

Contents

Windows Presentation Foundation for .NET Core

Get started

Overview

Create a WPF application

Migration

Differences from .NET Framework

Migrate from .NET Framework

Fundamentals

Windows

Overview

Dialogs boxes

Common tasks

Display a message box

Open a window

Close a window

Display a system dialog box

Get or set the main window

Controls

Styles and templates

Overview

Common tasks

Create and apply a style

Create and apply a template

Data binding

Overview

Declare a binding

Binding sources

Common tasks

Bind to an enumeration

Systems

Resources

Overview

Merged dictionaries

Resources in code

Common tasks

Define and reference resources

Use application resources

Use system resources

XAML with WPF

Overview

XAML Language Reference

Desktop Guide (WPF .NET)

4/15/2021 • 17 minutes to read • [Edit Online](#)

Welcome to the Desktop Guide for Windows Presentation Foundation (WPF), a UI framework that is resolution-independent and uses a vector-based rendering engine, built to take advantage of modern graphics hardware. WPF provides a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles, templates, documents, media, text, and typography. WPF is part of .NET, so you can build applications that incorporate other elements of the .NET API.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

There are two implementations of WPF:

1. .NET version (this guide):

An open-source implementation of WPF hosted on [GitHub](#), which runs on .NET 5 or higher (including .NET Core 3.1). The XAML designer requires, at a minimum, [Visual Studio 2019 version 16.8](#).

Even though .NET is a cross-platform technology, WPF isn't and only runs on Windows.

2. .NET Framework 4 version:

The .NET Framework implementation of WPF that's supported by Visual Studio 2019 and Visual Studio 2017.

.NET Framework 4 is a Windows-only version of .NET and is considered a Windows Operating System component. This version of WPF is distributed with .NET Framework. For more information about the .NET Framework version of WPF, see [Introduction to WPF for .NET Framework](#).

This overview is intended for newcomers and covers the key capabilities and concepts of WPF. To learn how to create a WPF app, see [Tutorial: Create a new WPF app](#).

Why migrate from .NET Framework

WPF for .NET 5.0 provides new features and enhancements over .NET Framework. To learn how to migrate an app, see [How to migrate a WPF desktop app to .NET 5](#).

Program with WPF

WPF exists as a subset of .NET types that are, mostly located in the [System.Windows](#) namespace. If you have previously built applications with .NET with frameworks like ASP.NET and Windows Forms, the fundamental WPF programming experience should be familiar, you:

- Instantiate classes
- Set properties
- Call methods
- Handle events

WPF includes more programming constructs that enhance properties and events: [dependency properties](#) and

[routed events](#).

Markup and code-behind

WPF lets you develop an application using both *markup* and *code-behind*, an experience with which ASP.NET developers should be familiar. You generally use XAML markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior. This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced because appearance-specific markup isn't tightly coupled with behavior-specific code.
- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.
- [Globalization and localization](#) for WPF applications is simplified.

Markup

XAML is an XML-based markup language that implements an application's appearance declaratively. You typically use it to define windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="Window with Button"
  Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Specifically, this XAML defines a window and a button by using the `Window` and `Button` elements. Each element is configured with attributes, such as the `Window` element's `Title` attribute to specify the window's title-bar text. At run time, WPF converts the elements and attributes that are defined in markup to instances of WPF classes. For example, the `Window` element is converted to an instance of the `Window` class whose `Title` property is the value of the `Title` attribute.

The following figure shows the user interface (UI) that is defined by the XAML in the previous example:



Since XAML is XML-based, the UI that you compose with it's assembled in a hierarchy of nested elements that is known as an [element tree](#). The element tree provides a logical and intuitive way to create and manage UIs.

Code-behind

The main behavior of an application is to implement the functionality that responds to user interactions. For example clicking a menu or button, and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind. The following example shows the updated markup from the previous example and the code-behind:

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

    <!-- Add button to window -->
    <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>

```

The updated markup defines the `xmlns:x` namespace and maps it to the schema that adds support for the code-behind types. The `x:Class` attribute is used to associate a code-behind class to this specific XAML markup. Considering this attribute is declared on the `<Window>` element, the code-behind class must inherit from the `Window` class.

```

using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}

```

```

Namespace SDKSample

    Partial Public Class AWindow
        Inherits System.Windows.Window

        Public Sub New()

            ' InitializeComponent call is required to merge the UI
            ' that is defined in markup with this class, including
            ' setting properties and registering event handlers
            InitializeComponent()

        End Sub

        Private Sub button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)

            ' Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!")

        End Sub

    End Class

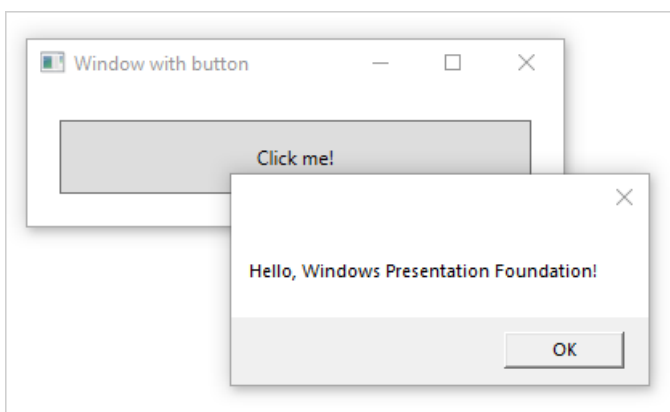
End Namespace

```

`InitializeComponent` is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class. (`InitializeComponent` is generated for you when your application is built, which is why you don't need to implement it manually.) The combination of `x:Class` and `InitializeComponent` ensure that your implementation is correctly initialized whenever it's created.

Notice that in the markup the `<Button>` element defined a value of `button_click` for the `Click` attribute. With the markup and code-behind initialized and working together, the `Click` event for the button is automatically mapped to the `button_click` method. When the button is clicked, the event handler is invoked and a message box is displayed by calling the `System.Windows.MessageBox.Show` method.

The following figure shows the result when the button is clicked:



Input and commands

Controls most often detect and respond to user input. The WPF input system uses both direct and routed events to support text input, focus management, and mouse positioning.

Applications often have complex input requirements. WPF provides a command system that separates user-input actions from the code that responds to those actions. The command system allows for multiple sources to invoke the same command logic. For example, take the common editing operations used by different applications: **Copy**, **Cut**, and **Paste**. These operations can be invoked by using different user actions if they're implemented by using commands.

Controls

The user experiences that are delivered by the application model are constructed controls. In WPF, *control* is an umbrella term that applies to a category of WPF classes that have the following characteristics:

- Hosted in either a window or a page.
- Have a user interface.
- Implement some behavior.

For more information, see [Controls](#).

WPF controls by function

The built-in WPF controls are listed here:

- **Buttons:** [Button](#) and [RepeatButton](#).
- **Data Display:** [DataGrid](#), [ListView](#), and [TreeView](#).
- **Date Display and Selection:** [Calendar](#) and [DatePicker](#).
- **Dialog Boxes:** [OpenFileDialog](#), [PrintDialog](#), and [SaveFileDialog](#).
- **Digital Ink:** [InkCanvas](#) and [InkPresenter](#).
- **Documents:** [DocumentViewer](#), [FlowDocumentPageViewer](#), [FlowDocumentReader](#), [FlowDocumentScrollViewer](#), and [StickyNoteControl](#).
- **Input:** [TextBox](#), [RichTextBox](#), and [PasswordBox](#).
- **Layout:** [Border](#), [BulletDecorator](#), [Canvas](#), [DockPanel](#), [Expander](#), [Grid](#), [GridView](#), [GridSplitter](#), [GroupBox](#), [Panel](#), [ResizeGrip](#), [Separator](#), [ScrollBar](#), [ScrollViewer](#), [StackPanel](#), [Thumb](#), [Viewbox](#), [VirtualizingStackPanel](#), [Window](#), and [WrapPanel](#).
- **Media:** [Image](#), [MediaElement](#), and [SoundPlayerAction](#).
- **Menus:** [ContextMenu](#), [Menu](#), and [ToolBar](#).
- **Navigation:** [Frame](#), [Hyperlink](#), [Page](#), [NavigationWindow](#), and [TabControl](#).
- **Selection:** [CheckBox](#), [ComboBox](#), [ListBox](#), [RadioButton](#), and [Slider](#).
- **User Information:** [AccessText](#), [Label](#), [Popup](#), [ProgressBar](#), [StatusBar](#), [TextBlock](#), and [ToolTip](#).

Layout

When you create a user interface, you arrange your controls by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. Rather than forcing you to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you.

The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. The layout system also manages the negotiation between controls to determine the layout. The negotiation is a two-step process: first, a control tells its parent what location and size it requires. Second, the parent tells the control what space it can have.

The layout system is exposed to child controls through base WPF classes. For common layouts such as grids, stacking, and docking, WPF includes several layout controls:

- [Canvas](#): Child controls provide their own layout.
- [DockPanel](#): Child controls are aligned to the edges of the panel.

- [Grid](#): Child controls are positioned by rows and columns.
- [StackPanel](#): Child controls are stacked either vertically or horizontally.
- [VirtualizingStackPanel](#): Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.
- [WrapPanel](#): Child controls are positioned in left-to-right order and wrapped to the next line when there isn't enough space on the current line.

The following example uses a [DockPanel](#) to lay out several [TextBox](#) controls:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.LayoutWindow"
  Title="Layout with the DockPanel" Height="143" Width="319">

  <!--DockPanel to layout four text boxes-->
  <DockPanel>
    <TextBox DockPanel.Dock="Top">Dock = "Top"</TextBox>
    <TextBox DockPanel.Dock="Bottom">Dock = "Bottom"</TextBox>
    <TextBox DockPanel.Dock="Left">Dock = "Left"</TextBox>
    <TextBox Background="White">This TextBox "fills" the remaining space.</TextBox>
  </DockPanel>

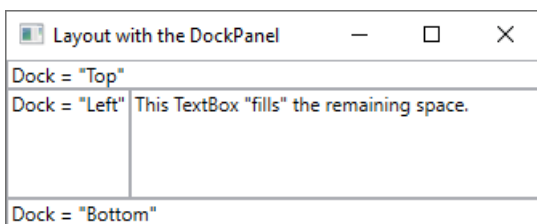
</Window>
```

