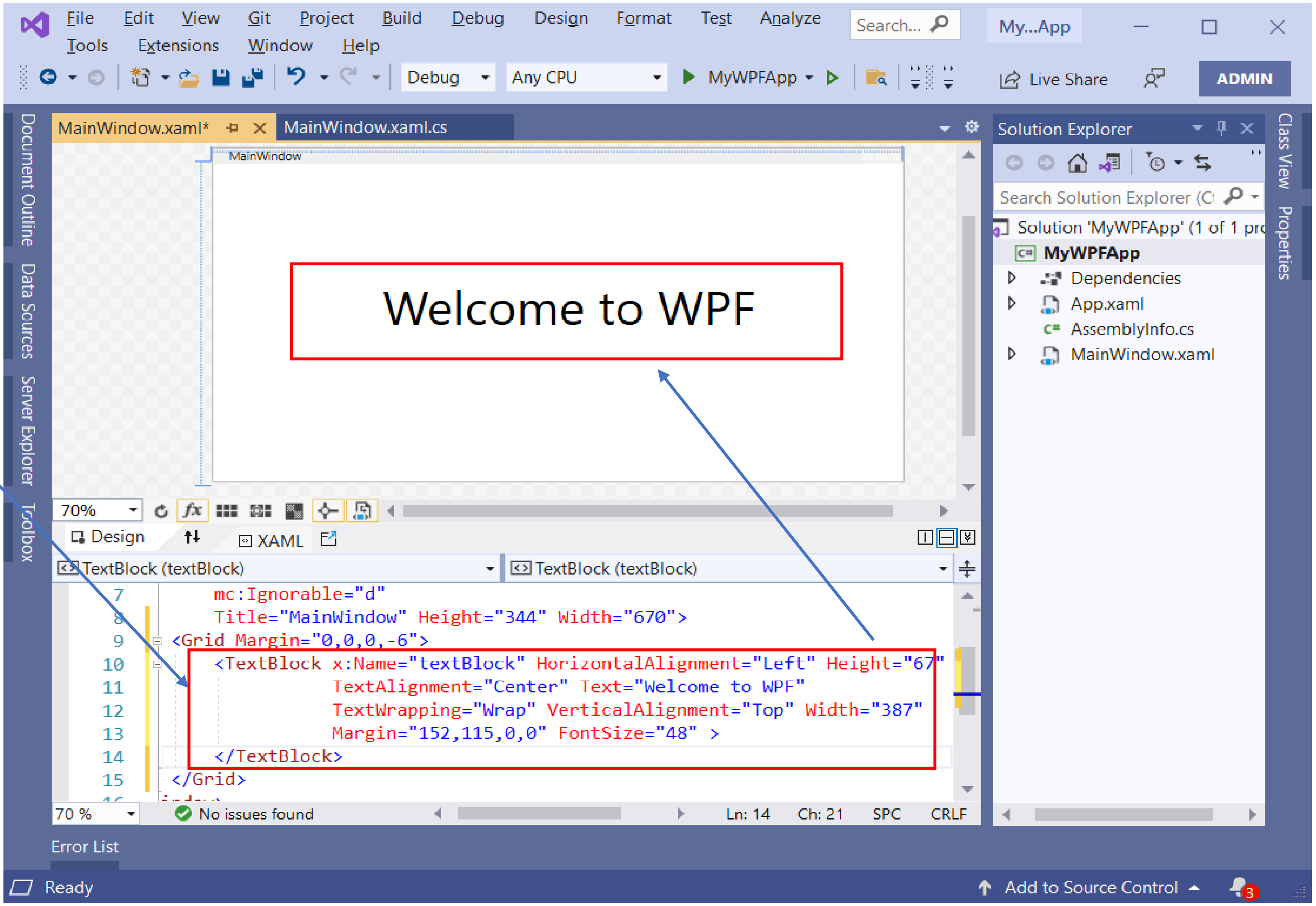

Additional information

WPF Application C# Windows Desktop

Target Framework

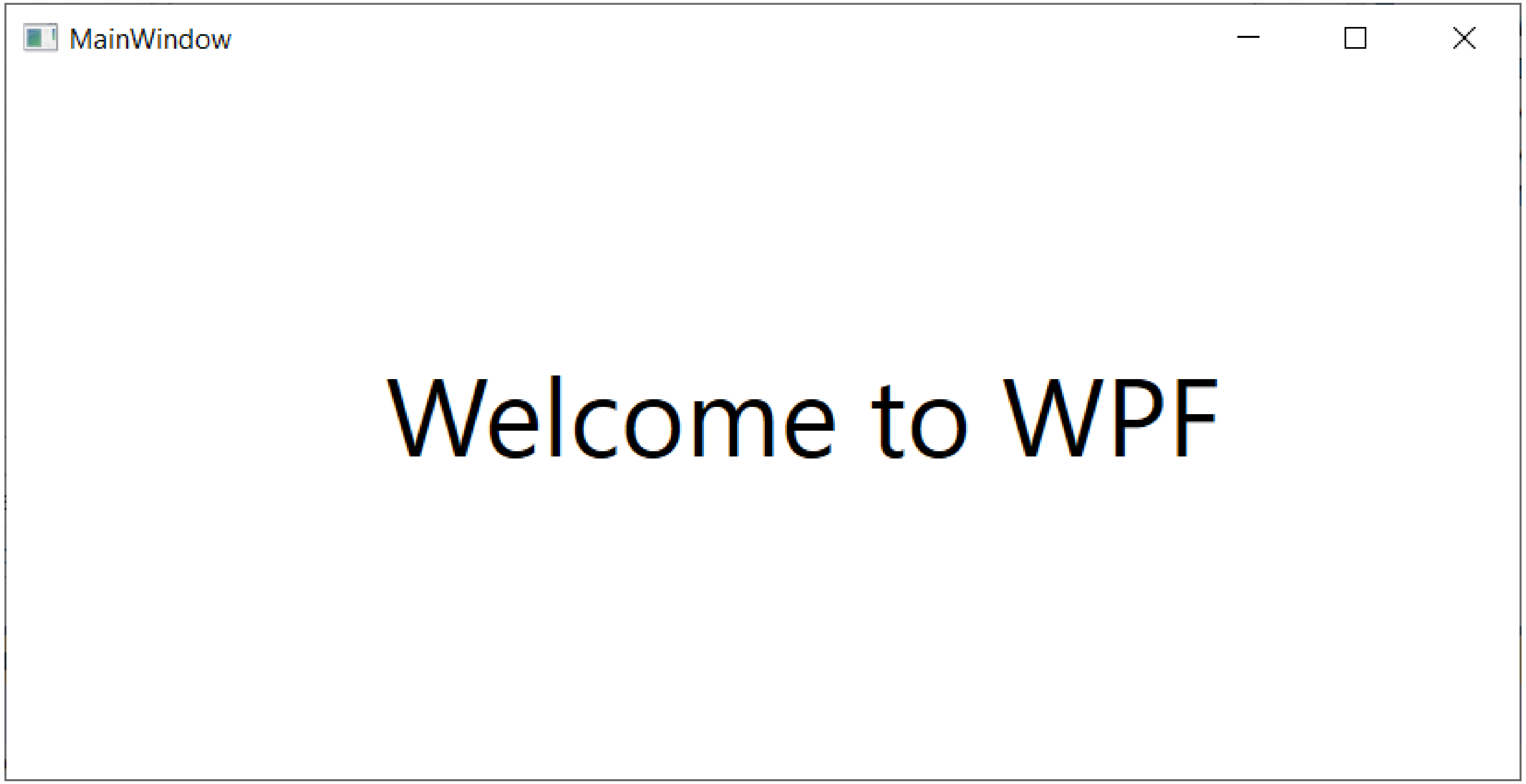
.NET 5.0 (Current)

4. Update codes of the MainWindow.xaml



```
    mc:Ignorable="d"
    Title="MainWindow" Height="344" Width="670">
<Grid Margin="0,0,0,-6">
    <TextBlock x:Name="textBlock" HorizontalAlignment="Left" Height="67"
               TextAlignment="Center" Text="Welcome to WPF"
               TextWrapping="Wrap" VerticalAlignment="Top" Width="387"
               Margin="152,115,0,0" FontSize="48" >
    </TextBlock>
</Grid>
```

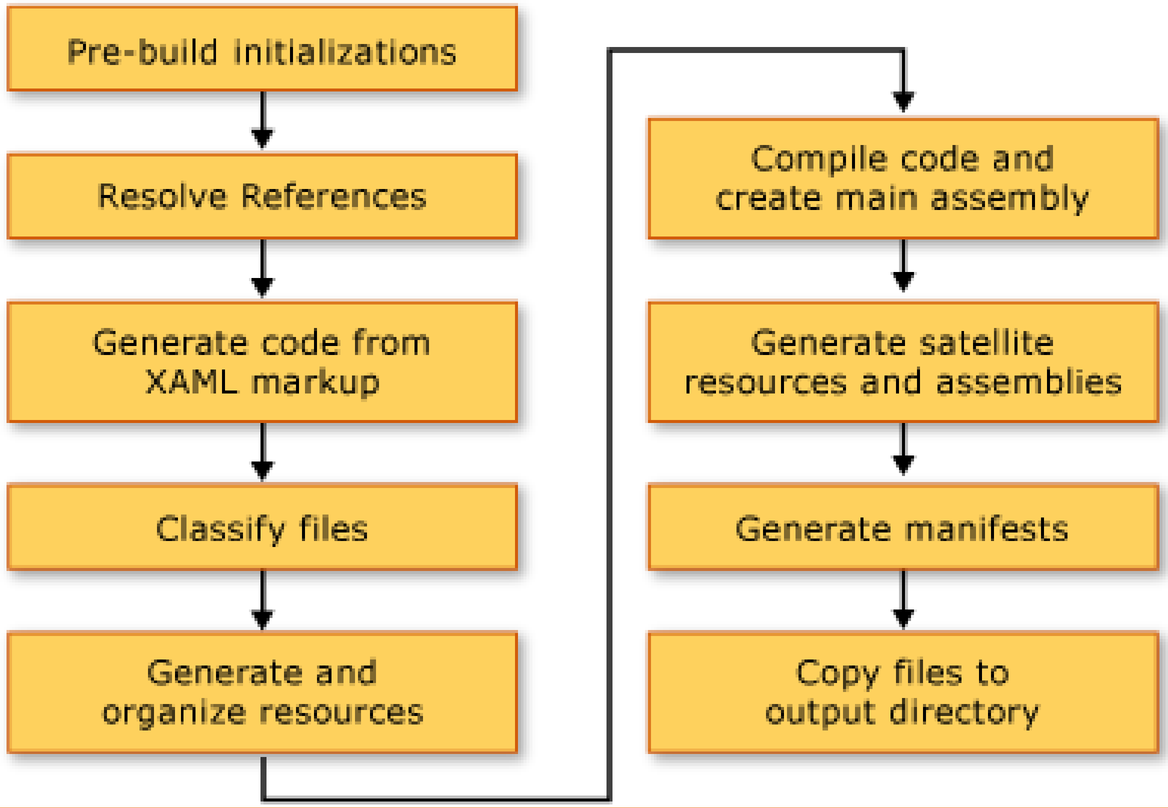
5. Press Ctrl+F5 to run application


MainWindow

Welcome to WPF

WPF Build Pipeline

- When a WPF project is built, the combination of language-specific and WPF-specific targets are invoked. The process of executing these targets is called the build pipeline, and the key steps are illustrated by the following figure:


Pre-build initializations
Resolve References
Generate code from XAML markup
Classify files
Generate and organize resources
Compile code and create main assembly
Generate satellite resources and assemblies
Generate manifests
Copy files to output directory

eXtensible Application Markup Language (XAML)

Understanding XAML

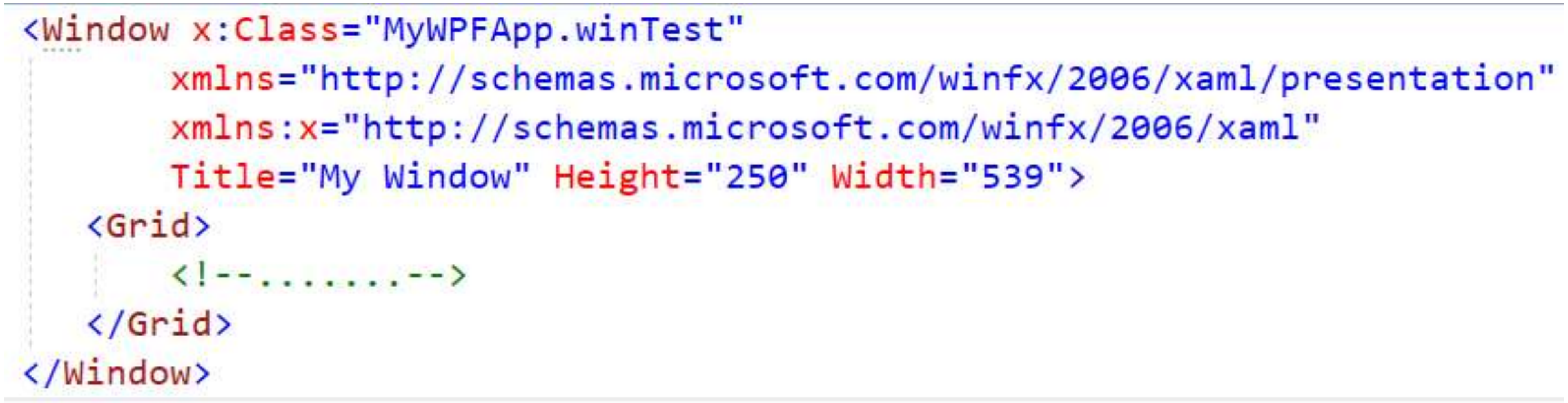
- XAML is a declarative markup language. As applied to the .NET Core programming model, XAML simplifies creating a UI for a .NET Core app
- We can create visible UI elements in the declarative XAML markup, and then separate the UI definition from the run-time logic by using code-behind files that are joined to the markup through partial class definitions
- XAML directly represents the instantiation of objects in a specific set of backing types defined in assemblies
- XAML enables a workflow where separate parties can work on the UI and the logic of an app, using potentially different tools
- When represented as text, XAML files are XML files that generally have the **.xaml** extension. The files can be encoded by any XML encoding, but encoding as UTF-8 is typical

The Features of XAML

- **UI and Business Logic Separation:** This is one of the greatest benefits of XAML. It separates design and development from each other. This provides more collaboration and efficiency between developers and designers of an application
- **High User Experience:** XAML files are basically simple XML format files, so transferring user interfaces between platforms is easy. To design user interfaces using XAML is easier and also needs lesser code
- **Easier Extension:** In XAML, .NET classes are placed in a hierarchical manner, where each element is the equivalent of a Core Common Language Runtime (Core CLR) class. So, extension of the .NET classes will be easier
- **Easier to implement Styles for UI:** XAML makes the development of any user interface much faster and easier. It provides features such as creating layout, applying styles, and templates for the UI application

Basic Structure of XAML

- A WPF application contains windows or pages. A Window is a top-level window with the tag, whereas the Page is a browser-hosted page with the <Page> tag in a XAML file
- Apart from Window and Page, XAML has ResourceDictionary and Application root elements for specifying the external dictionary and application definition



```
<Window x:Class="MyWPFApp.winTest"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="My Window" Height="250" Width="539">
    <Grid>
        <!--........-->
    </Grid>
</Window>
```

Basic Structure of XAML

- **Window**: One of commonly used root element which contains other elements
- **xmlns**: Namespace declared specifically for WPF
- **xmlns:x**: Namespace with keywords and markup extensions in XAML. It includes mapping with the x: prefix
- Some commonly used prefixes are as follows:
  - **x:Type** : To specify the type
  - **x:Null**: To assign a null value
  - **x:Class**: Specifies the related code-behind file. Here, MyWPFApp is the name of an application and MyWin is the name of the class that binds the XAML file with the related codebehind file
  - **Title**: The title of the window
  - **Grid**: Displays tabular data in a row and column format

```
<Window x:Class="MyWPFApp.MyWin"
        xmlns=“…"
        xmlns:x=“…"
        Title="My Window" >
    <Grid>
        <!--........-->
    </Grid>
 </Window>
```

Attributes in XAML

- The attribute element assigns the names for an event or value to the property
- The attribute is mentioned using attribute name, an assignment operator, and the value of the attribute in quotation marks (“”)
- The attributes are of two types
  - Property Attribute which defines properties for the element
  - Event Attribute which specifies handler for the element

Syntax	<...attribute_name="value"...>

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

Elements in XAML

- An XAML element instantiates a Core Common Language Runtime (Core CLR) class. The syntax of declaring elements is the same as element syntax of markup languages such as HTML, included two types:
  - Property element: enables to assign other element as a value of a property
  - Event element: handles an event of the control

Syntax – Property Element	<typeName.propertyName>...</typeName.propertyName >

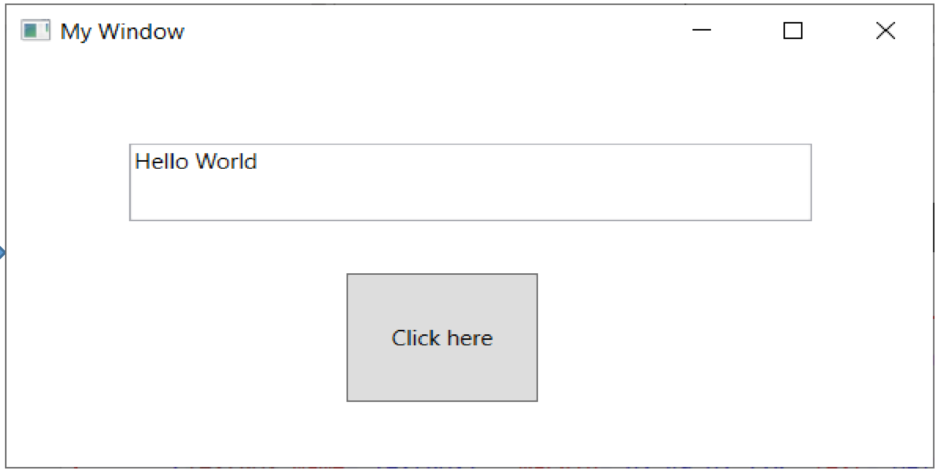
```
<Button>
   <Button.Foreground> <SolidColorBrush Color="Red"/> </Button.Foreground>
   <Button.Content> This is a button </Button.Content>
</Button>
```