The [DockPanel](#) allows the child [TextBox](#) controls to tell it how to arrange them. To do this, the [DockPanel](#) implements a [Dock](#) attached property that is exposed to the child controls to allow each of them to specify a dock style.

NOTE

A property that's implemented by a parent control for use by child controls is a WPF construct called an [attached property](#).

The following figure shows the result of the XAML markup in the preceding example:



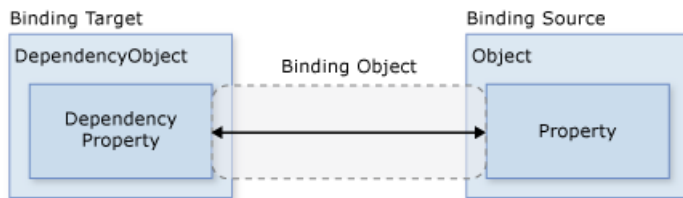
Data binding

Most applications are created to provide users with the means to view and edit data. For WPF applications, the work of storing and accessing data is already provided for by many different .NET data access libraries such as SQL and Entity Framework Core. After the data is accessed and loaded into an application's managed objects, the hard work for WPF applications begins. Essentially, this involves two things:

1. Copying the data from the managed objects into controls, where the data can be displayed and edited.
2. Ensuring that changes made to data by using controls are copied back to the managed objects.

To simplify application development, WPF provides a powerful data binding engine to automatically handle

these steps. The core unit of the data binding engine is the [Binding](#) class, whose job is to bind a control (the binding target) to a data object (the binding source). This relationship is illustrated by the following figure:



WPF supports declaring bindings in the XAML markup directly. For example, the following XAML code binds the [Text](#) property of the [TextBox](#) to the `Name` property of an object using the "`{Binding ... }`" XAML syntax. This assumes there's a data object set to the [DataContext](#) property of the `Window` with a `Name` property.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.DataBindingWindow">

  <!-- Bind the TextBox to the data source (TextBox.Text to Person.Name) -->
  <TextBox Name="personNameTextBox" Text="{Binding Path=Name}" />