Syntax – Event Element	<typeName event="event handler">...</ typeName>

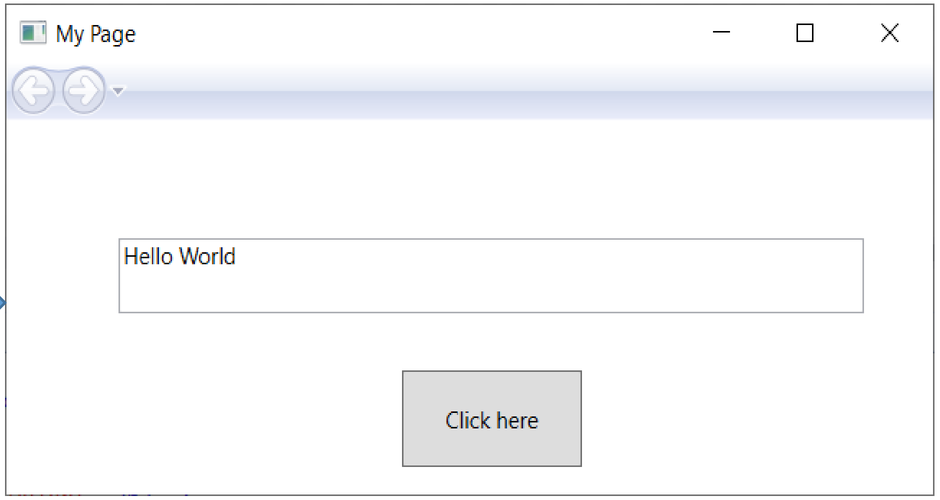
```
<Page xmlns="…" xmlns:x="…" x:Class="ExampleNamespace.ExamplePage">
       <Button Click="Button_Click" >Click Me!</Button>
</Page>
```

Defining the Window and Page

```
<Window x:Class="MyWPFApp.MyWin"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="My Window" Height="242" Width="476">
    <Grid>
        <TextBox Name="TextBox1"  Margin="62,49,62,118" Text="Hello World" />
        <Button Name="Button1"  Margin="173,122,201,57" >
            Click here
        </Button>
    </Grid>
</Window>
```



```
<Page
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     x:Class="MyWPFApp.MyPage"
     WindowTitle="My Page" Width="539" Height="263" >
    <Grid>
        <TextBox Name="TextBox1" Margin="62,62,62,162" Text="Hello World" />
        <Button Name="Button1" Width="100" Height="50" Margin="220,131,219,82" >
            Click here
        </Button>
    </Grid>
</Page>
```

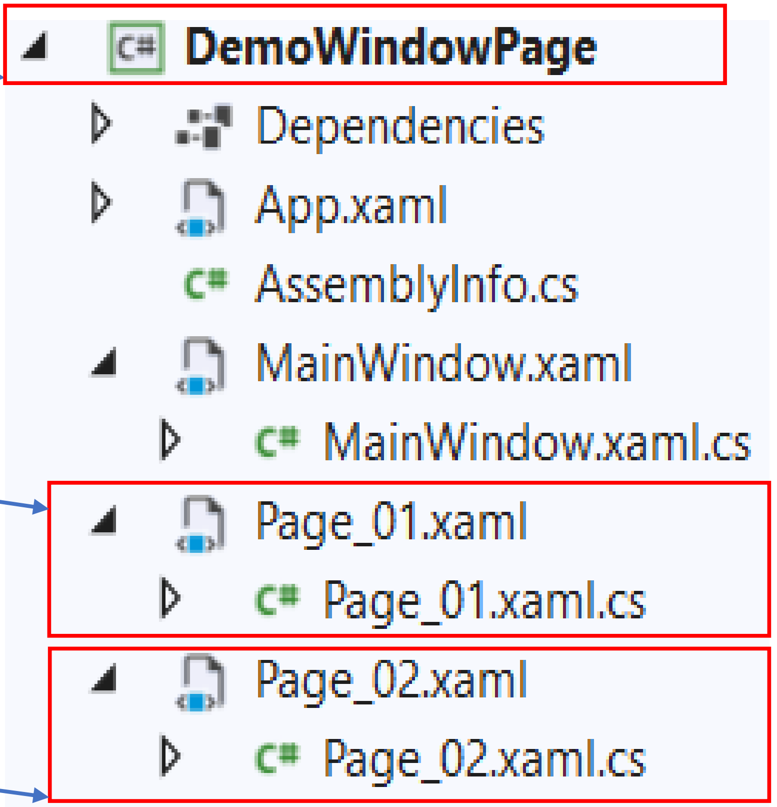


Demo Create Window and Page

1.Create a WPF project named **DemoWindowPage**

2.Right-click on project, select Add | Page (WPF)… and named **Page_01.xaml**

3.Right-click on project, select Add | Page (WPF)… and named **Page_02.xaml**



4.Open the **Page_01.xaml** and write codes as the follows:

```
<Page x:Class="DemoWindowPage.Page_01"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:local="clr-namespace:Demo01_Window_Page"
      mc:Ignorable="d" Width="500" Height="323"
      Title="Page 01: Welcome to WPF" >

    <Grid Background="LightGreen">
        <TextBlock TextAlignment="Center"  Width="362"
                   FontSize="24" FontWeight="Bold" Foreground="Red"
                   Text="Welcome to WPF"
                   Margin="69,130,69,151"/>
    </Grid>
</Page>
```

5.Open the **Page_02.xaml** and write codes as the follows:

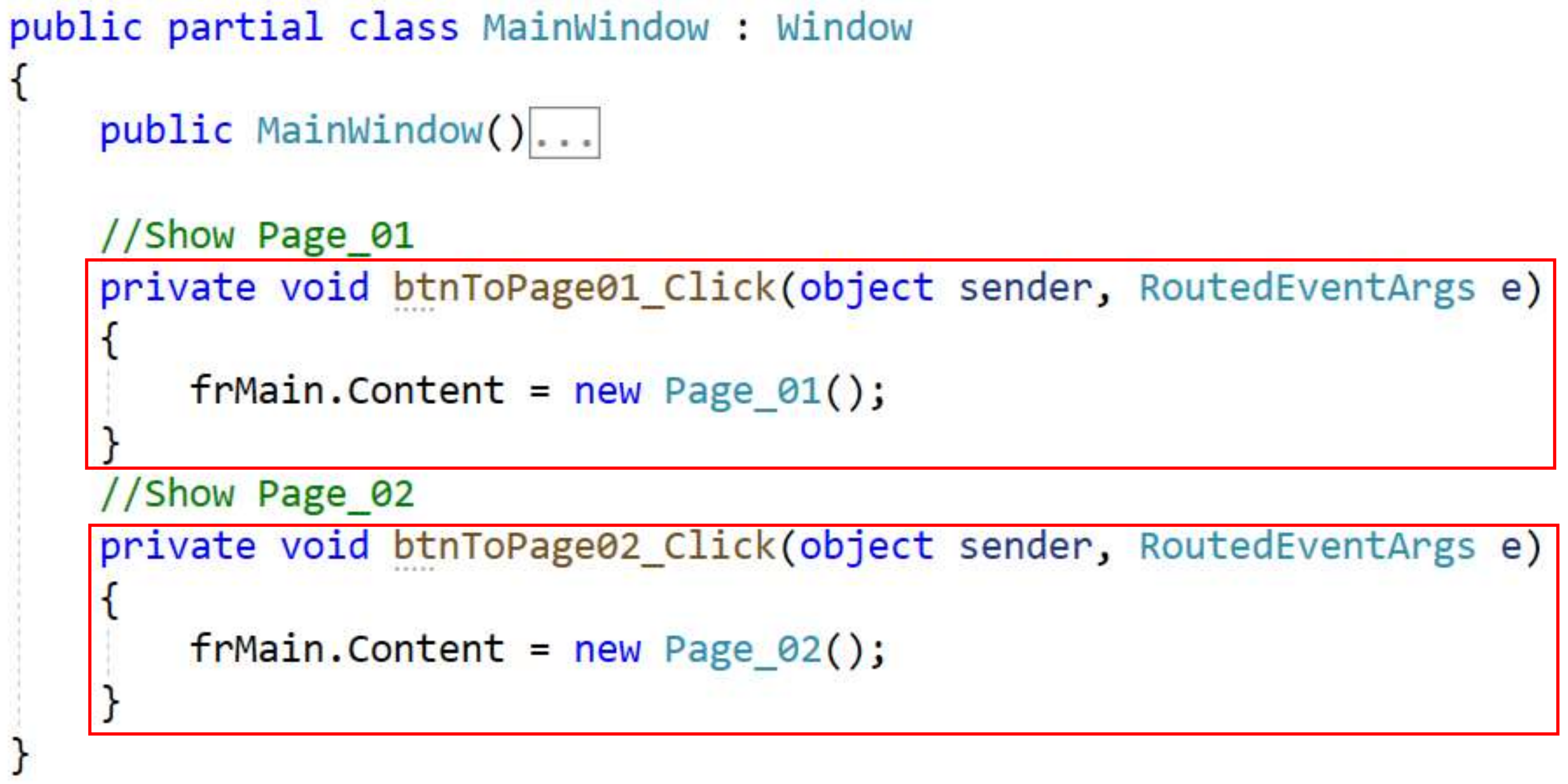
```
<Page x:Class="DemoWindowPage.Page_02"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:local="clr-namespace:Demo01_Window_Page"
      mc:Ignorable="d" Height="323" Width="500"
      Title="Page 02: .NET Programming">

   <Grid Background="PaleTurquoise">
        <TextBlock TextAlignment="Center" Width="362"
                   FontSize="24" FontWeight="Bold"
                   Text=".NET Programming" Foreground="ForestGreen"
                   Margin="69,130,69,151"/>
    </Grid>
</Page>
```

6.Open the **MainWindow.xaml** and write codes as the follows:

```
<Window x:Class="DemoWindowPage.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:Demo01_Window_Page"
        mc:Ignorable="d" Title="Main Window" Height="456" Width="800">
    <Grid>
        <Frame x:Name="frMain" VerticalAlignment="Stretch" NavigationUIVisibility="Visible"/>
        <Button Content="To page 1" Name="btnToPage01" Click="btnToPage01_Click"
                HorizontalAlignment="Left" Margin="316,383,0,0"
                Height="28" Width="79"/>
        <Button Content="To page 2" Name="btnToPage02" Click="btnToPage02_Click"
                HorizontalAlignment="Left" Margin="400,383,0,0"
                Height="28" Width="79"/>
    </Grid>
</Window>
```

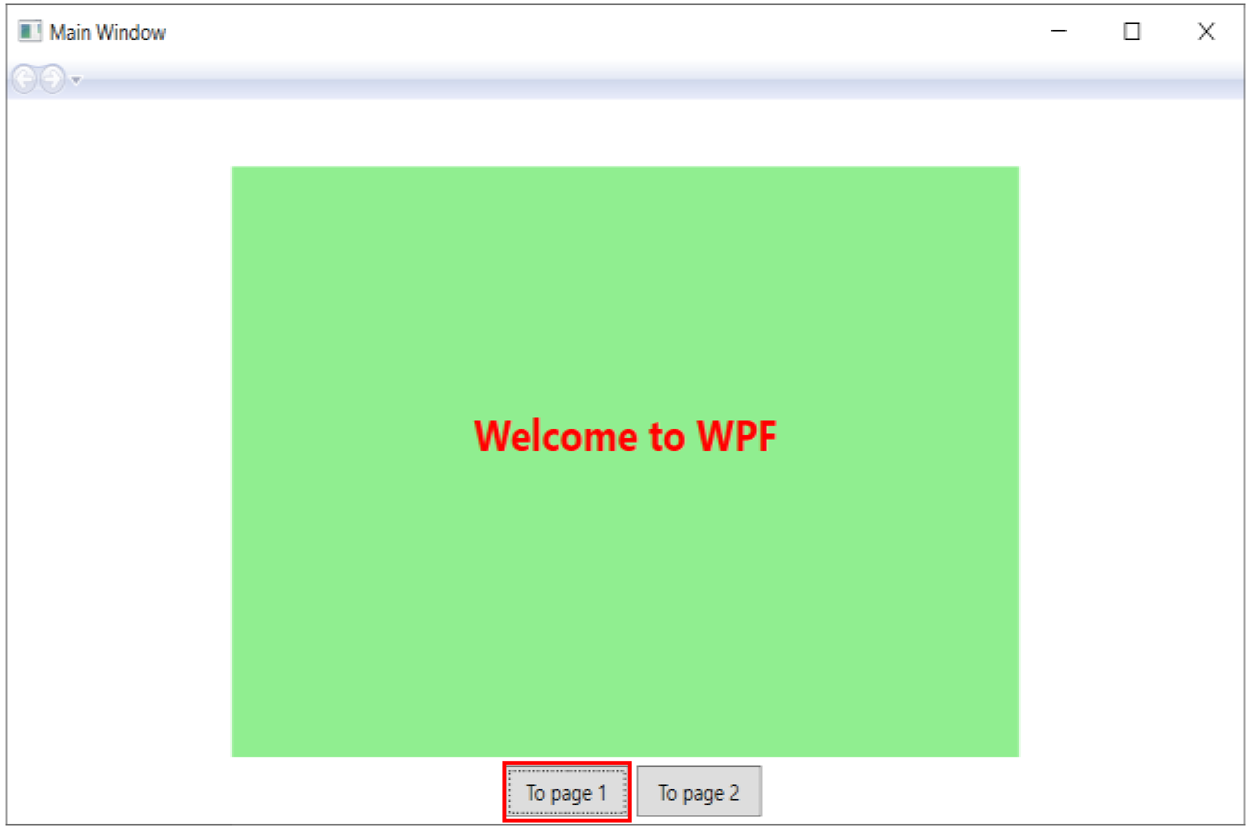
Event Handler: ***Click***

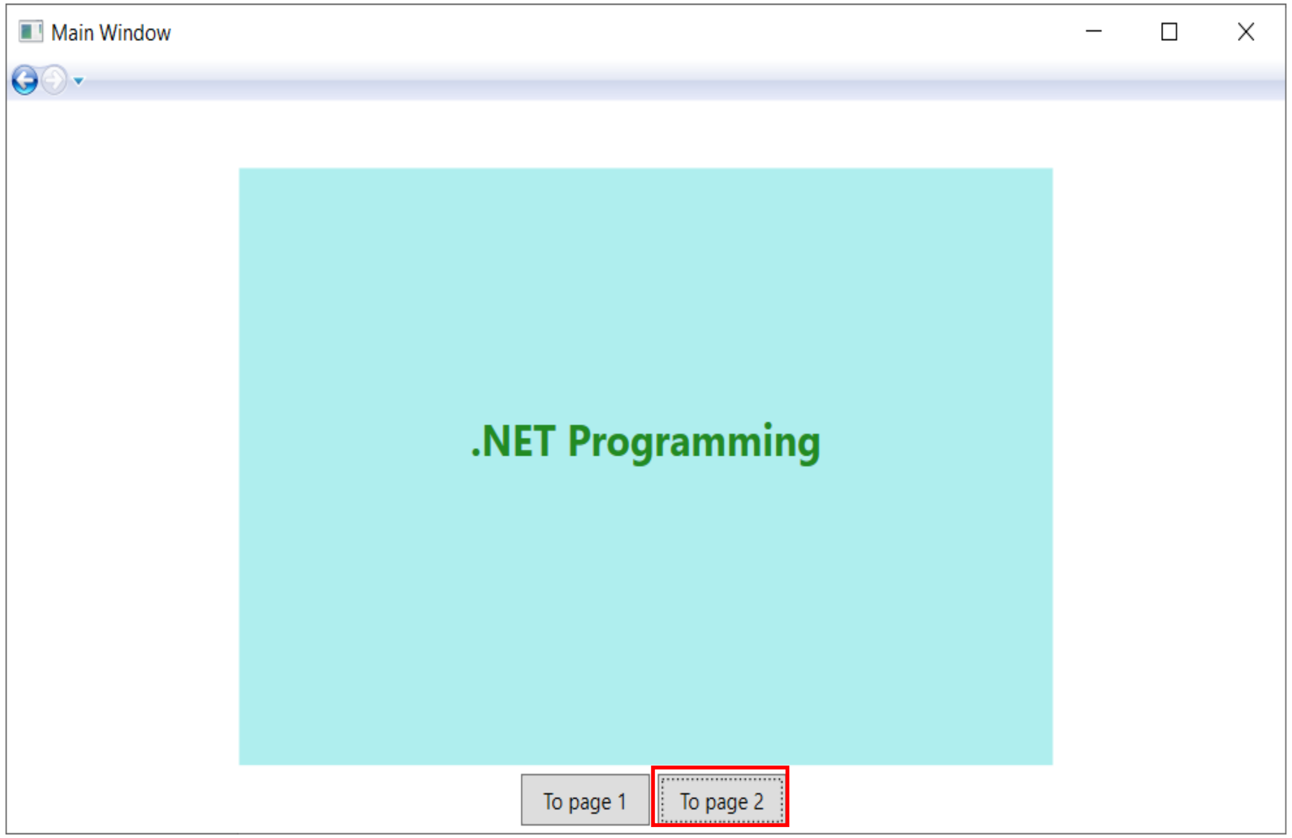
7.Open the **MainWindow.xaml.cs** and write codes as the follows:



```
public partial class MainWindow : Window
{
    public MainWindow()...

    //Show Page_01
    private void btnToPage01_Click(object sender, RoutedEventArgs e)
    {
        frMain.Content = new Page_01();
    }
    //Show Page_02
    private void btnToPage02_Click(object sender, RoutedEventArgs e)
    {
        frMain.Content = new Page_02();
    }
}
```

8.Press Ctrl+F5 to run project then click **To page 1** or **To page 2** button to view result





The Window Class

- The System.Windows.Window class (located in the PresentationFramework.dll assembly) represents a single window owned by the Application-derived class, including any dialog boxes displayed by the main window

- It serves as the root of a window and provides us with the standard border, title bar and maximize, minimize and close buttons

- A WPF window is a combination of a XAML (.xaml) file, where the <Window> element is the root, and a CodeBehind (.cs) file

The Window Class

```
<Window
xmlns="http://schemas.microsoft.com/winfx
/2006/xaml/presentation"
Title="Main   Window   in   Markup   Only"
Height="300"
Width="300"
/>
```

```
using System;
using System.Windows;
namespace CSharp {
 public partial class CodeOnlyWindow : Window {
  public CodeOnlyWindow() {
    this.Title = "Main Window in Code Only";
    this.Width = 300; this.Height = 300;
  }
 }
}
```

The Window Class

- The following table describes some of the key properties:

Property	Description
DataContext	Gets or sets the data context for an element when it participates in data binding
DialogResult	Gets or sets the dialog result value, which is the value that is returned from the ShowDialog() method
SizeToContent	Decide if the Window should resize itself to automatically fit its content. The default is Manual, which means that the window doesn't automatically resize
Topmost	The default is false, but if set to true, our Window will stay on top of other windows unless minimized. Only useful for special situations
Opacity	Gets or sets the opacity factor applied to the entire UIElement when it is rendered in the user interface (UI). This is a dependency property
WindowStartupLocation	Gets or sets the position of the window when first shown
WindowState	Gets or sets a value that indicates whether a window is restored, minimized, or maximized
WindowStyle	Gets or sets a window's border style

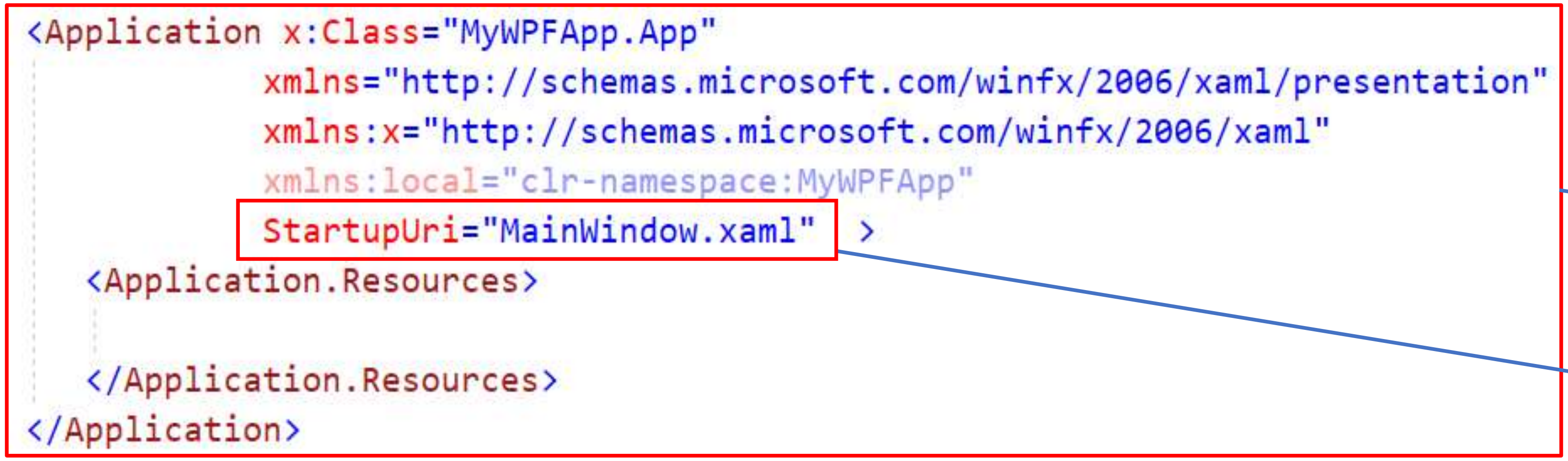
The Window Class

- The following table describes some of the key methods:

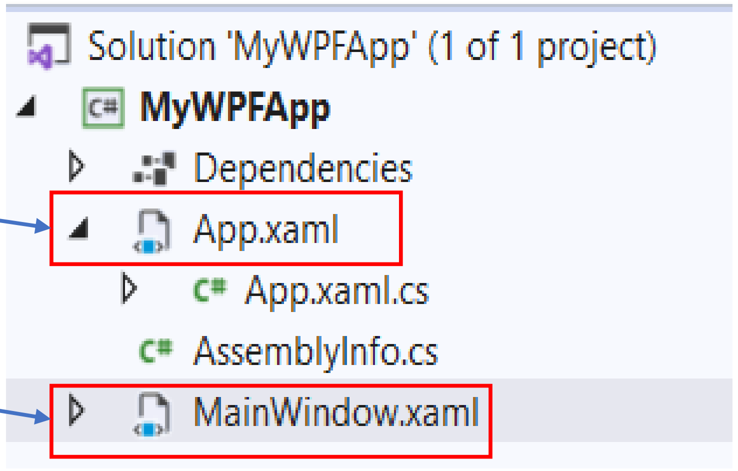
Method	Description
Activate()	Attempts to bring the window to the foreground and activates it
AddText(String)	Adds a specified text string to a ContentControl
Close()	Manually closes a Window
Hide()	Makes a window invisible
Show()	Opens a window and returns without waiting for the newly opened window to close
ShowDialog()	Opens a window and returns only when the newly opened window is closed
Show()	Opens a window and returns without waiting for the newly opened window to close
UpdateLayout()	Ensures that all visual child elements of this element are properly updated for layout
TransformToDescendant(Visual)	Returns a transform that can be used to transform coordinates from the Visual to the specified visual object descendant
BeginStoryboard(Storyboard)	Begins the sequence of actions that are contained in the provided storyboard

Defining the Application

- Visual Studio generates a XAML application file that specifies:
  - The code-behind class for the application
  - The startup window or page
  - Application-wide resources



```
<Application x:Class="MyWPFApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:MyWPFApp"
             StartupUri="MainWindow.xaml" >
    <Application.Resources>

    </Application.Resources>
</Application>
```

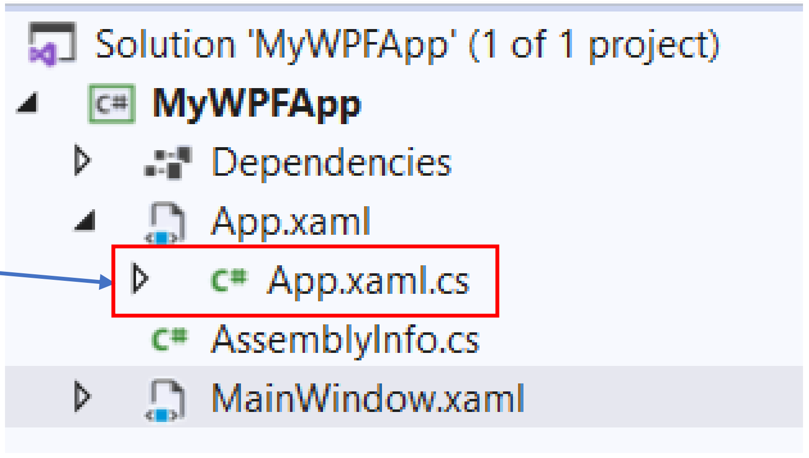


```
Solution 'MyWPFApp' (1 of 1 project)
◢ MyWPFApp
   ▷ Dependencies
   ◢ App.xaml
      ▷ App.xaml.cs
      AssemblyInfo.cs
   ▷ MainWindow.xaml
```

The Application Class

- The System.Windows.Application class represents a global instance of a running WPF application
- This class supplies a series of events that we are able to handle in order to interact with the application's lifetime (such as OnStartup and OnExit)

```
/// <summary>
/// Interaction logic for App.xaml
/// </summary>
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        //......
    }
    protected override void OnExit(ExitEventArgs e)
    {
        //.......
    }
}
```


Solution 'MyWPFApp' (1 of 1 project)
MyWPFApp
Dependencies
App.xaml
App.xaml.cs
AssemblyInfo.cs
MainWindow.xaml

The Application Class

- The following table describes some of the key properties:

Property	Description
Current	This static property allows us to gain access to the running Application object from anywhere in our code. This can be helpful when a window or dialog box needs to gain access to the Application object that created it, typically to access application-wide variables and functionality
MainWindow	This property allows us to programmatically get or set the main window of the application
Properties	This property allows us to establish and obtain data that is accessible throughout all aspects of a WPF application (windows, dialog boxes, etc.)
StartupUri	This property gets or sets a URI that specifies a window or page to open automatically when the application starts
Windows	This property returns a WindowCollection type, which provides access to each window created from the thread that created the Application object. This can be helpful when we want to iterate over each open window of an application and alter its state (such as minimizing all windows)

Controls and Layouts in WPF

System.Windows.Controls.Control

- Represents the base class for user interface (UI) elements that use a ControlTemplate to define their appearance
- The Template property, which is a ControlTemplate, specifies the appearance of the Control. If we want to change the appearance of a control but retain its functionality, we should consider creating a new ControlTemplate instead of creating a new class
- The Control class is the base class for many of the controls we add to an application such as TextBlock, Button, ListBox, etc

System.Windows.Controls.Control

- The following table describes some of the key members of the Control type:

Members	Description
Background, Foreground, BorderBrush, BorderThickness, Padding, HorizontalContentAlignment, VerticalContentAlignment	These properties allow us to set basic settings regarding how the control will be rendered and positioned
FontFamily, FontSize, FontStretch, FontWeight	These properties control various font-centric settings
IsTabStop, TabIndex	These properties are used to establish tab order among controls on a window
MouseDoubleClick, PreviewMouseDoubleClick	These events handle the act of double-clicking a widget
Template	This property allows us to get and set the control’s template, which can be used to change the rendering output of the widget

Styles and Templates

- WPF styling and templating refer to a suite of features that let developers and designers create visually compelling effects and a consistent appearance for their product

- When customizing the appearance of an app, we want a strong styling and templating model that enables maintenance and sharing of appearance within and among apps. WPF provides that model

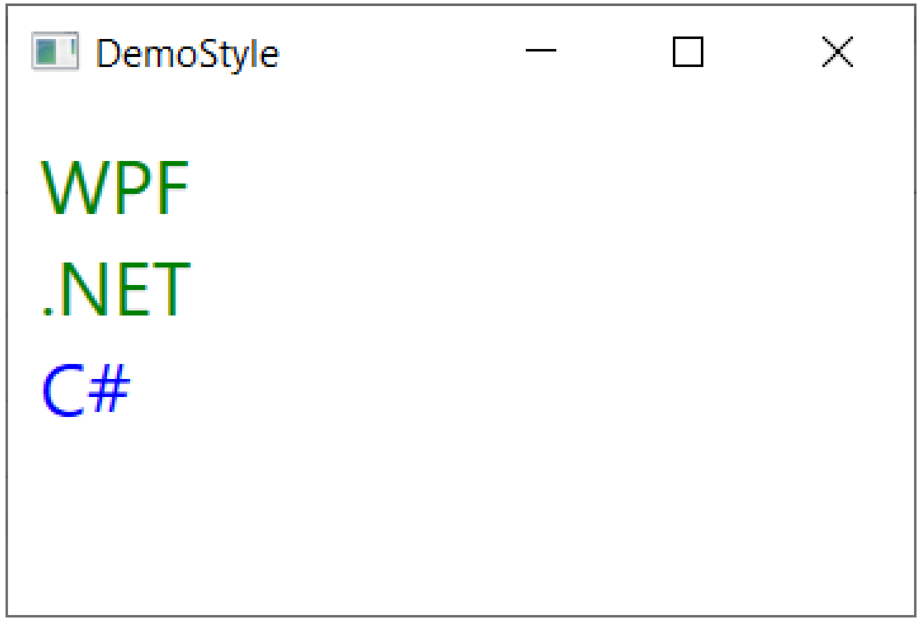
- Another feature of the WPF styling model is the separation of presentation and logic. Designers can work on the appearance of an app by using only XAML at the same time that developers work on the programming logic by using C# or Visual Basic

Styles

- A Style as a convenient way to apply a set of property values to multiple elements
- We can use a style on any element that derives from FrameworkElement or FrameworkContentElement such as a Window or a Button
- The most common way to declare a style is as a resource in the Resources section in a XAML file
- If we declare the style in the root element of app definition XAML file, the style can be used anywhere in app

Styles

```
<Window x:Class="DemoWPFControls.DemoStyle"
        <!--xmlns:-->
        Title="DemoStyle" Height="200" Width="300">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Green" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock>WPF</TextBlock>
        <TextBlock>.NET</TextBlock>
        <TextBlock Foreground="Blue">C#</TextBlock>
    </StackPanel>
</Window>
```



Templates

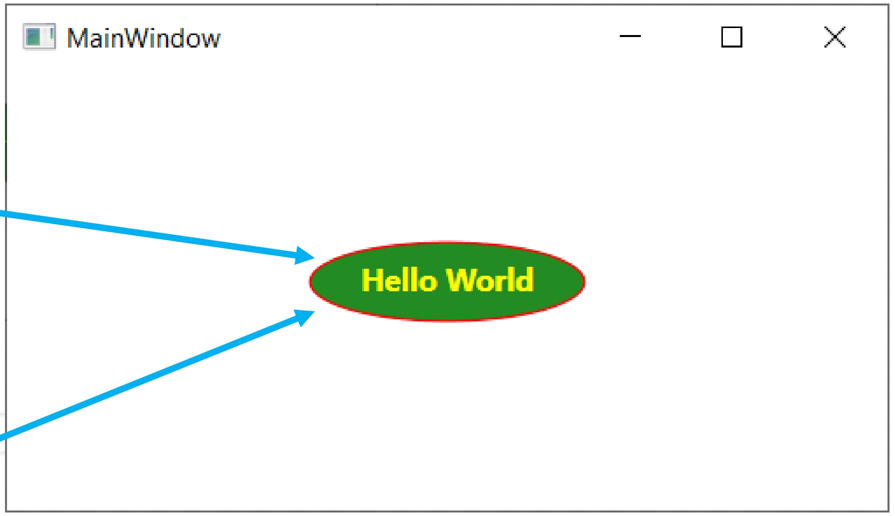
- A template describes the overall look and visual appearance of a control. For each control, there is a default template associated with it which gives the control its appearance
- In WPF applications, we can easily create templates when we want to customize the visual behavior and visual appearance of a control
- Connectivity between the logic and the template can be achieved by data binding. The main difference between styles and templates are listed below
  - Styles can only change the appearance of control with default properties of that control
  - With templates, we can access more parts of a control than in styles. we can also specify both existing and new behavior of a control

Templates

- There are two types of templates which are most commonly used: **Control Template** and **Data Template**
- **Control Template** defines the visual appearance of a control. All of the UI elements have some kind of appearance as well as behavior, e.g. Templates can be applied globally to application, windows and pages, or directly to controls. Most scenarios that require we to create a new control can be covered by instead creating a new template for an existing control
- **Data Template** defines and specifies the appearance and structure of a collection of data. It provides the flexibility to format and define the presentation of the data on any UI element. It is mostly used on data related Item controls such as ComboBox, ListBox, etc

Control Template with Button Demo

```xml
<Grid>
    <Button  Margin="228,127,227,75">
        <Button.Resources>
            <Style x:Key="TextStyle">
                <Setter Property="Control.Foreground" Value="Yellow" />
                <Setter Property="Control.FontFamily" Value="Tohoma"/>
                <Setter Property="Control.FontSize" Value="14px"/>
                <Setter Property="Control.FontWeight" Value="Bold"/>
            </Style>
            <Style TargetType="Button">
                <Setter Property="Template">
                    <Setter.Value>
                        <ControlTemplate TargetType="Button">
                            <Grid>
                                <Ellipse Fill="ForestGreen" StrokeThickness="1" Stroke="Red" />
                                <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"
                                  Content="Hello World" Style="{StaticResource TextStyle}"/>
                            </Grid>
                        </ControlTemplate>
                    </Setter.Value>
                </Setter>
            </Style>
        </Button.Resources>
    </Button>
</Grid>
```



Controlling Content Layout Using Panels

- A WPF application invariably contains a good number of UI elements (e.g., user input controls, graphical content, menu systems, and status bars) that need to be well organized within various windows.
- After place the UI elements, we need to make sure they behave as intended when the end user resizes the window or possibly a portion of the window (as in the case of a splitter window)
- To ensure our WPF controls retain their position within the hosting window, we can take advantage of a good number of panel types (also known as layout managers)
- By default, a new WPF Window created with Visual Studio will use a layout manager of type Grid

Controlling Content Layout Using Panels

- The System.Windows.Controls namespace provides numerous panels, each of which controls how sub elements are maintained. The following table documents the role of some commonly used WPF panel controls:

Panel Control	Description
Canvas	Provides a classic mode of content placement. Items stay exactly where we 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