</Window>
```

The WPF data binding engine provides more than just binding, it provides validation, sorting, filtering, and grouping. Furthermore, data binding supports the use of data templates to create custom user interface for bound data.

For more information, see [Data binding overview](#).

Graphics & animation

WPF provides an extensive and flexible set of graphics features that have the following benefits:

- **Resolution-independent and device-independent graphics.** The basic unit of measurement in the WPF graphics system is the device-independent pixel, which is 1/96th of an inch, and provides the foundation for resolution-independent and device-independent rendering. Each device-independent pixel automatically scales to match the dots-per-inch (dpi) setting of the system it renders on.
- **Improved precision.** The WPF coordinate system is measured with double-precision floating-point numbers rather than single-precision. Transformations and opacity values are also expressed as double-precision. WPF also supports a wide color gamut (sRGB) and provides integrated support for managing inputs from different color spaces.
- **Advanced graphics and animation support.** WPF simplifies graphics programming by managing animation scenes for you; there's no need to worry about scene processing, rendering loops, and bilinear interpolation. Additionally, WPF provides hit-testing support and full alpha-compositing support.
- **Hardware acceleration.** The WPF graphics system takes advantage of graphics hardware to minimize CPU usage.

2D graphics

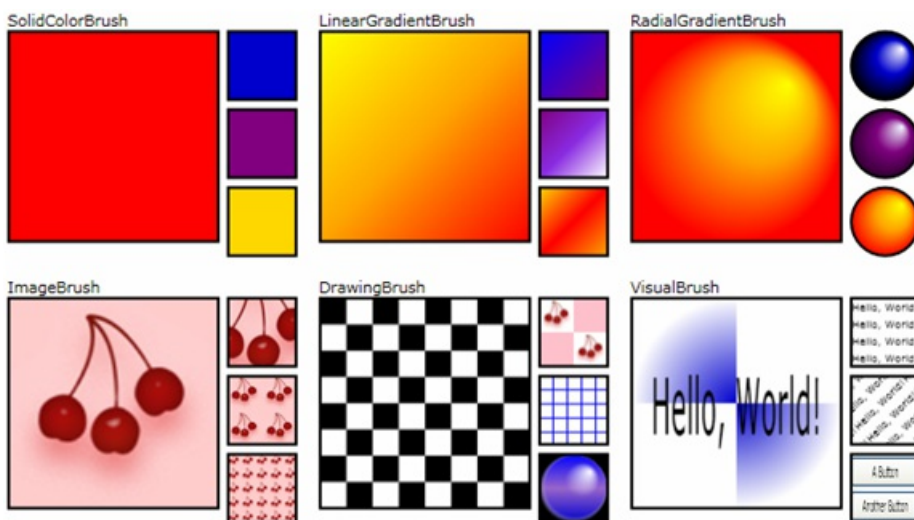
WPF provides a library of common vector-drawn 2D shapes, such as the rectangles and ellipses. The shapes aren't just for display; shapes implement many of the features that you expect from controls, including keyboard and mouse input.

The 2D shapes provided by WPF cover the standard set of basic shapes. However, you may need to create custom shapes to help the design of a customized user interface. WPF provides geometries to create a custom

shape that can be drawn directly, used as a brush, or used to clip other shapes and controls.

For more information, see [Geometry overview](#).

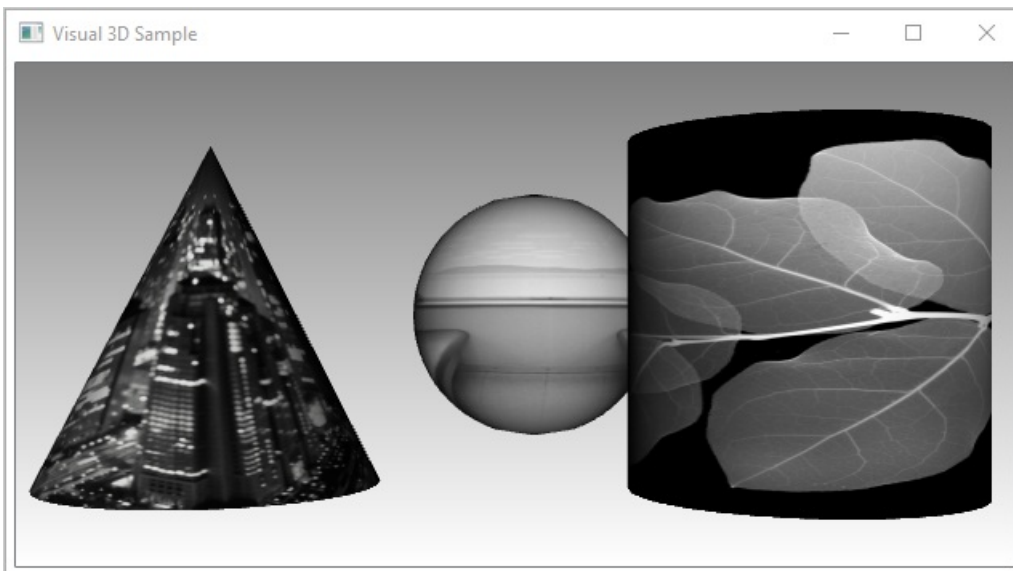
A subset of WPF 2D capabilities includes visual effects, such as gradients, bitmaps, drawings, painting with videos, rotation, scaling, and skewing. These effects are all achieved with brushes. The following figure shows some examples:



For more information, see [WPF brushes overview](#).

3D rendering

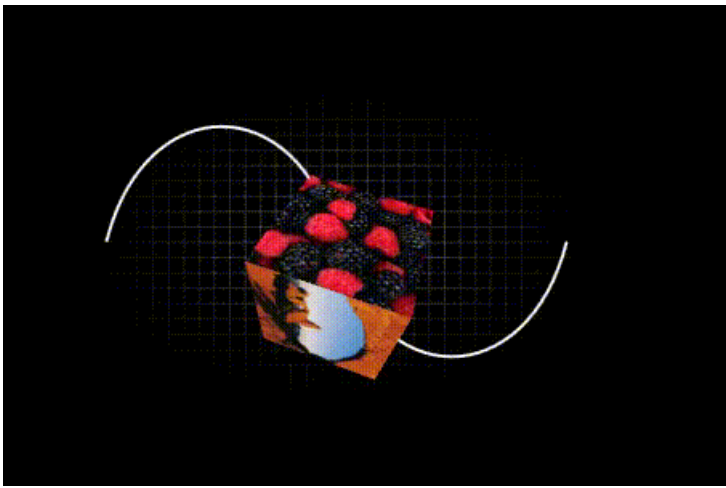
WPF also includes 3D rendering capabilities that integrate with 2D graphics to allow the creation of more exciting and interesting user interfaces. For example, the following figure shows 2D images rendered onto 3D shapes:



For more information, see [3D graphics overview](#).

Animation

WPF animation support lets you make controls grow, shake, spin, and fade, to create interesting page transitions, and more. You can animate most WPF classes, even custom classes. The following figure shows a simple animation in action:



For more information, see [Animation overview](#).

Text and typography

To provide high-quality text rendering, WPF offers the following features:

- OpenType font support.
- ClearType enhancements.
- High performance that takes advantage of hardware acceleration.
- Integration of text with media, graphics, and animation.
- International font support and fallback mechanisms.

As a demonstration of text integration with graphics, the following figure shows the application of text decorations:

Basic Text Decorations with XAML

The lazy dog ~~The lazy dog~~ The lazy dog The lazy dog

Changing the Color of a Text Decoration with XAML

The lazy dog ~~The lazy dog~~ The lazy dog The lazy dog

Creating Dash Text Decorations with XAML

The lazy dog ~~The lazy dog~~ The lazy dog The lazy dog

For more information, see [Typography in Windows Presentation Foundation](#).

Customize WPF apps

Up to this point, you've seen the core WPF building blocks for developing applications:

- You use the application model to host and deliver application content, which consists mainly of controls.
- To simplify the arrangement of controls in a user interface, you use the WPF layout system.
- You use data binding to reduce the work of integrating your user interface with data.
- To enhance the visual appearance of your application, you use the comprehensive range of graphics, animation, and media support provided by WPF.

Often, though, the basics aren't enough for creating and managing a truly distinct and visually stunning user experience. The standard WPF controls might not integrate with the desired appearance of your application. Data might not be displayed in the most effective way. Your application's overall user experience may not be suited to the default look and feel of Windows themes.

For this reason, WPF provides various mechanisms for creating unique user experiences.

Content Model

The main purpose of most of the WPF controls is to display content. In WPF, the type and number of items that can constitute the content of a control is referred to as the control's *content model*. Some controls can contain a single item and type of content. For example, the content of a [TextBox](#) is a string value that is assigned to the [Text](#) property.

Other controls, however, can contain multiple items of different types of content; the content of a [Button](#), specified by the [Content](#) property, can contain various items including layout controls, text, images, and shapes.

For more information on the kinds of content that is supported by various controls, see [WPF content model](#).

Triggers

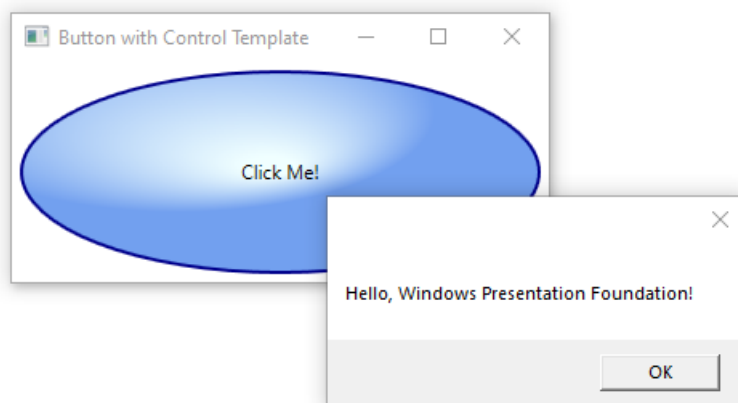
Although the main purpose of XAML markup is to implement an application's appearance, you can also use XAML to implement some aspects of an application's behavior. One example is the use of triggers to change an application's appearance based on user interactions. For more information, see [Styles and templates](#).

Templates

The default user interfaces for WPF controls are typically constructed from other controls and shapes. For example, a [Button](#) is composed of both [ButtonChrome](#) and [ContentPresenter](#) controls. The [ButtonChrome](#) provides the standard button appearance, while the [ContentPresenter](#) displays the button's content, as specified by the [Content](#) property.

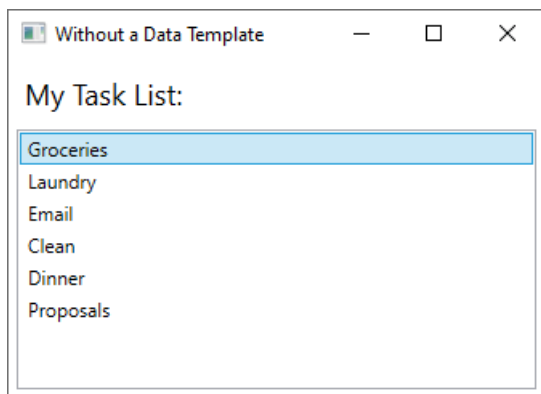
Sometimes the default appearance of a control may conflict with the overall appearance of an application. In this case, you can use a [ControlTemplate](#) to change the appearance of the control's user interface without changing its content and behavior.

For example, a [Button](#) raises the [Click](#) event when it's clicked. By changing the template of a button to display an [Ellipse](#) shape, the visual of the aspect of the control has changed, but the functionality hasn't. You can still click on the visual aspect of the control and the [Click](#) event is raised as expected.

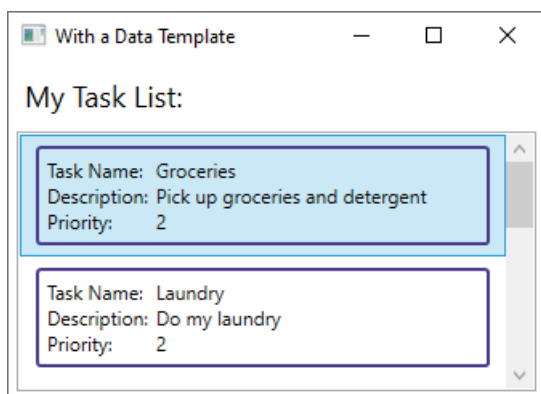


Data templates

Whereas a control template lets you specify the appearance of a control, a data template lets you specify the appearance of a control's content. Data templates are frequently used to enhance how bound data is displayed. The following figure shows the default appearance for a [ListBox](#) that is bound to a collection of [Task](#) objects, where each task has a name, description, and priority:



The default appearance is what you would expect from a [ListBox](#). However, the default appearance of each task contains only the task name. To show the task name, description, and priority, the default appearance of the [ListBox](#) control's bound list items must be changed by using a [DataTemplate](#). Here is an example of applying a data template that was created for the `Task` object.



The [ListBox](#) retains its behavior and overall appearance and only the appearance of the content being displayed by the list box has changed.

For more information, see [Data templating overview](#).

Styles

Styles enable developers and designers to standardize on a particular appearance for their product. WPF provides a strong style model, the foundation of which is the [Style](#) element. Styles can apply property values to types. They can be applied automatically to the everything according to the type or individual objects when referenced. The following example creates a style that sets the background color for every [Button](#) on the window to `Orange` :