Canvas Panel Demo

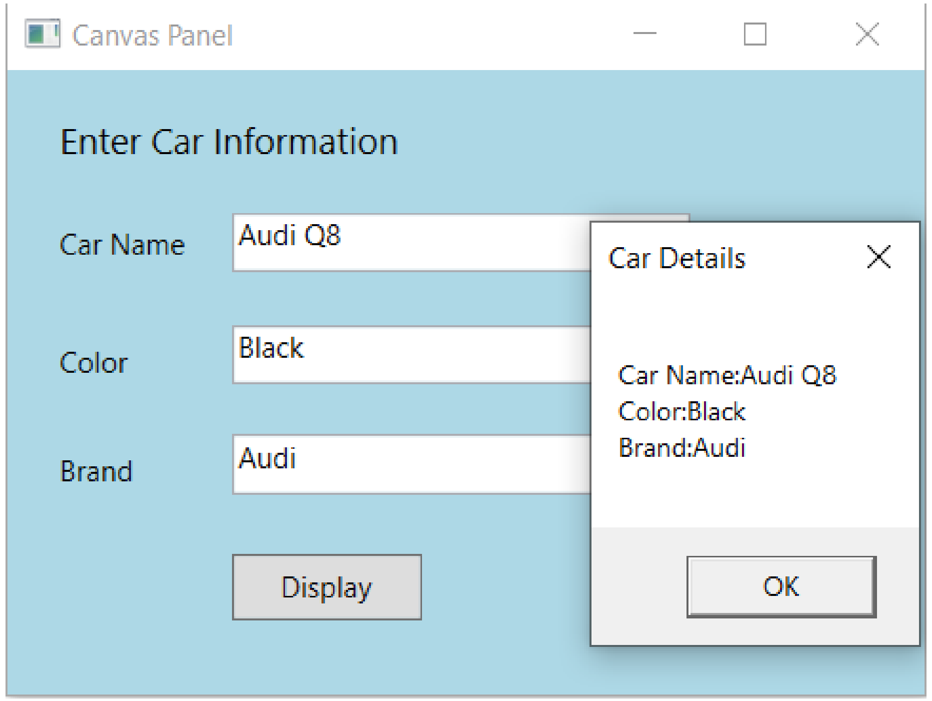
1. Create a DemoCanvasPanel.xaml and write codes as follows:

```
<Window x:Class="MyWPFApp.DemoCanvasPanel"
        <!-- xmlns=… -->
        Title="Canvas Panel" Height="300" Width="400" WindowStartupLocation="CenterScreen" >
  <Grid>
    <Canvas Background="LightBlue">
            <Button x:Name="btnDisplay" Canvas.Left="94" Height="28" Canvas.Top="203"
                    Width="80" Content="Display" Click="btnDisplay_Click" />
            <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
                   Width="328" Height="27" FontSize="15" Content="Enter Car Information"/>
            <Label x:Name="lblCarName" Canvas.Left="17" Canvas.Top="60" Content="Car Name"/>
            <TextBox x:Name="txtCarName" 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="lblBrand" Canvas.Left="17" Canvas.Top="155" Content="Brand"/>
            <TextBox x:Name="txtBrand" Canvas.Left="94" Canvas.Top="153" Width="193" Height="25"/>
    </Canvas>
  </Grid>
</Window>
```

Canvas Panel Demo

2. Open DemoCanvasPanel.xaml.cs then write codes and run

```
public partial class DemoCanvasPanel : Window {
    public DemoCanvasPanel()...
    private void btnDisplay_Click(object sender, RoutedEventArgs e)
    {
        string CarInfo = $"Car Name:{txtCarName.Text}\n" +
            $"Color:{txtColor.Text} \nBrand:{txtBrand.Text}";
        MessageBox.Show(CarInfo, "Car Details");
    }
}
```



Audi Q8

Black

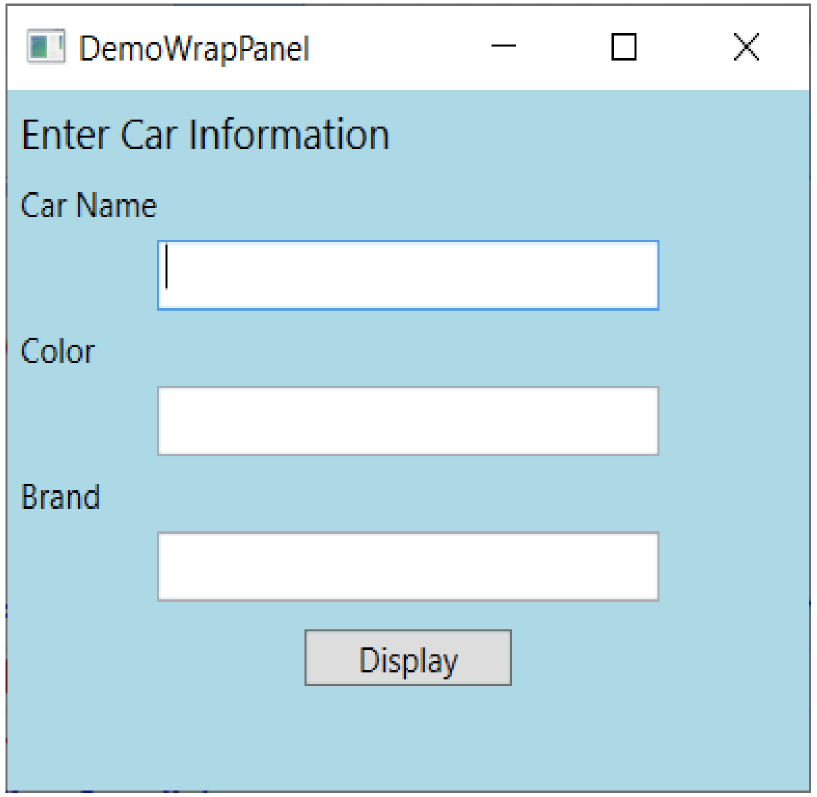
Audi

```
Car Name:Audi Q8
Color:Black
Brand:Audi
```

WrapPanel Demo

- Positions child elements in sequential position from left to right, breaking 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

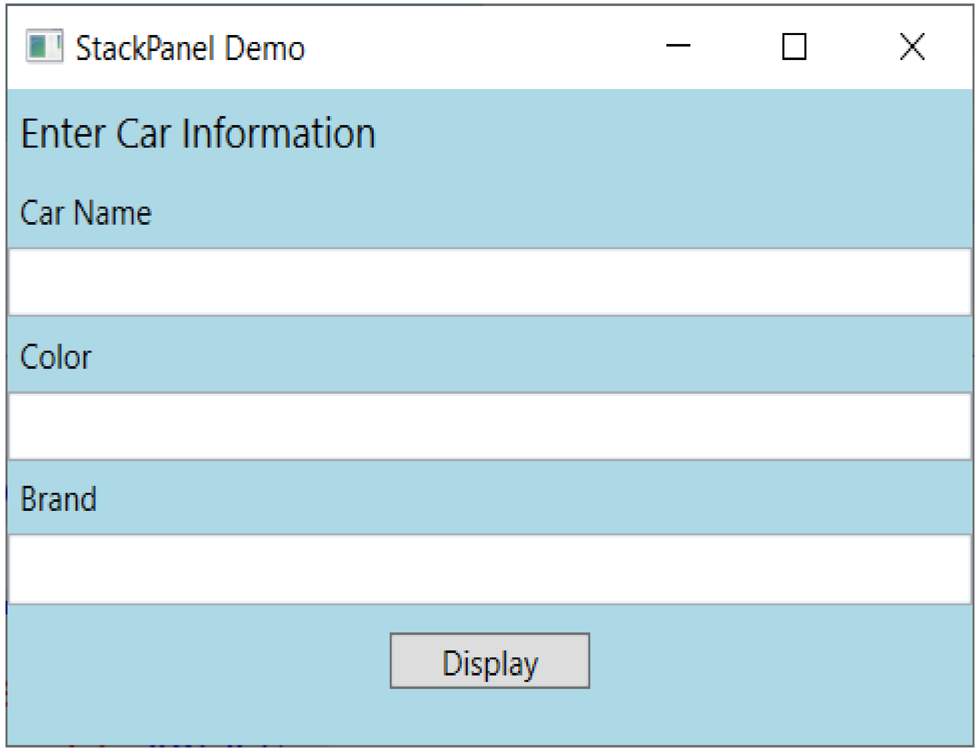
```
Grid>
    <WrapPanel Background="LightBlue"  Orientation ="Vertical">
       <Label Name="lblInstruction" Width="328" Height="27"
          FontSize="15" Content="Enter Car Information"/>
       <Label Name="lblCarName" Content="Car Name"/>
       <TextBox Name="txtCarName" Width="193" Height="25"/>
       <Label Name="lblColor" Content="Color"/>
       <TextBox Name="txtColor" Width="193" Height="25"/>
       <Label Name="lblBrand" Content="Brand"/>
       <TextBox Name="txtBrand" Width="193" Height="25"/>
       <Button Name="btnDisplay" Width="80"
          Margin="0,10,0,0" Content="Display"/>
    </WrapPanel>
</Grid><
```



StackPanel Demo

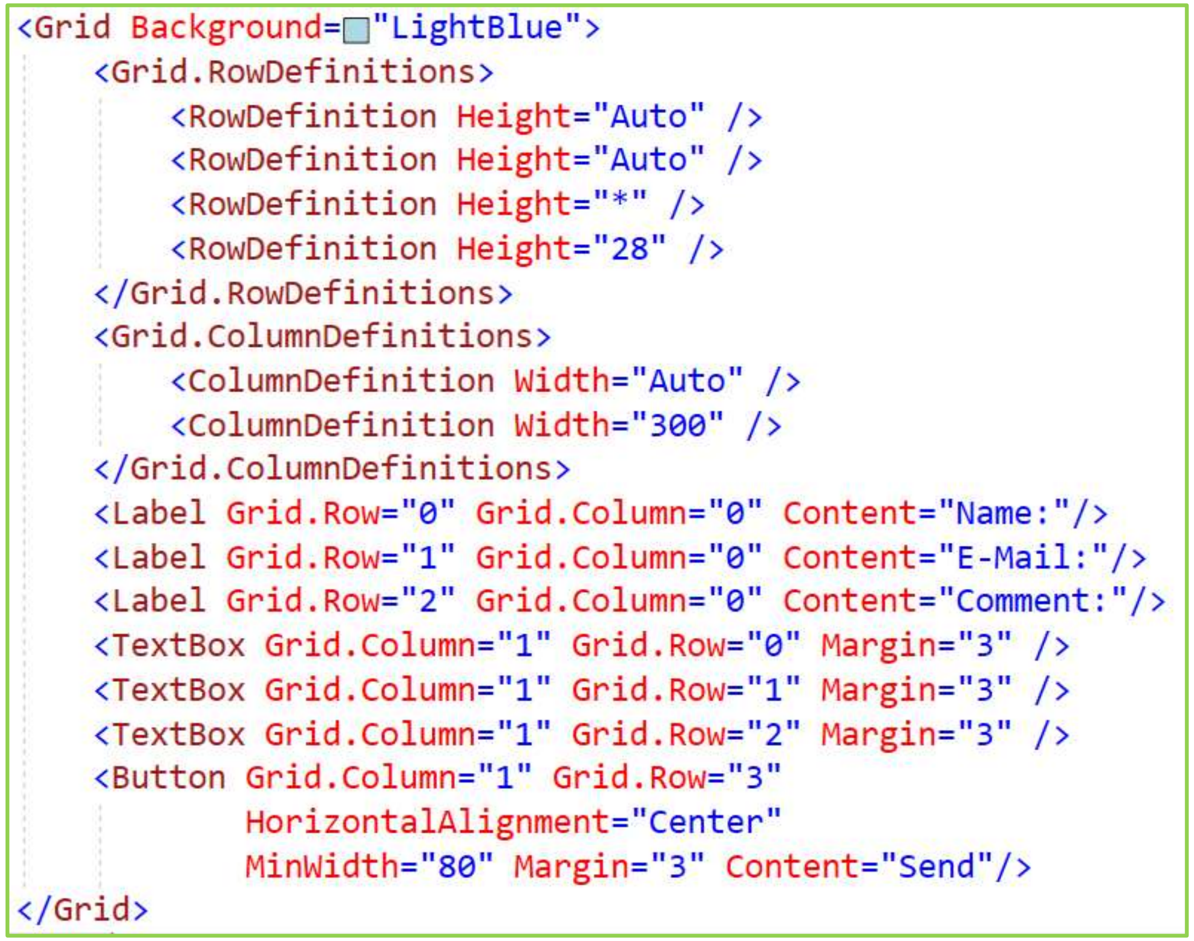
- Arranges child elements into a single line that can be oriented horizontally or vertically

```
<Grid>
  <StackPanel Background="LightBlue“ Orientation ="Vertical">
     <Label Name="lblInstruction"
             FontSize="15" Content="Enter Car Information"/>
     <Label Name="lblCarName" Content="Car Name"/>
     <TextBox Name="txtCarName"  Height="25"/>
     <Label Name="lblColor" Content="Color"/>
     <TextBox Name="txtColor"  Height="25"/>
     <Label Name="lblBrand" Content="Brand"/>
     <TextBox Name="txtBrand" Height="25"/>
     <Button Name="btnDisplay" Width="80"
             Margin="0,10,0,0" Content="Display"/>
  </StackPanel>
</Grid>
```

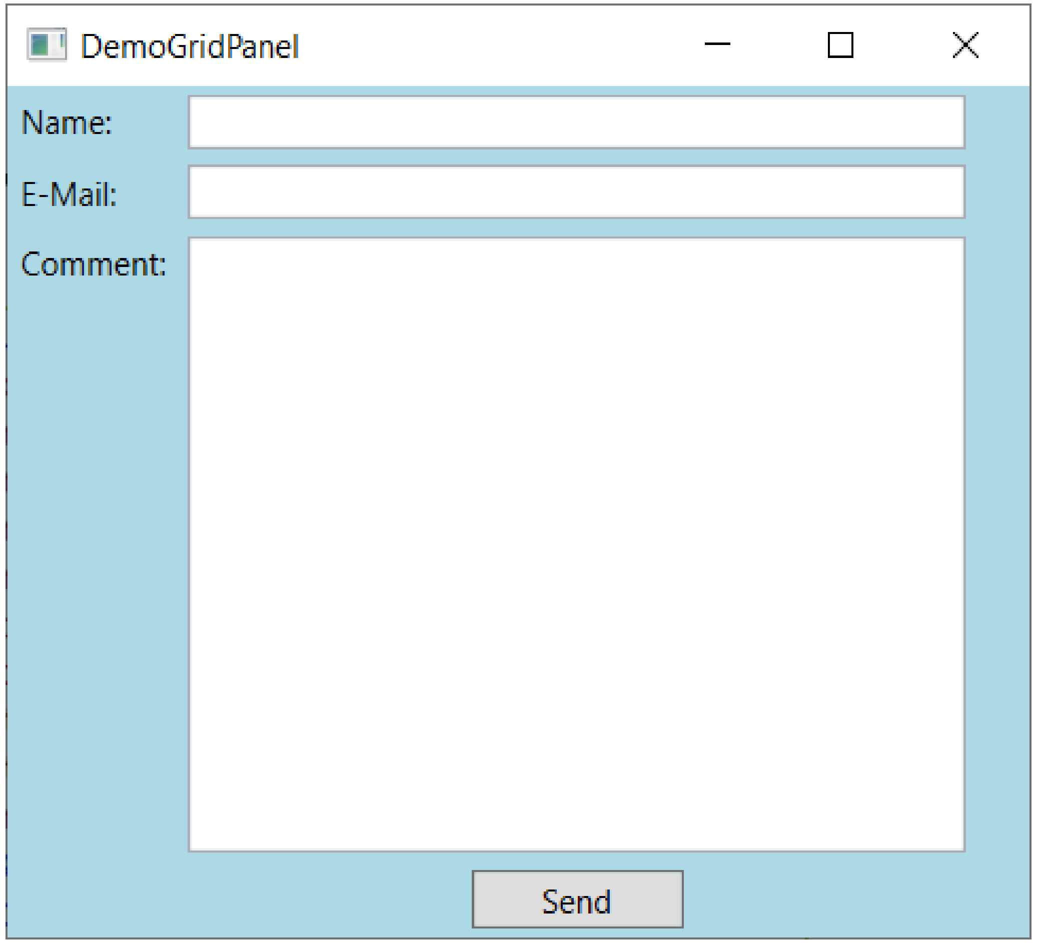


Grid Panel Demo

- Defines a flexible grid area that consists of columns and rows

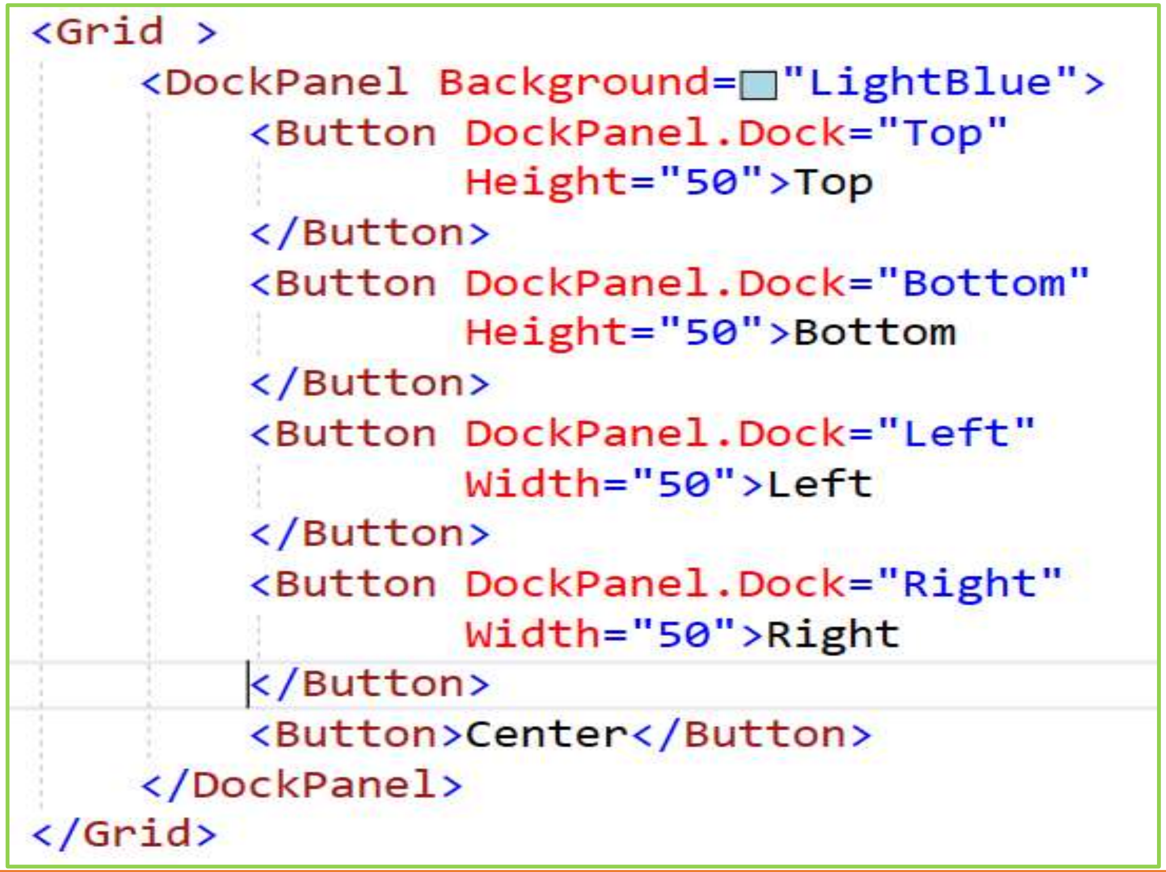


```
<Grid Background="LightBlue">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="28" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="300" />
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Name:"/>
    <Label Grid.Row="1" Grid.Column="0" Content="E-Mail:"/>
    <Label Grid.Row="2" Grid.Column="0" Content="Comment:"/>
    <TextBox Grid.Column="1" Grid.Row="0" Margin="3" />
    <TextBox Grid.Column="1" Grid.Row="1" Margin="3" />
    <TextBox Grid.Column="1" Grid.Row="2" Margin="3" />
    <Button Grid.Column="1" Grid.Row="3"
            HorizontalAlignment="Center"
            MinWidth="80" Margin="3" Content="Send"/>
</Grid>
```

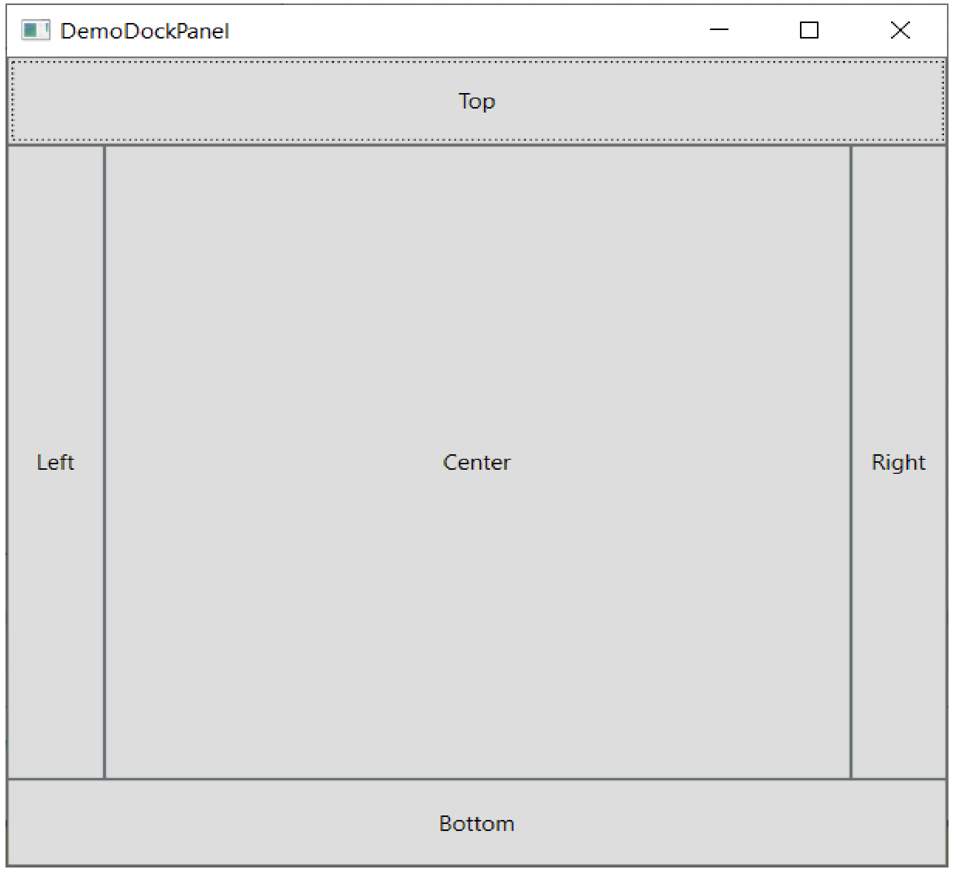


DockPanel Demo

- Defines an area where we can arrange child elements either horizontally or vertically, relative to each other



```
<Grid >
    <DockPanel Background="LightBlue">
        <Button DockPanel.Dock="Top"
                Height="50">Top
        </Button>
        <Button DockPanel.Dock="Bottom"
                Height="50">Bottom
        </Button>
        <Button DockPanel.Dock="Left"
                Width="50">Left
        </Button>
        <Button DockPanel.Dock="Right"
                Width="50">Right
        </Button>
        <Button>Center</Button>
    </DockPanel>
</Grid>
```



Controls in WPF

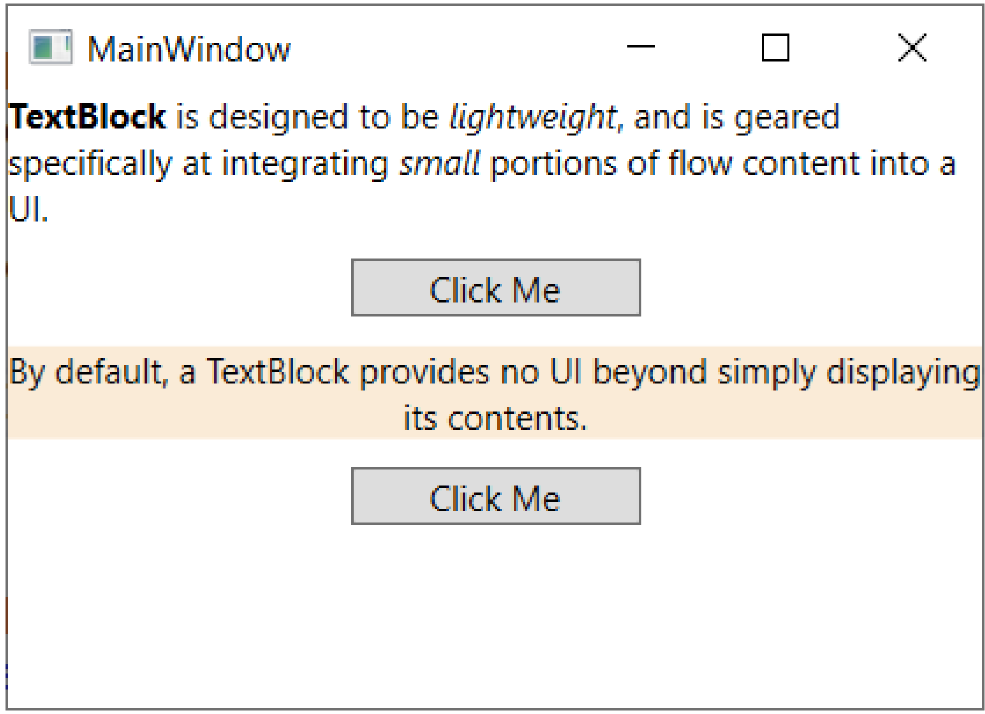
- WPF has a rich set of UI controls. These controls are grouped into various categories depending on their functionality, as shown in the following table:

Category	Controls
Layout	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, StackPanel, Viewbox, WrapPanel
Core user input controls	Button, Calendar, DatePicker, Expander, DataGrid, ToggleButton, ScrollBar, Slider,TextBlock, TextBox, RepeatButton, RichTextBox, Label, PasswordBox
Menus	ContextMenu, Menu, ToolBar
Selection	CheckBox, ComboBox, ListBox, ListView, TreeView, RadioButton
Navigation	Frame, Hyperlink, Page, NavigationWindow, TabControl
User Information	AccessText, Label, Popup, ProgressBar, StatusBar, TextBlock, ToolTip
Media	Image, MediaElement, SoundPlayerAction

TextBlock

- Provides a lightweight control for displaying small amounts of flow content

```
<Grid>
    <StackPanel>
      <TextBlock Name="textBlock1" TextWrapping="Wrap">
           <Bold>TextBlock</Bold> is designed to be
           <Italic>lightweight</Italic>,
            and is geared specifically at integrating
            <Italic>small</Italic>
                portions of flow content into a UI.
      </TextBlock>
      <Button Width="100" Margin="10">Click Me</Button>
      <TextBlock  Name="textBlock2" TextWrapping="Wrap"
            Background="AntiqueWhite" TextAlignment="Center">
            By default, a TextBlock provides no UI
            beyond simply displaying its contents.
      </TextBlock>
     <Button Width="100" Margin="10">Click Me</Button>
   </StackPanel>
</Grid>
```


MainWindow

TextBlock is designed to be *lightweight*, and is geared specifically at integrating *small* portions of flow content into a UI.

Click Me

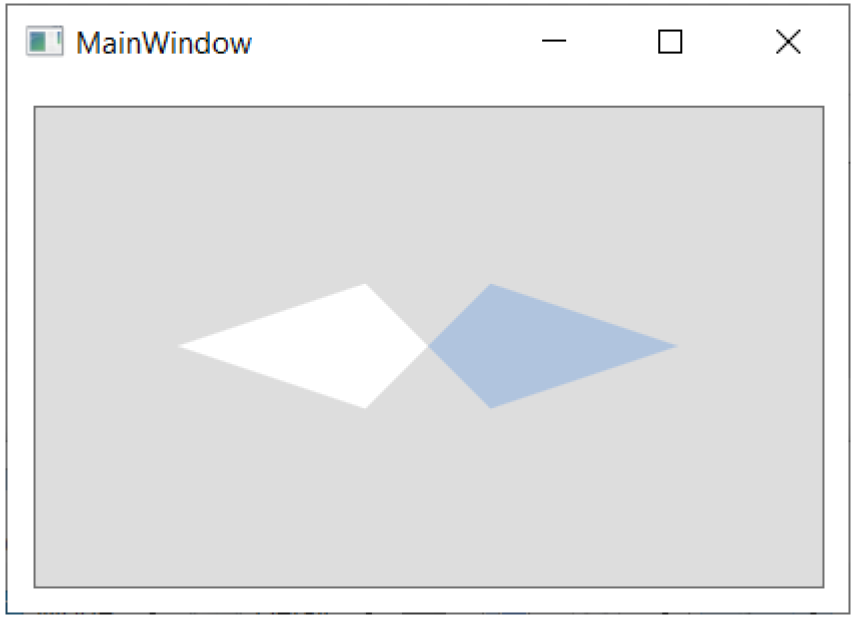
By default, a TextBlock provides no UI beyond simply displaying its contents.

Click Me

Button

- Represents a Windows button control, which reacts to the Click event

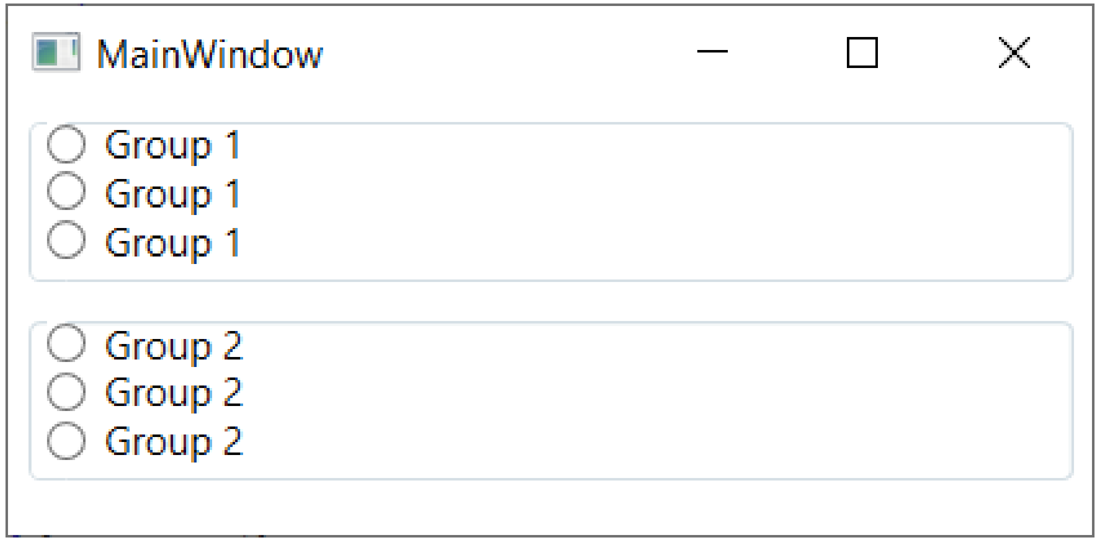
```
<Window x:Class="DemoWPFControls.MainWindow"
    //xmlns:…
    Title="MainWindow" Height="250" Width="350">
    <Grid>
        <Button Margin="10">
            <Grid>
                <Polygon Points="100,25 125,0 200,25 125,50"
                 Fill="LightSteelBlue" />
                <Polygon Points="100,25 75,0 0,25 75,50"
                 Fill="White"/>
            </Grid>
        </Button>
    </Grid>
</Window>
```



RadioButton

- Represents a button that can be selected, but not cleared, by a user

```
<Grid>
    <StackPanel>
        <GroupBox Margin="5">
            <StackPanel>
                <RadioButton>Group 1</RadioButton>
                <RadioButton>Group 1</RadioButton>
                <RadioButton>Group 1</RadioButton>
            </StackPanel>
        </GroupBox>
        <GroupBox Margin="5">
            <StackPanel>
                <RadioButton GroupName="Group2">Group 2
                </RadioButton>
                <RadioButton GroupName="Group2">Group 2
                </RadioButton>
                <RadioButton GroupName="Group2">Group 2
                </RadioButton>
            </StackPanel>
        </GroupBox>
    </StackPanel>
</Grid>
```

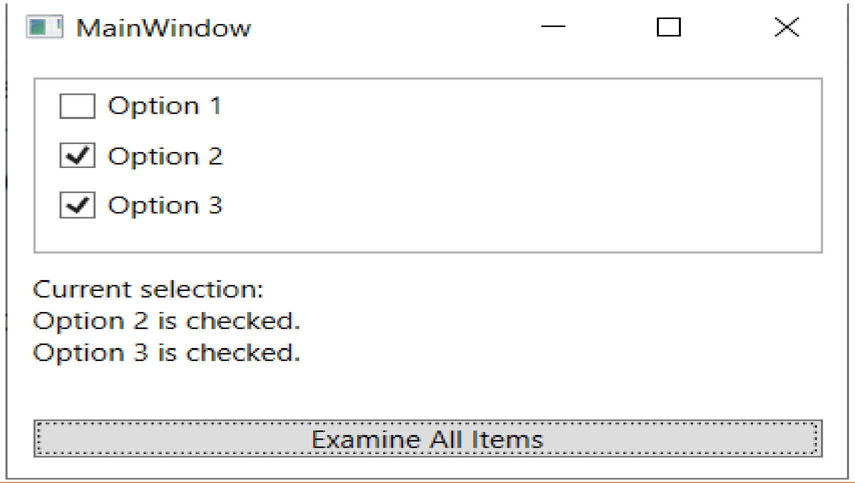


ListBox

- Contains a list of selectable items

```
<Grid Margin="10">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <ListBox Name="lst" >
        <CheckBox Margin="3">Option 1</CheckBox>
        <CheckBox Margin="3">Option 2</CheckBox>
        <CheckBox Margin="3">Option 3</CheckBox>
    </ListBox>
    <StackPanel Grid.Row="1" Margin="0,10,0,0">
        <TextBlock>Current selection:</TextBlock>
        <TextBlock  Name="txtSelection" TextWrapping="Wrap"></TextBlock>
        <Button Margin="0,10,0,0" Click="cmd_CheckAllItems">
            Examine All Items</Button>
    </StackPanel>
</Grid>
```

```
public partial class MainWindow : Window{
    public MainWindow()...
    private void cmd_CheckAllItems(object sender, RoutedEventArgs e){
        StringBuilder sb = new StringBuilder();
        foreach (CheckBox item in lst.Items){
            if (item.IsChecked == true) {
                sb.Append(
                    item.Content + " is checked.");
                sb.Append("\r\n");
            }
        }
        txtSelection.Text = sb.ToString();
    }
}
```


MainWindow

- [ ] Option 1
- [x] Option 2
- [x] Option 3

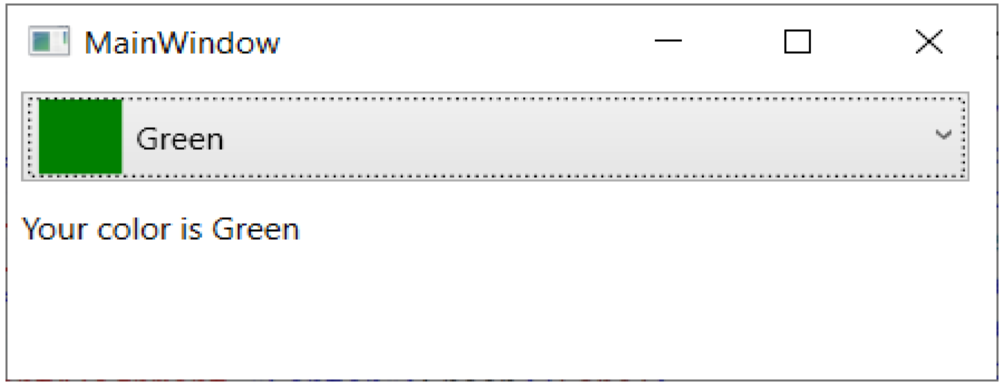
Current selection:
Option 2 is checked.
Option 3 is checked.