```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.StyleWindow"
  Title="Styles">

  <Window.Resources>
    <!-- Style that will be applied to all buttons for this window -->
    <Style TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Orange" />
      <Setter Property="BorderBrush" Value="Crimson" />
      <Setter Property="FontSize" Value="20" />
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Window.Resources>
  <StackPanel>

    <!-- This button will have the style applied to it -->
    <Button>Click Me!</Button>

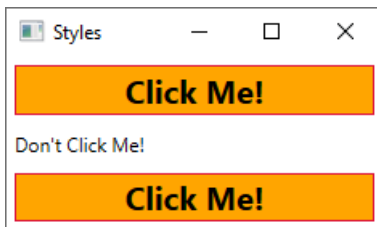
    <!-- This label will not have the style applied to it -->
    <Label>Don't Click Me!</Label>

    <!-- This button will have the style applied to it -->
    <Button>Click Me!</Button>

  </StackPanel>
</Window>

```

Because this style targets all [Button](#) controls, the style is automatically applied to all the buttons in the window, as shown in the following figure:



For more information, see [Styles and templates](#).

Resources

Controls in an application should share the same appearance, which can include anything from fonts and background colors to control templates, data templates, and styles. You can use WPF's support for user interface resources to encapsulate these resources in a single location for reuse.

The following example defines a common background color that is shared by a [Button](#) and a [Label](#):

```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.ResourcesWindow"
  Title="Resources Window">

  <!-- Define window-scoped background color resource -->
  <Window.Resources>
    <SolidColorBrush x:Key="defaultBackground" Color="Red" />
  </Window.Resources>

  <!-- Button background is defined by window-scoped resource -->
  <Button Background="{StaticResource defaultBackground}">One Button</Button>