ComboBox

- Represents a selection control with a drop-down list that can be shown or hidden by clicking the arrow on the control

```
<StackPanel HorizontalAlignment="Left" Width="350">
    <ComboBox Margin="5"  Name="cboColor"
              SelectionChanged="cboColor_SelectionChanged" >
        <StackPanel Orientation="Horizontal">
            <Rectangle Fill=🟩"Green"  Width="30" Height="30"></Rectangle>
            <Label VerticalContentAlignment="Center">Green</Label>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <Rectangle Fill=🟥"Red"  Width="30" Height="30"></Rectangle>
            <Label VerticalContentAlignment="Center">Red</Label>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <Rectangle Fill=🟨"Yellow"  Width="30" Height="30"></Rectangle>
            <Label VerticalContentAlignment="Center">Yellow</Label>
        </StackPanel>
    </ComboBox>
    <Label Name="lbMsg"/>
```

```
public partial class MainWindow : Window{
    public MainWindow()...
    private void cboColor_SelectionChanged(object sender,
        SelectionChangedEventArgs e){
        var stackPanel = (StackPanel)cboColor.SelectedItem;
        var label = stackPanel.Children[1] as Label;
        string color = label.Content.ToString();
        lbMsg.Content = "Your color is "+color;
    }
}
```



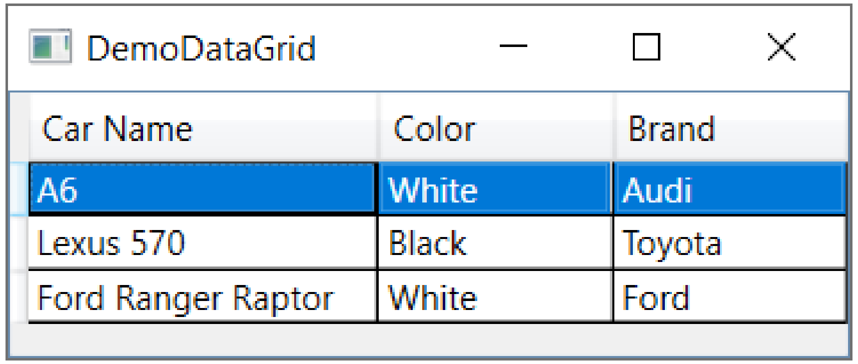
Your color is Green

DataGrid

- Represents a control that displays data in a customizable grid

```
<DataGrid Name="dgCarList" AutoGenerateColumns="False" >
    <DataGrid.Columns>
        <DataGridTextColumn Width="*" Header="Car Name"
                            Binding="{Binding CarName}"/>
        <DataGridTextColumn Width="*" Header="Color"
                            Binding="{Binding Color}" />
        <DataGridTextColumn Width="*" Header="Brand"
                            Binding="{Binding Brand}" />
    </DataGrid.Columns>
</DataGrid>
```

```
public partial class DemoDataGrid : Window{
    public DemoDataGrid() {
        InitializeComponent();
        //Event Handler: Loaded
        Loaded += DemoDataGrid_Loaded;
    }
    private void DemoDataGrid_Loaded(object sender, RoutedEventArgs e) {
        //Create a Car List
        List<dynamic> cars = new List<dynamic> {
            new {CarName = "A6", Color="White", Brand="Audi" },
            new {CarName = "Lexus 570", Color="Black", Brand="Toyota" },
            new {CarName = "Ford Ranger Raptor", Color="White", Brand="Ford" }
        };
        //Binding on DataGrid
        dgCarList.ItemsSource = cars;
    }
}
```


DemoDataGrid

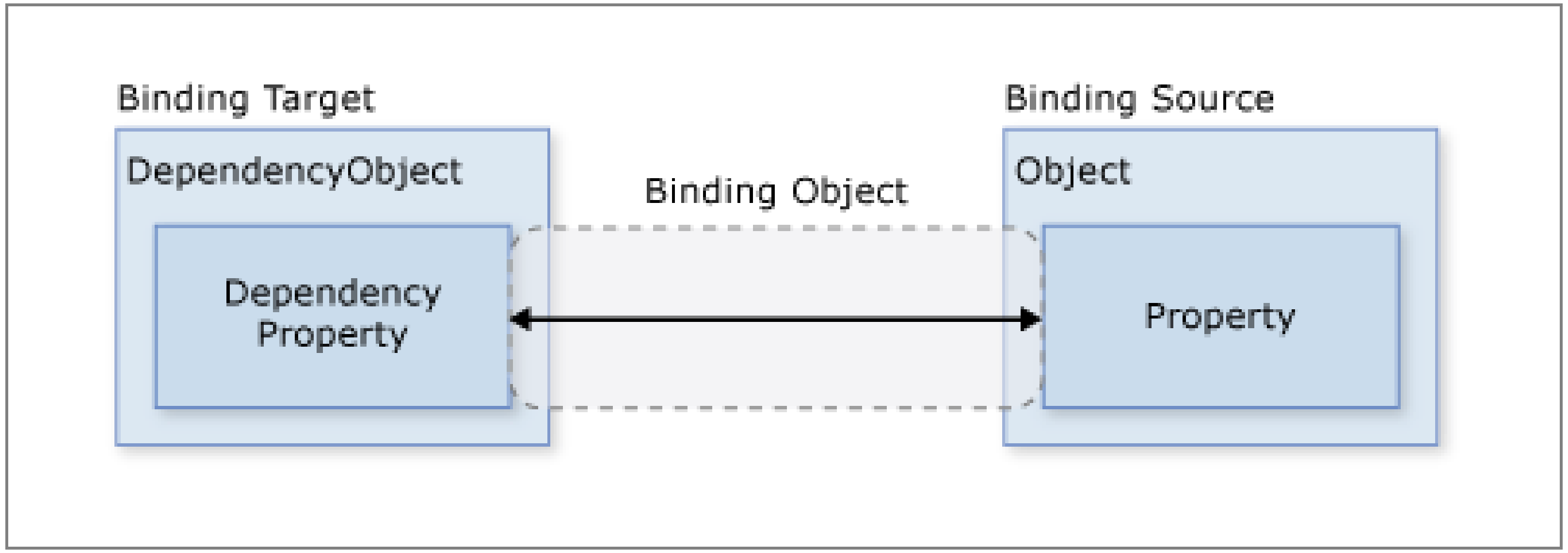
Car Name	Color	Brand
A6	White	Audi
Lexus 570	Black	Toyota
Ford Ranger Raptor	White	Ford

Understanding Data Binding

- Data binding is the process that establishes a connection between the app UI and the data it displays. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically
- Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change.
  - For example: if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change
- A typical use of data binding is to place server or local configuration data into forms or other UI controls. In WPF, this concept is expanded to include binding a broad range of properties to a variety of data sources

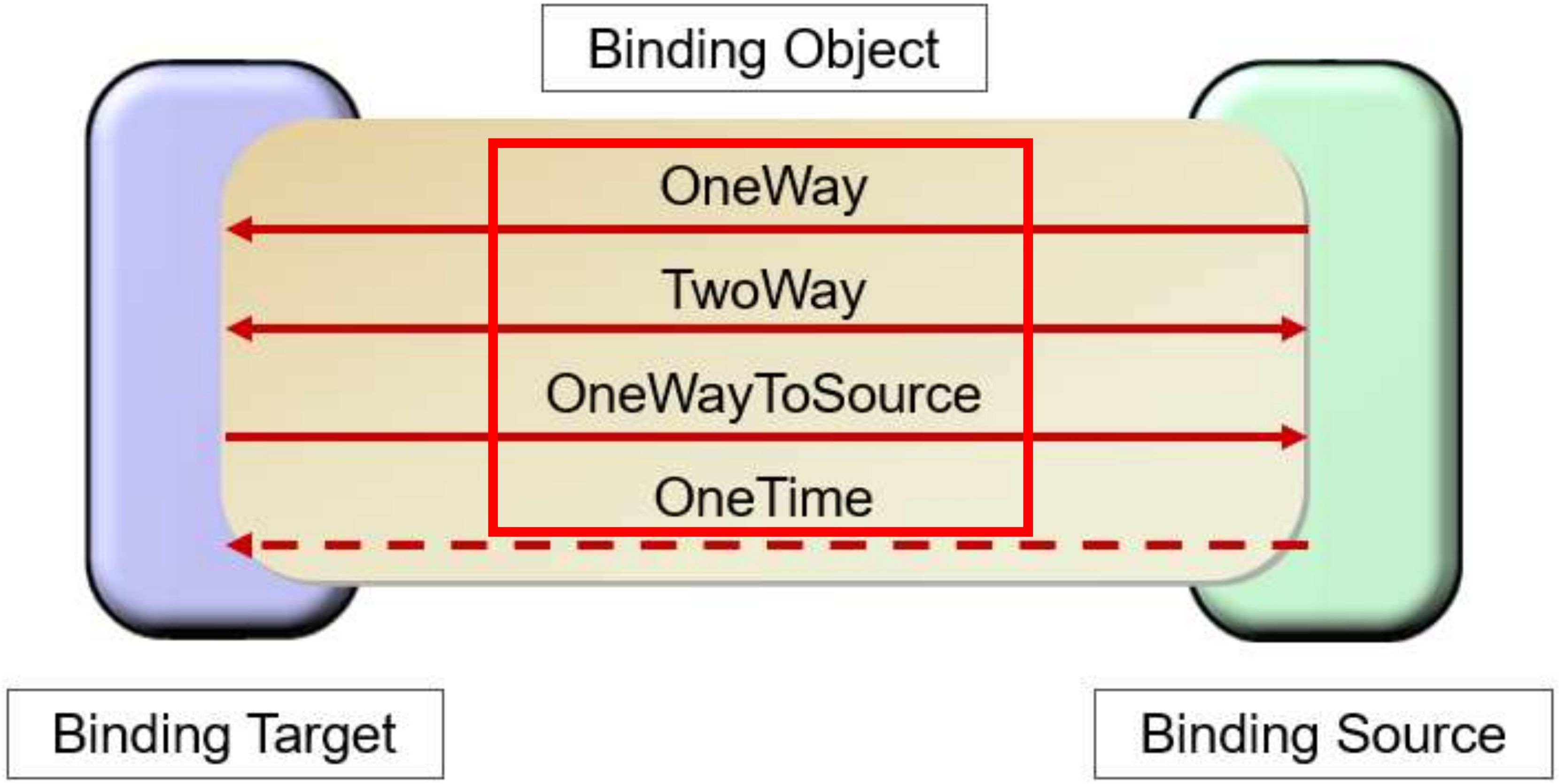
Understanding Data Binding

- Typically, each binding has four components:
  - A binding target object
  - A target property
  - A binding source
  - A path to the value in the binding source to use



Understanding Data Binding

- WPF supports four data-binding modes:



Understanding Data Binding

- **OneWay** binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. This type of binding is appropriate if the control being bound is implicitly read-only

- **TwoWay** binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully interactive UI scenarios. Most properties default to OneWay binding, but some dependency properties (typically properties of user-editable controls such as the TextBox.Text and CheckBox.IsChecked) default to TwoWay binding

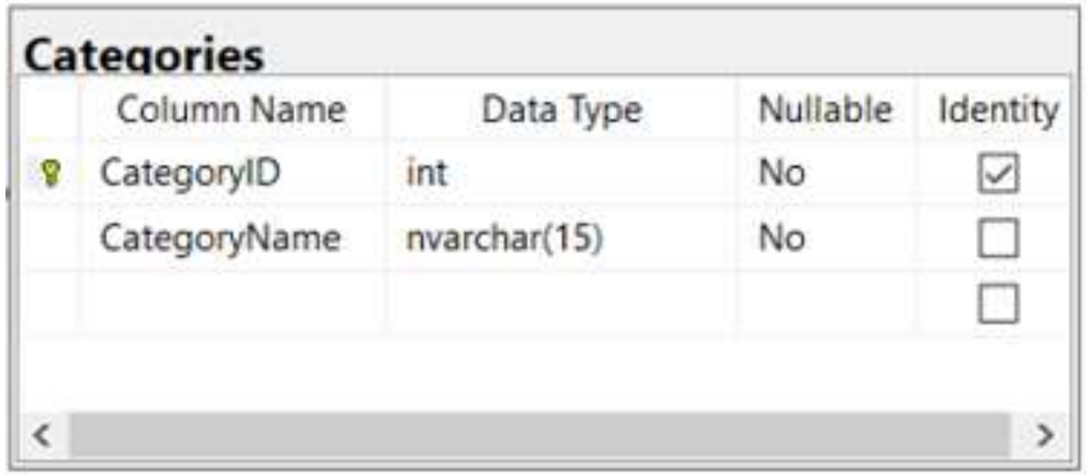
Understanding Data Binding

- **OneWayToSource** is the reverse of OneWay binding; it updates the source property when the target property changes. One example scenario is if we only need to reevaluate the source value from the UI

- **OneTime** is essentially a simpler form of OneWay binding that provides better performance in cases where the source value does not change. Updates the binding target when the application starts or when the data context changes. This type of binding is appropriate if we are using data where either a snapshot of the current state is appropriate to use or the data is truly static

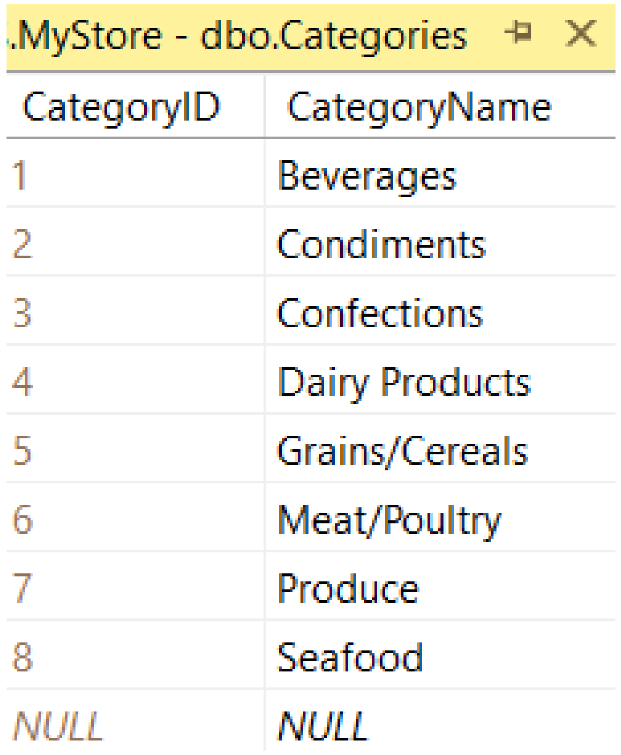
Access to Database Demonstration

- Create a sample database named **MyStore** includes a table named **Categories** as follows:



Categories

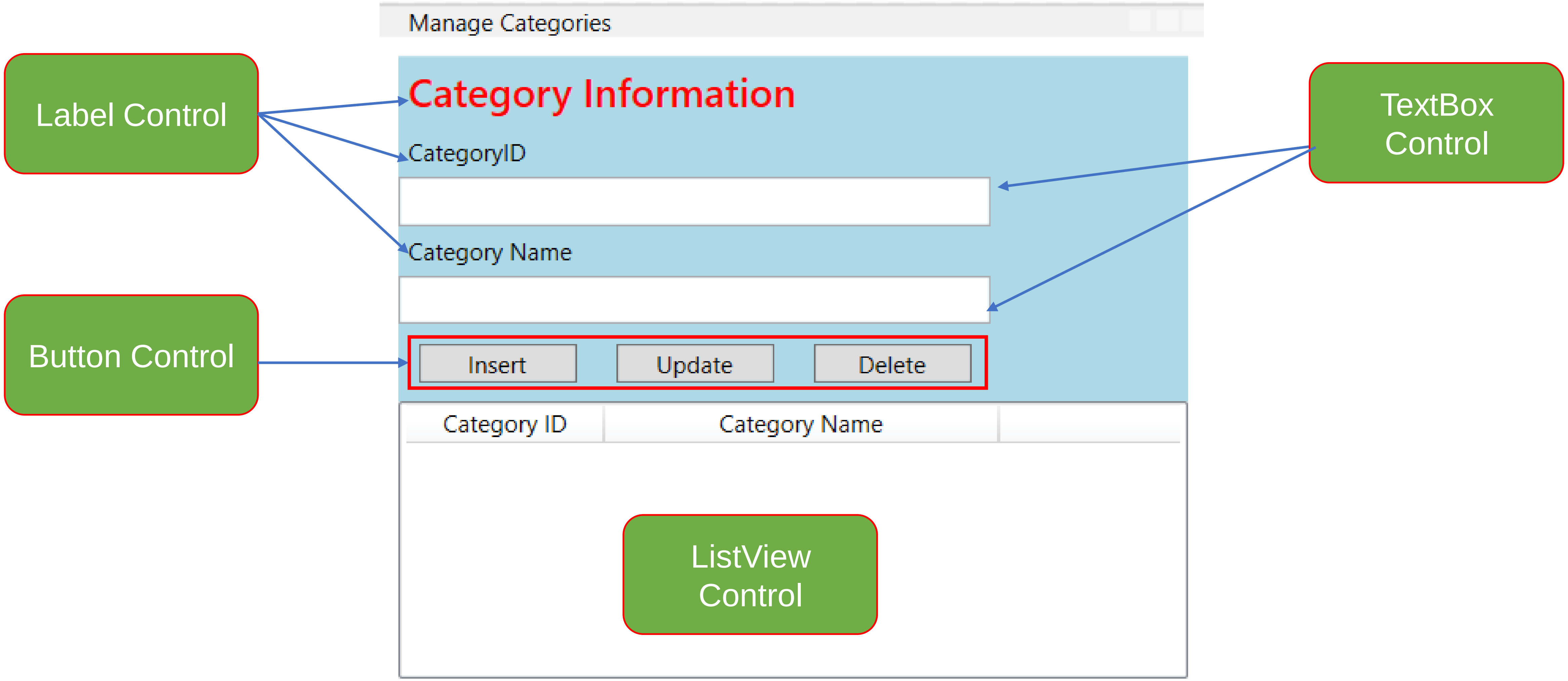
	Column Name	Data Type	Nullable	Identity
	CategoryID	int	No	☑
	CategoryName	nvarchar(15)	No	☐
				☐



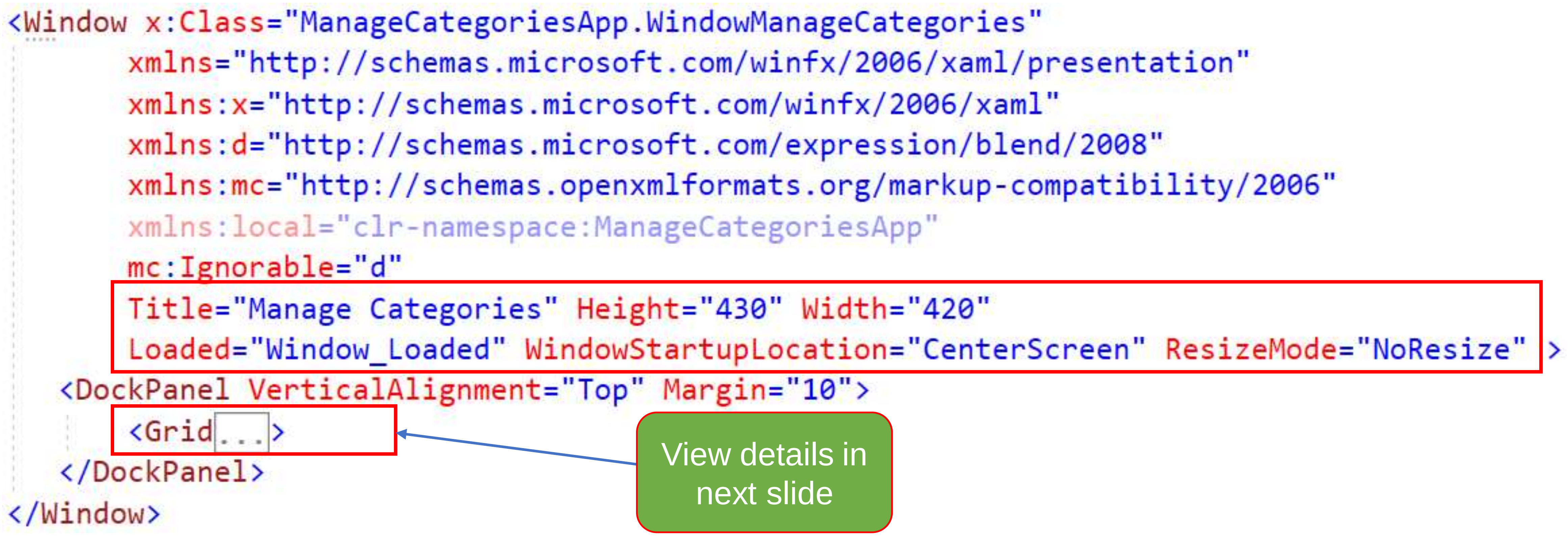
.MyStore - dbo.Categories

CategoryID	CategoryName
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood
NULL	*NULL*

1.Create a WPF app named **ManageCategoriesApp** includes a window named **WindowManageCategories.xaml** that has controls as follows :



XAML code of **WindowManageCategories.xaml**:



```
<Window x:Class="ManageCategoriesApp.WindowManageCategories"
        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:ManageCategoriesApp"
        mc:Ignorable="d"
        Title="Manage Categories" Height="430" Width="420"
        Loaded="Window_Loaded" WindowStartupLocation="CenterScreen" ResizeMode="NoResize" >
    <DockPanel VerticalAlignment="Top" Margin="10">
        <Grid...>
    </DockPanel>
</Window>
```

XAML code of **Grid** tag

```
<Grid>
     <Grid.RowDefinitions>
         <RowDefinition Height="*"/>
         <RowDefinition Height="*"/>
     </Grid.RowDefinitions>
     <StackPanel Background="LightBlue"  Orientation ="Vertical"
                HorizontalAlignment="Left" Width="400">

         <Label Name="lblInstruction" Foreground="Red" FontWeight="DemiBold"
            FontSize="20" Content="Category Information"/>

         <Label Name="lblCategoryID" Content="CategoryID"/>
         <TextBox Name="txtCategoryID" HorizontalAlignment="Left"
              IsReadOnly="True" Height="25" Width="300"
              Text="{Binding Path=CategoryID, Mode=OneWay} "
              DataContext="{Binding ElementName=lvCategories, Path=SelectedItem} " />

         <Label Name="lbCategoryName" Content="Category Name" />
         <TextBox Name="txtCategoryName" HorizontalAlignment="Left"
              Height="25" Width="300"  Text="{Binding Path=CategoryName, Mode=OneWay} "
              DataContext="{Binding ElementName=lvCategories, Path=SelectedItem} " />
```

Data binding: OneWay mode

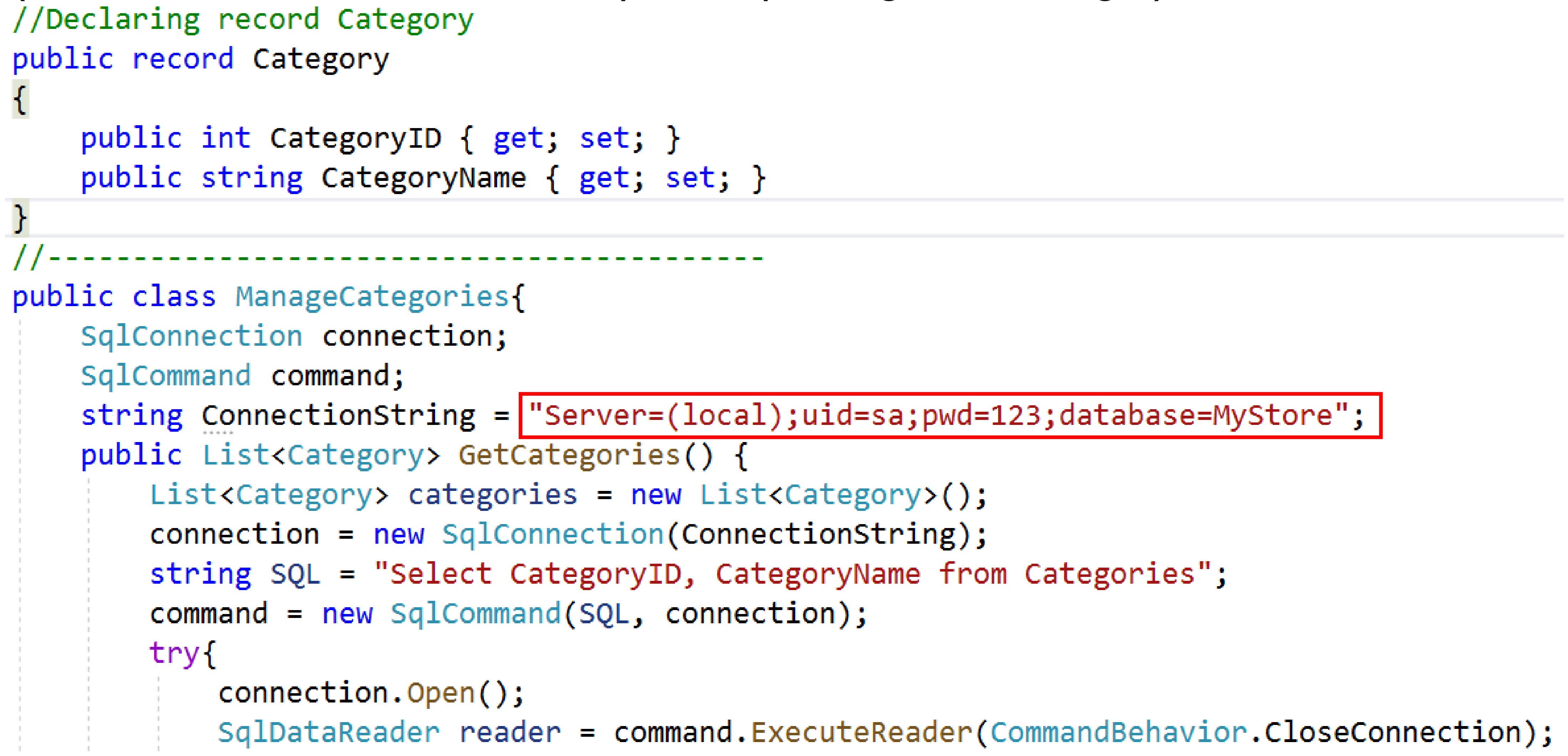
XAML code of **Grid** tag (cont.)

```
                <StackPanel Orientation="Horizontal"  HorizontalAlignment="Left">
                    <Button x:Name="btnInsert" Margin="10" Width="80" Content="Insert"
                        Click="btnInsert_Click" />
                    <Button x:Name="btnUpdate"  Margin="10" Width="80"  Content="Update"
                        Click="btnUpdate_Click" />
                    <Button x:Name="btnDelete" Margin="10"  Width="80" Content="Delete"
                        Click="btnDelete_Click" />
                </StackPanel>
            </StackPanel>
            <ListView Grid.Row="1" Name="lvCategories" Width="400" >
                <ListView.View>
                    <GridView>
                        <GridViewColumn Header="Category ID"  Width="100"
                                    DisplayMemberBinding="{Binding Path=CategoryID } "/>
                        <GridViewColumn Header="Category Name" Width="200"
                                    DisplayMemberBinding="{Binding Path=CategoryName} "/>
                    </GridView>
                </ListView.View>
            </ListView>
        </Grid>
```

Event Handler: Click

Data binding

2.Right-click on the project | **Add** | **Class**, named **ManageCategories.cs** then write codes as follows (Note: Install Microsoft.Data.SqlClient package from Nuget)



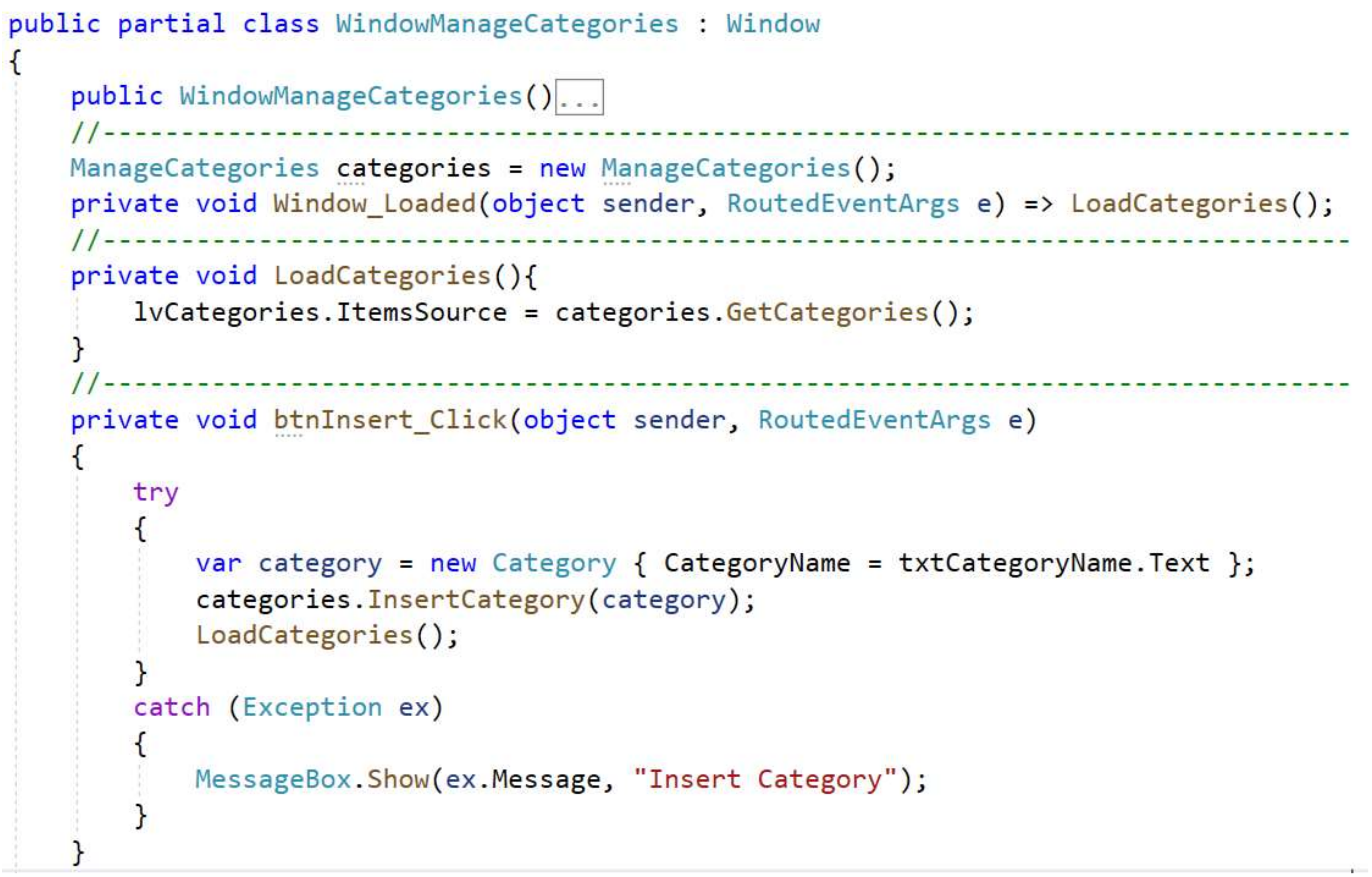
```
//Declaring record Category
public record Category
{
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
}
//----------------------------------------
public class ManageCategories{
    SqlConnection connection;
    SqlCommand command;
    string ConnectionString = "Server=(local);uid=sa;pwd=123;database=MyStore";
    public List<Category> GetCategories() {
        List<Category> categories = new List<Category>();
        connection = new SqlConnection(ConnectionString);
        string SQL = "Select CategoryID, CategoryName from Categories";
        command = new SqlCommand(SQL, connection);
        try{
            connection.Open();
            SqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection);
        if (reader.HasRows == true){
            while (reader.Read()){
                categories.Add(new Category{
                    CategoryID = reader.GetInt32("CategoryID"),
                    CategoryName = reader.GetString("CategoryName")
                });
            }//end while
        }//end if
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally
    {
        connection.Close();
    }
    return categories;
}//end GetCategories
//------------------------------------------
public void InsertCategory(Category category)
{
    connection = new SqlConnection(ConnectionString);
    //CategoryID is identity
    command = new SqlCommand("Insert Categories values(@CategoryName)", connection);
```

```
    try {
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally{
        connection.Close();
    }
}//end InsertCategory
```