  <!-- Label background is defined by window-scoped resource -->
  <Label Background="{StaticResource defaultBackground}">One Label</Label>
</Window>

```

For more information, see [How to define and reference a WPF resource](#).

Custom controls

Although WPF provides a host of customization support, you may encounter situations where existing WPF controls do not meet the needs of either your application or its users. This can occur when:

- The user interface that you require cannot be created by customizing the look and feel of existing WPF implementations.
- The behavior that you require isn't supported (or not easily supported) by existing WPF implementations.

At this point, however, you can take advantage of one of three WPF models to create a new control. Each model targets a specific scenario and requires your custom control to derive from a particular WPF base class. The three models are listed here:

- **User Control Model**

A custom control derives from [UserControl](#) and is composed of one or more other controls.

- **Control Model** A custom control derives from [Control](#) and is used to build implementations that separate their behavior from their appearance using templates, much like most WPF controls. Deriving from [Control](#) allows you more freedom for creating a custom user interface than user controls, but it may require more effort.

- **Framework Element Model.**

A custom control derives from [FrameworkElement](#) when its appearance is defined by custom rendering logic (not templates).

For more information on custom controls, see [Control authoring overview](#).

See also

- [Tutorial: Create a new WPF app](#)
- [Migrate a WPF app to .NET Core](#)
- [Overview of WPF windows](#)
- [Data binding overview](#)
- [XAML overview](#)

Tutorial: Create a new WPF app (WPF .NET)

4/30/2021 • 11 minutes to read • [Edit Online](#)

In this short tutorial, you'll learn how to create a new Windows Presentation Foundation (WPF) app with Visual Studio. Once the initial app has been generated, you'll learn how to add controls and how to handle events. By the end of this tutorial, you'll have a simple app that adds names to a list box.

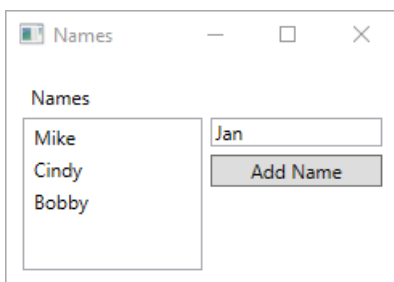
IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

In this tutorial, you learn how to:

- Create a new WPF app
- Add controls to a form
- Handle control events to provide app functionality
- Run the app

Here's a preview of the app you'll build while following this tutorial:



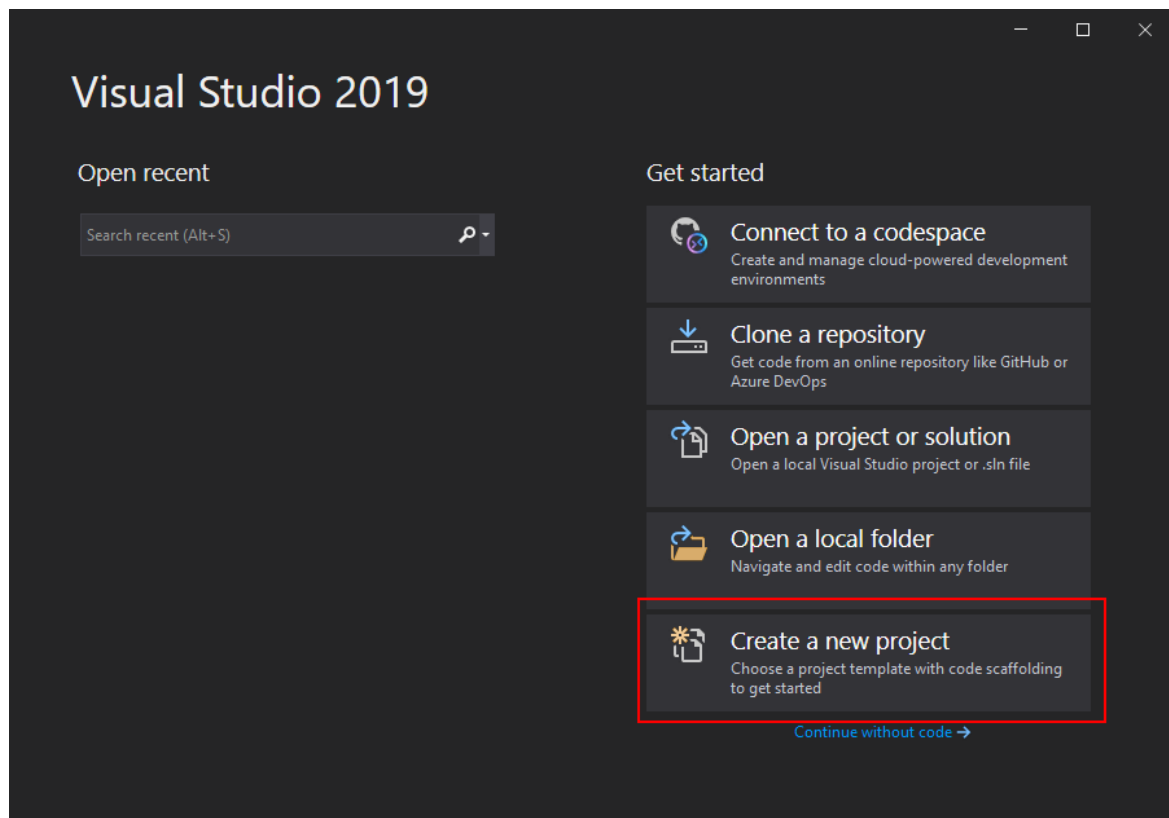
Prerequisites

- [Visual Studio 2019 version 16.9 or later versions](#)
 - Select the [Visual Studio Desktop workload](#)
 - Select the [.NET 5 individual component](#)

Create a WPF app

The first step to creating a new app is opening Visual Studio and generating the app from a template.

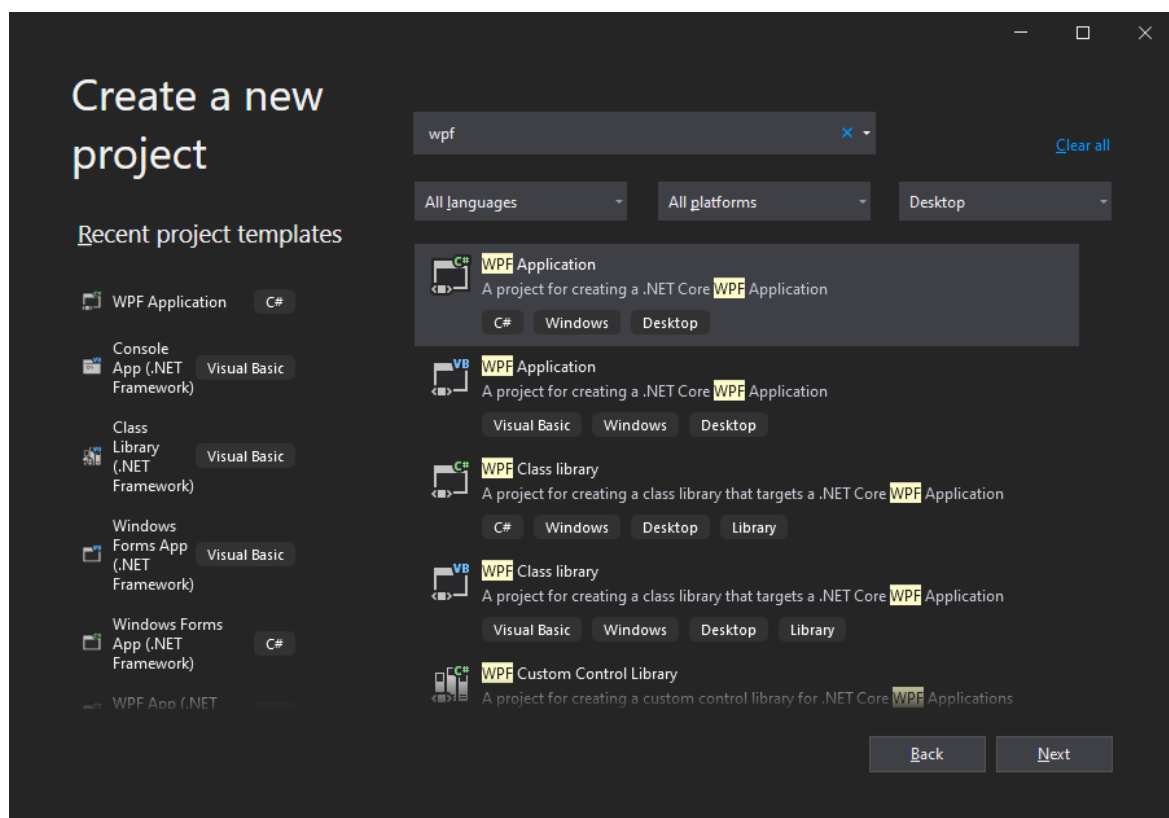
1. Open Visual Studio.
2. Select **Create a new project**.



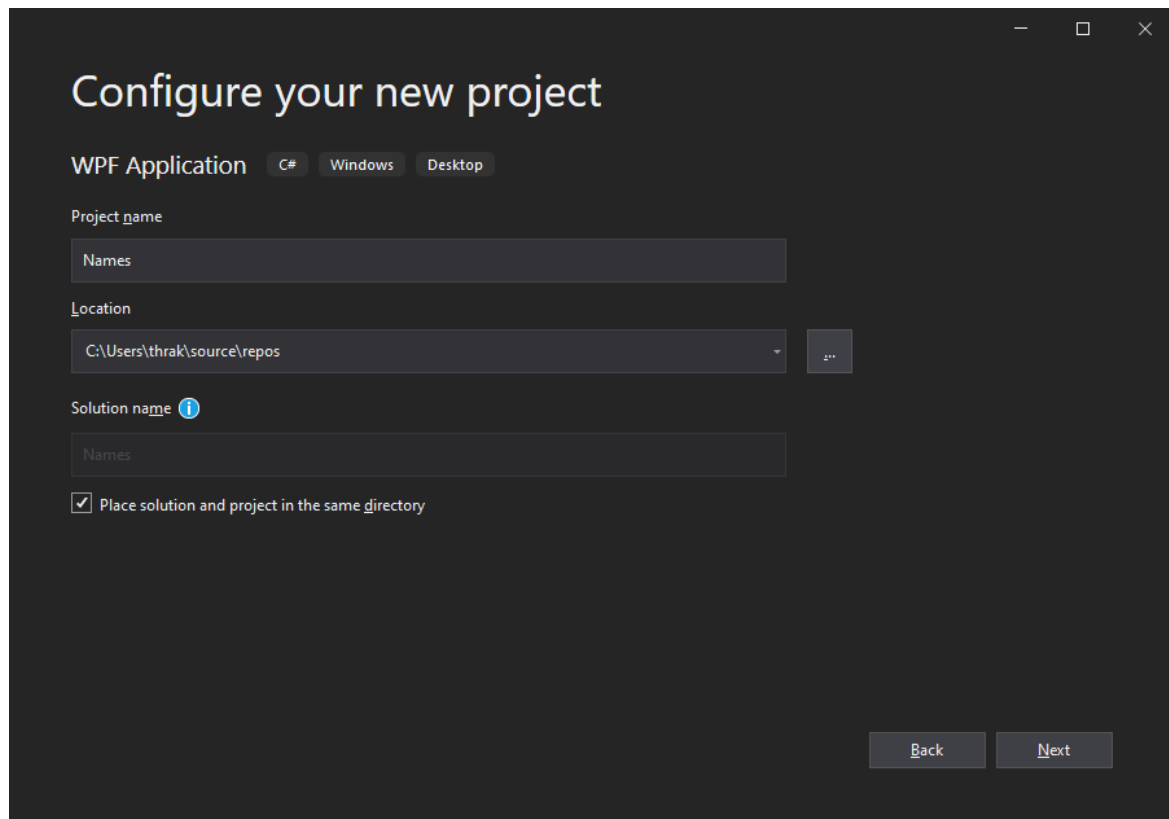
3. In the **Search for templates** box, type **wpf**, and then press Enter.
4. In the **code language** dropdown, choose **C#** or **Visual Basic**.
5. In the templates list, select **WPF Application** and then select **Next**.

IMPORTANT

Don't select the **WPF Application (.NET Framework)** template.

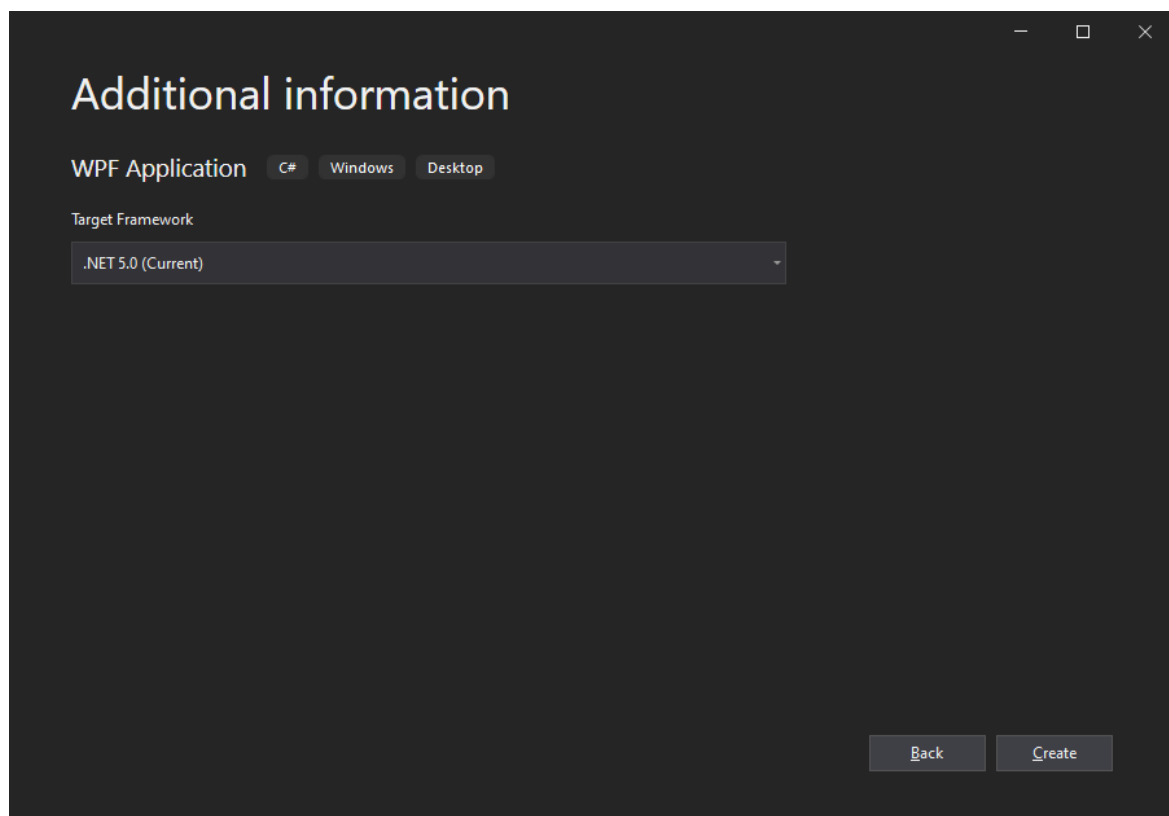


6. In the **Configure your new project** window, do the following:
 - a. In the **Project name** box, enter **Names**.
 - b. Select the **Place solution and project in the same directory** check box.
 - c. Optionally, choose a different **Location** to save your code.
 - d. Select the **Next** button.



The screenshot shows the 'Configure your new project' dialog box. At the top, it says 'WPF Application' with tabs for 'C#', 'Windows', and 'Desktop'. Below this, there are three input fields: 'Project name' with the text 'Names', 'Location' with the path 'C:\Users\thrak\source\repos', and 'Solution name' with the text 'Names'. A checkbox labeled 'Place solution and project in the same directory' is checked. At the bottom right, there are 'Back' and 'Next' buttons.

7. In the **Additional information** window, select **.NET 5.0 (Current)** for **Target Framework**. Select the **Create** button.



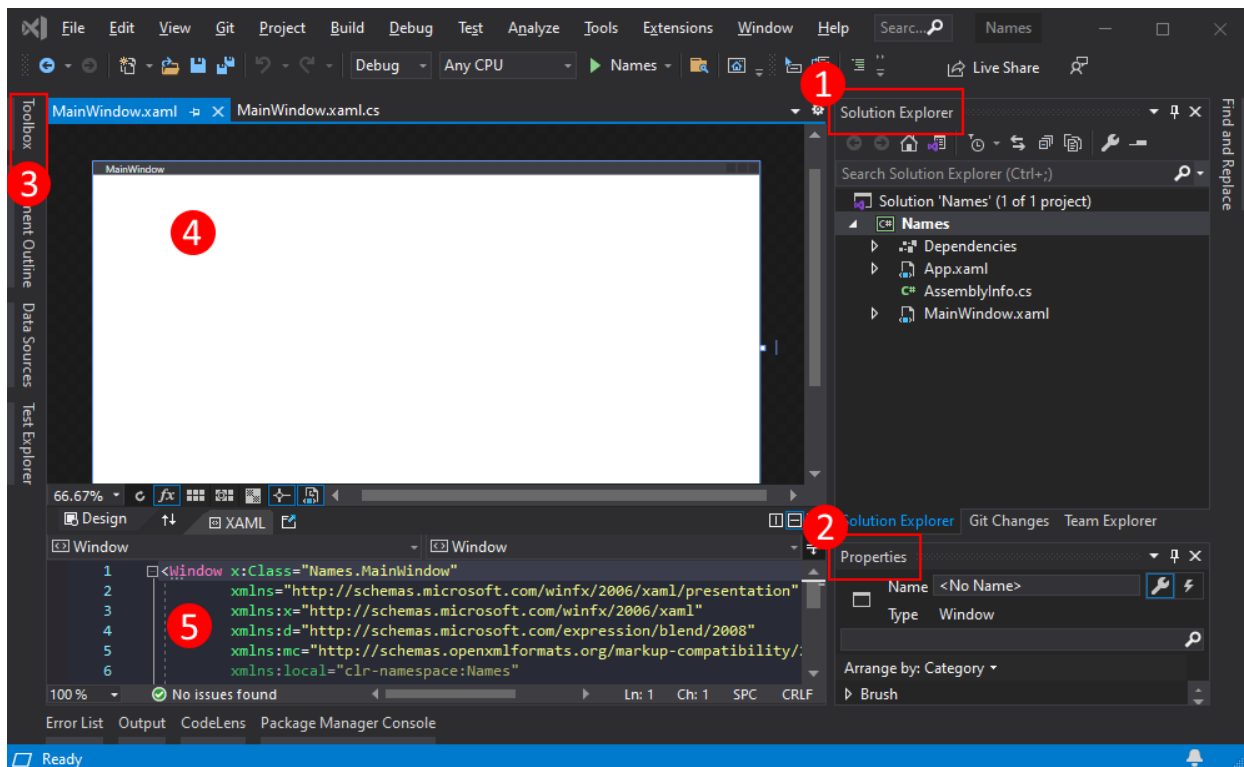
The screenshot shows the 'Additional information' dialog box. It has the same 'WPF Application' header and tabs as the previous window. Below, there is a 'Target Framework' dropdown menu currently set to '.NET 5.0 (Current)'. At the bottom right, there are 'Back' and 'Create' buttons.

Once the app is generated, Visual Studio should open the XAML designer pane for the default window,

MainWindow. If the designer isn't visible, double-click on the *MainWindow.xaml* file in the **Solution Explorer** pane to open the designer.

Important parts of Visual Studio

Support for WPF in Visual Studio has five important components that you'll interact with as you create an app:



1. Solution Explorer

All of your project files, code, windows, resources, will appear in this pane.

2. Properties

This pane shows property settings you can configure based on the item selected. For example, if you select an item from **Solution Explorer**, you'll see property settings related to the file. If you select an object in the **Designer**, you'll see settings for that item.

3. Toolbox

The toolbox contains all of the controls you can add to a form. To add a control to the current form, double-click a control or drag-and-drop the control.

4. XAML designer

This is the designer for a XAML document. It's interactive and you can drag-and-drop objects from the **Toolbox**. By selecting and moving items in the designer, you can visually compose the user interface (UI) for your app.

When both the designer and editor are visible, changes to one is reflected in the other. When you select items in the designer, the **Properties** pane displays the properties and attributes about that object.

5. XAML code editor

This is the XAML code editor for a XAML document. The XAML code editor is a way to craft your UI by hand without a designer. The designer may infer the values of properties on a control when the control is added in the designer. The XAML code editor gives you a lot more control.

When both the designer and editor are visible, changes to one is reflected in the other. As you navigate the text caret in the code editor, the **Properties** pane displays the properties and attributes about that

object.

Examine the XAML

After your project is created, the XAML code editor is visible with a minimal amount of XAML code to display the window. If the editor isn't open, double-click the *MainWindow.xaml* item in the **Solution Explorer**. You should see XAML similar to the following:

```
<Window x:Class="Names.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:Names"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Let's break down this XAML code to understand it better. XAML is simply XML that can be processed by the compilers that WPF uses. It describes the WPF UI and interacts with .NET code. To understand XAML, you should, at a minimum, be familiar with the basics of XML.

The document root `<Window>` represents the type of object being described by the XAML file. There are eight attributes declared, and they generally belong to three categories:

- Namespaces

An XML namespace provides structure to the XML, determining what is or isn't allowed to be declared in the file.

The main `xmlns` attribute imports the XML namespace for the entire file, and in this case, maps to the types declared by WPF. The other XML namespaces declare a prefix and import other types and objects for the XAML file. For example, the `xmlns:local` namespace declares the `local` prefix and maps to the objects declared by your project, the ones declared in the `Names` code namespace.

- `x:Class` attribute

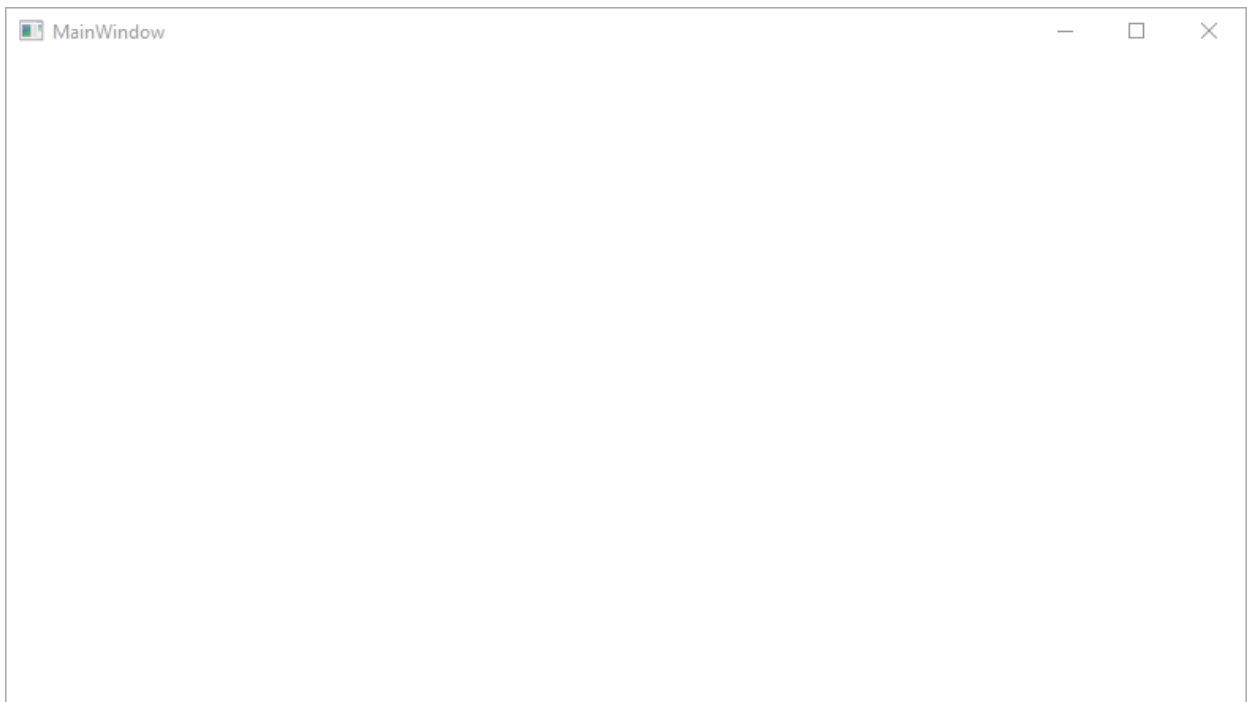
This attribute maps the `<Window>` to the type defined by your code: the *MainWindow.xaml.cs* or *MainWindow.xaml.vb* file, which is the `Names.MainWindow` class.

- `Title` attribute

Any normal attribute declared on the XAML object sets a property of that object. In this case the `Title` attribute sets the `Window.Title` property.

Change the window

First, let's run the project and see the default output. You'll see that a window that pops up, without any controls, and a title of **MainWindow**:



For our example app, this window is too large, and the title bar isn't descriptive. Change the title and size of the window by changing the appropriate attributes in the XAML to the following values:

```
<Window x:Class="Names.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:Names"
        mc:Ignorable="d"
        Title="Names" Height="180" Width="260">
    <Grid>

    </Grid>
</Window>
```

Prepare the layout

WPF provides a powerful layout system with many different layout controls. Layout controls help place and size child controls, and can even do so automatically. The default layout control provided to you in this XAML is the `<Grid>` control.

The `Grid` control lets you define rows and columns, much like a table, and place controls within the bounds of a specific row and column combination. You can have any number of child controls or other layout controls added to the `Grid`. For example, you can place another `Grid` control in a specific row and column combination, and that new `Grid` can then define more rows and columns and have its own children.

The `<Grid>` control defines rows and columns in which your controls will be. A grid always has a single row and column declared, meaning, the grid by default is a single cell. That doesn't really give you much flexibility in placing controls.

Before we add the new rows and columns, add a new attribute to the `<Grid>` element: `Margin="10"`. This insets the grid from the window and makes it look a little nicer.

Next, define two rows and two columns, dividing the grid into four cells:

```

<Window x:Class="Names.LayoutStep2"
    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:Names"
    mc:Ignorable="d"
    Title="Names" Height="180" Width="260">
    <Grid Margin="10">

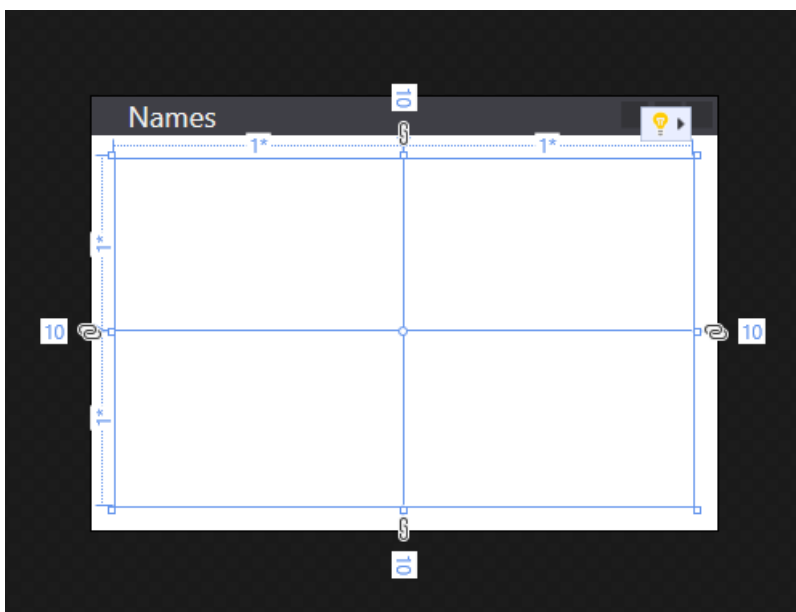
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

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

    </Grid>
</Window>

```

Select the grid in either the XAML code editor or XAML designer, you'll see that the XAML designer shows each row and column:



Add the first control

Now that the grid has been created, we can start adding controls to it. First, start with the label control. Create a new `<Label>` element inside the `<Grid>` element, after the row and column definitions, and give it a string value of `Names`:

```

<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

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

    <Label>Names</Label>

</Grid>

```

The `<Label>Names</Label>` defines the content `Names`. Some controls understand how to handle content, others don't. The content of a control maps to the `Content` property. Setting the content through XAML attribute syntax, you would use this format: `<Label Content="Names" />`. Both ways accomplish the same thing, setting the content of the label to display the text `Names`.