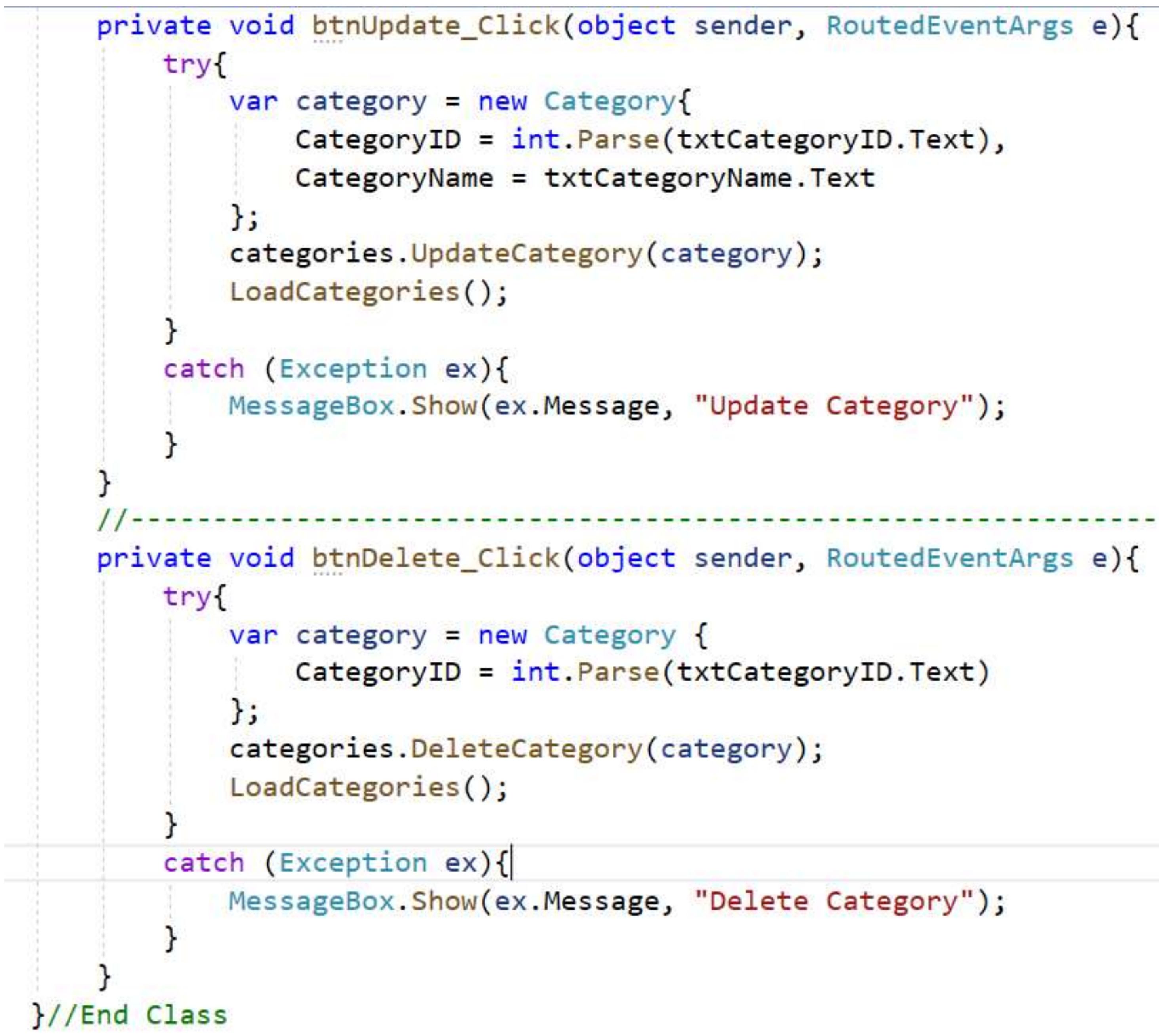
```
//-----------------------------------------
public void UpdateCategory(Category category){
    connection = new SqlConnection(ConnectionString);
    string SQL = "Update Categories set CategoryName=@CategoryName where CategoryID=@CategoryID";
    command = new SqlCommand(SQL, connection);
    command.Parameters.AddWithValue("@CategoryID",category.CategoryID);
    command.Parameters.AddWithValue("@CategoryName",category.CategoryName);
    try {
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally{
        connection.Close();
    }
}//end UpdateCategory
```

```
    //------------------------------------------
    public void DeleteCategory(Category category) {
        connection = new SqlConnection(ConnectionString);
        string SQL = "Delete Categories where CategoryID=@CategoryID";
        command = new SqlCommand(SQL, connection);
        command.Parameters.AddWithValue("@CategoryID", category.CategoryID);
        try {
            connection.Open();
            command.ExecuteNonQuery();
        }
        catch (Exception ex){
            throw new Exception(ex.Message);
        }
        finally{
            connection.Close();
        }
    }//end DeleteCategory
}//end ManageCategories
```

3.Write codes in **WindowManageCategories.xaml.cs** as follows:

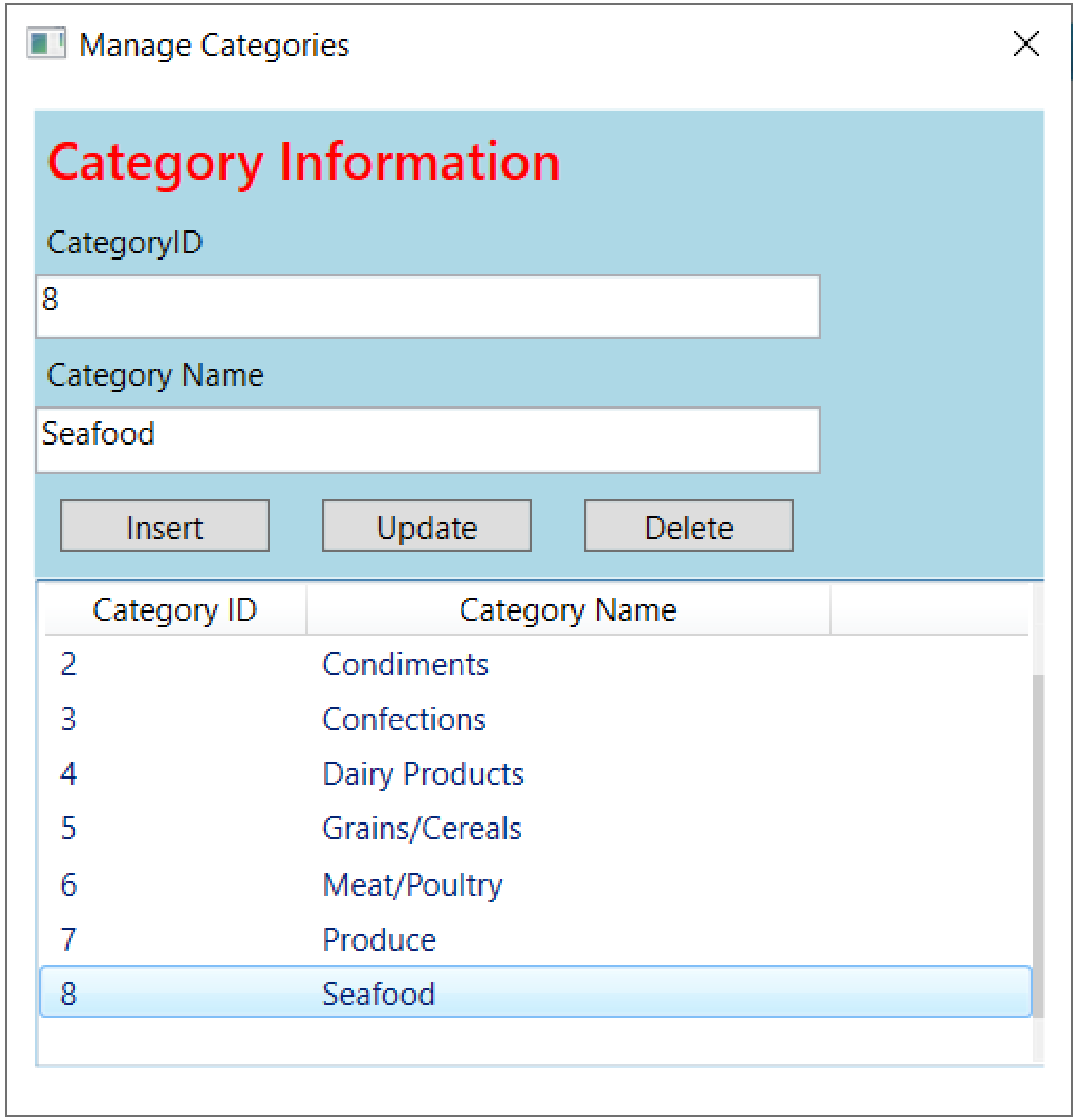


```
public partial class WindowManageCategories : Window
{
    public WindowManageCategories()...
    //-----------------------------------------------------------------------------
    ManageCategories categories = new ManageCategories();
    private void Window_Loaded(object sender, RoutedEventArgs e) => LoadCategories();
    //-----------------------------------------------------------------------------
    private void LoadCategories(){
        lvCategories.ItemsSource = categories.GetCategories();
    }
    //-----------------------------------------------------------------------------
    private void btnInsert_Click(object sender, RoutedEventArgs e)
    {
        try
        {
            var category = new Category { CategoryName = txtCategoryName.Text };
            categories.InsertCategory(category);
            LoadCategories();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, "Insert Category");
        }
    }
```



```
    private void btnUpdate_Click(object sender, RoutedEventArgs e){
        try{
            var category = new Category{
                CategoryID = int.Parse(txtCategoryID.Text),
                CategoryName = txtCategoryName.Text
            };
            categories.UpdateCategory(category);
            LoadCategories();
        }
        catch (Exception ex){
            MessageBox.Show(ex.Message, "Update Category");
        }
    }
    //-----------------------------------------------------------------
    private void btnDelete_Click(object sender, RoutedEventArgs e){
        try{
            var category = new Category {
                CategoryID = int.Parse(txtCategoryID.Text)
            };
            categories.DeleteCategory(category);
            LoadCategories();
        }
        catch (Exception ex){
            MessageBox.Show(ex.Message, "Delete Category");
        }
    }
}//End Class
```

4. Press **Ctrl+F5** to run project and view the output



Manage Categories

Category Information

CategoryID

8

Category Name

Seafood

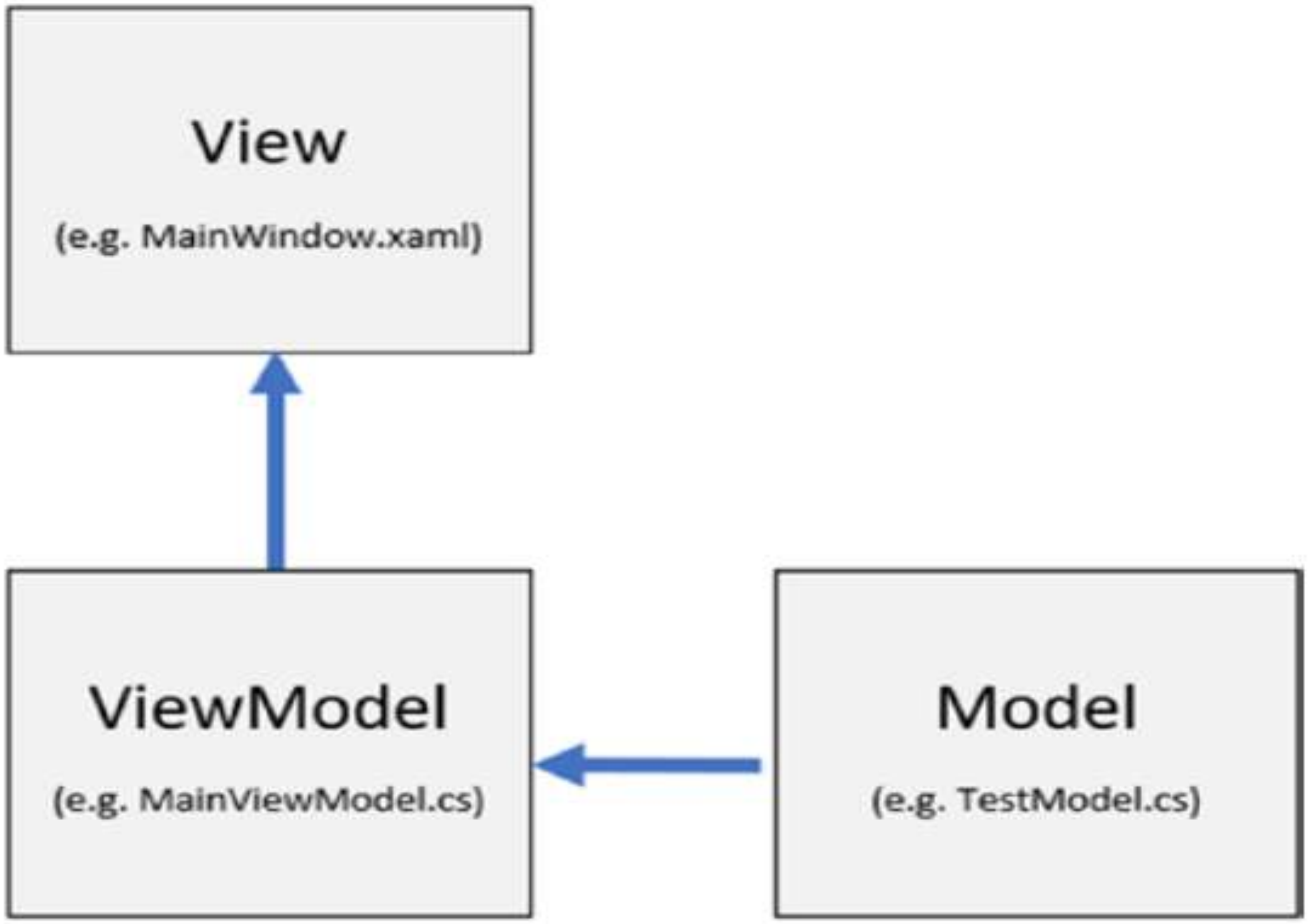
Category ID	Category Name
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

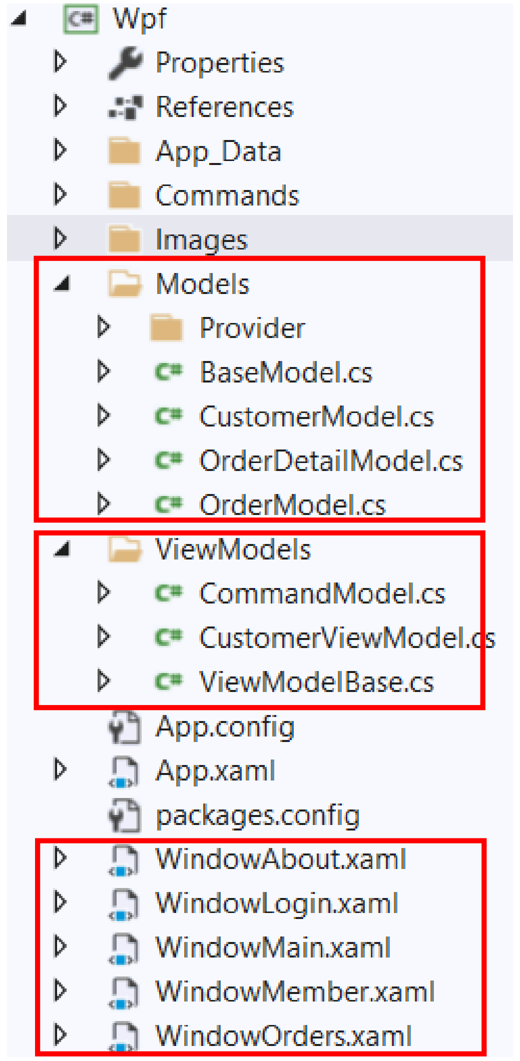
Introduction to MVVM Pattern (Model-View-ViewModel)

MVVM Pattern (Model-View-ViewModel)

- MVVM was introduced by John Gossman in 2005 specifically for use with WPF as a concrete application of Martin Fowler's broader Presentation Model pattern
- The implementation of an application, based on the MVVM patterns, uses various platform capabilities that are available in some form for WPF, Silverlight Desktop/web, and on Windows. Many commercial applications, including Microsoft Expression products, were built following MVVM
- The Model, View, ViewModel (MVVM pattern) is all about guiding us in how to organize and structure our code to write **maintainable**, **testable** and **extensible** applications

MVVM Pattern (Model-View-ViewModel)





```
◢ Wpf
  ▷ Properties
  ▷ References
  ▷ App_Data
  ▷ Commands
  ▷ Images
  ◢ Models
    ▷ Provider
    ▷ BaseModel.cs
    ▷ CustomerModel.cs
    ▷ OrderDetailModel.cs
    ▷ OrderModel.cs
  ◢ ViewModels
    ▷ CommandModel.cs
    ▷ CustomerViewModel.cs
    ▷ ViewModelBase.cs
    App.config
  ▷ App.xaml
    packages.config
  ▷ WindowAbout.xaml
  ▷ WindowLogin.xaml
  ▷ WindowMain.xaml
  ▷ WindowMember.xaml
  ▷ WindowOrders.xaml
```

MVVM Pattern (Model-View-ViewModel)

- **Model**: The model is the object representation of data. In MVVM, models are conceptually the same as the models from data access layer (DAL)
- **ViewModel** is a non-visual class. The MVVM Design Pattern does not derive from any WPF based class. The ViewModel is unaware of the view directly. Communication between the View and ViewModel is through some property and binding. Models are connected directly to the ViewModel and invoke a method by the model class, it knows what the model has, like properties, methods etcetera and also is aware of what the view needs
- **View**: The View is the graphical interface incharge of displaying data to users and interacting with them. In a WPF application, the View might be a UserControl, a Window, or a Page

```
// Model
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
}

// ViewModel
public class StudentViewModel : INotifyPropertyChanged
{
    private Student _student;

    public string Name
    {
        get { return _student.Name; }
        set
        {
            _student.Name = value;
            OnPropertyChanged("Name");
        }
    }

    public int Age
    {
        get { return _student.Age; }
        set
        {
            _student.Age = value;
            OnPropertyChanged("Age");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string name)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

```
// View (XAML)
<Window x:Class = "MVVMExample.MainWindow"
        xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns: x = "http://schemas.microsoft.com/winfx/2006/xaml"
        Title = "MVVM Example" Height = "200" Width = "400" >
    <Grid>
        <StackPanel>
            <TextBox Text = "{Binding Name, UpdateSourceTrigger=PropertyChanged}" />
            <TextBox Text = "{Binding Age, UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
    </Grid>
</Window>
```

MVVM Advantages

- **Maintainability**
  - A clean separation of different kinds of code should make it easier to go into one or several of those more granular and focused parts and make changes without worrying. That means we can remain agile and keep moving out to new releases quickly
- **Testability**
  - With MVVM each piece of code is more granular and if it is implemented right our external and internal dependences are in separate pieces of code from the parts with the core logic that we would like to test. That makes it a lot easier to write unit tests against a core logic
- **Extensibility**
  - It sometimes overlaps with maintainability, because of the clean separation boundaries and more granular pieces of code. We have a better chance of making any of those parts more reusable

Summary

- Concepts were introduced:
  - Overview Windows Presentation Foundation (WPF)
  - Overview XAML(eXtensible Application Markup Language) in WPF
  - Explain about Controls and Layouts in WPF
  - Explain about Styles and Templates in WPF
  - Explain about WPF Data-Binding Model
  - Demo create WPF application by dotnet CLI and Visual Studio.NET
  - Demo access to the database by WPF Application
  - Explain about MVVM Pattern (Model-View-ViewModel)

Lab and Assigment

1. Do Hands-on Lab:

    Lab_01_AutomobileManagement_Using_EntityFramework and WPF

2. Do Assigment:

    Assignment_01_HotelManagement.pdf