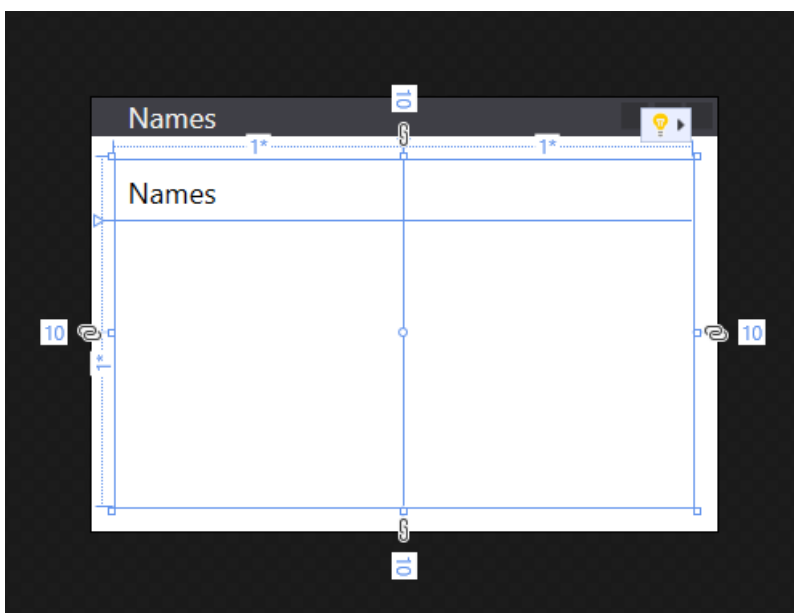
We have a problem though, the label takes up half the window as it was automatically assigned to the first row and column of the grid. For our first row, we don't need that much space because we're only going to use that row for the label. Change the `Height` attribute of the first `<RowDefinition>` from `*` to `Auto`. The `Auto` value automatically sizes the grid row to the size of its contents, in this case, the label control.

```

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

```

Notice that the designer now shows the label occupying a small amount of the available height. There's now more room for the next row to occupy. Most controls define some sort of height and width value that they should occupy that looks best for them. In the case of the label control, it has a height value that ensures that you can read it.



Control placement

Let's talk about control placement. The label created in the section above was automatically placed in row 0 and column 0 of the grid. The numbering for rows and columns starts at 0 and increments by 1 for each new row or

column. The control doesn't know anything about the grid, and the control doesn't define any properties to control its placement within the grid. The control could have even been placed within some other layout control which has its own set of rules defining how to place controls.

How do you tell a control to use a different row or column when the control has no knowledge of the grid? Attached properties! The grid takes advantage of the powerful property system provided by WPF. The grid defines new properties that the child controls can declare and use. The properties don't actually exist on the control itself, they're attached by the grid when the control is added to the grid.

The grid defines two properties to determine the row and column placement of a child control: `Grid.Row` and `Grid.Column`. If these properties are omitted from the control, it's implied that they have the default values of 0, so, the control is placed in row 0 and column 0 of the grid. Try changing the placement of the `<Label>` control by setting the `Grid.Column` attribute to 1:

```
<Label Grid.Column="1">Names</Label>
```

Notice how your label now moved to the second column. You can use the `Grid.Row` and `Grid.Column` attached properties to place the next controls we'll create. For now though, restore the label to row 0.

Create the name list box

Now that the grid is correctly sized and the label created, add a list box control on the row below the label. The list box will be in row 1 and column 0. We'll also give this control the name of `lstNames`. Once a control is named, it can be referenced in the code behind. The name is assigned to the control with the `x:Name` attribute.

```
<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

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

    <Label>Names</Label>
    <ListBox Grid.Row="1" x:Name="lstNames" />

</Grid>
```

Add the remaining controls

The last two controls we'll add are a text box and a button, which the user will use to enter a name to add to the list box. However, instead of trying to create more rows and columns for the grid, we'll put these controls into the `<StackPanel>` layout control.

The stack panel differs from the grid in how the controls are placed. While you tell the grid where you want the controls to be with the `Grid.Row` and `Grid.Column` attached properties, the stack panel works automatically by placing the first control, then placing the next control after it, continuing until all controls have been placed. It "stacks" each control below the other.

Create the `<StackPanel>` control after the list box and put it in grid row 1 column 1. Add another attribute named `Margin` with a value of `5,0,0,0`:


```

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

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

<Label>Names</Label>
<ListBox Grid.Row="1" x:Name="lstNames" />

<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">

</StackPanel>

```

The `Margin` attribute was previously used on the grid, but we only put in a single value, `10`. Now we've used a value of `5,0,0,0` on the stack panel. The margin is a `Thickness` type and can interpret both values. A thickness defines the space around each side of a rectangular frame, **left**, **top**, **right**, **bottom**, respectively. If the value for the margin is a single value, it uses that value for all four sides.

Next, create a `<TextBox>` and `<Button>` control in the `<StackPanel>`.

```

<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0">Add Name</Button>
</StackPanel>

```

The layout for the window is complete. However, our app doesn't have any logic in it to actually be functional. Next, we need to hook up the control events to code and get the app to actually do something.

Add code for the Click event

The `<Button>` we created has a `Click` event that is raised when the user presses the button. You can subscribe to this event and add code to add a name to the list box. Just like you set a property on a control by adding a XAML attribute, you can use a XAML attribute to subscribe to an event. Set the `Click` attribute to

`ButtonAddName_Click`

```

<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0" Click="ButtonAddName_Click">Add Name</Button>
</StackPanel>

```

Now you need to generate the handler code. Right-click on `ButtonAddName_Click` and select **Go To Definition**. This will generate a method in the code behind for you that matches the handler name you've entered.

```

private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{

}

```

```

Private Sub ButtonAddName_Click(sender As Object, e As RoutedEventArgs)

End Sub

```

Next, add the following code to do these three steps:

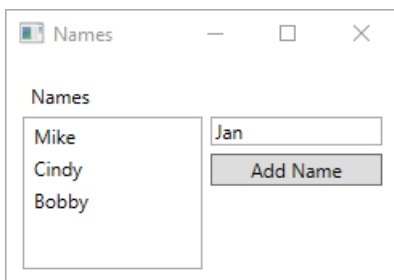
1. Make sure that the text box contains a name.
2. Validate that the name entered in the text box doesn't already exist.
3. Add the name to the list box.

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrEmpty(txtName.Text) && !lstNames.Items.Contains(txtName.Text))
        lstNames.Items.Add(txtName.Text);
}
```

```
Private Sub ButtonAddName_Click(sender As Object, e As RoutedEventArgs)
    If Not String.IsNullOrEmpty(txtName.Text) And Not lstNames.Items.Contains(txtName.Text) Then
        lstNames.Items.Add(txtName.Text)
    End If
End Sub
```

Run the app

Now that the event has been coded, you can run the app by pressing the F5 key or by selecting **Debug > Start Debugging** from the menu. The window is displayed and you can enter a name in the textbox and then add it by clicking the button.



Next steps

[Learn more about Windows Presentation Foundation](#)

Differences in WPF .NET

4/15/2021 • 2 minutes to read • [Edit Online](#)

This article describes the differences between Windows Presentation Foundation (WPF) on .NET 5 (or .NET Core 3.1) and .NET Framework. WPF for .NET 5 is an [open-source framework](#) forked from the original WPF for .NET Framework source code.

There are a few features of .NET Framework that .NET 5 doesn't support. For more information on unsupported technologies, see [.NET Framework technologies unavailable on .NET 5 and .NET Core](#).

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

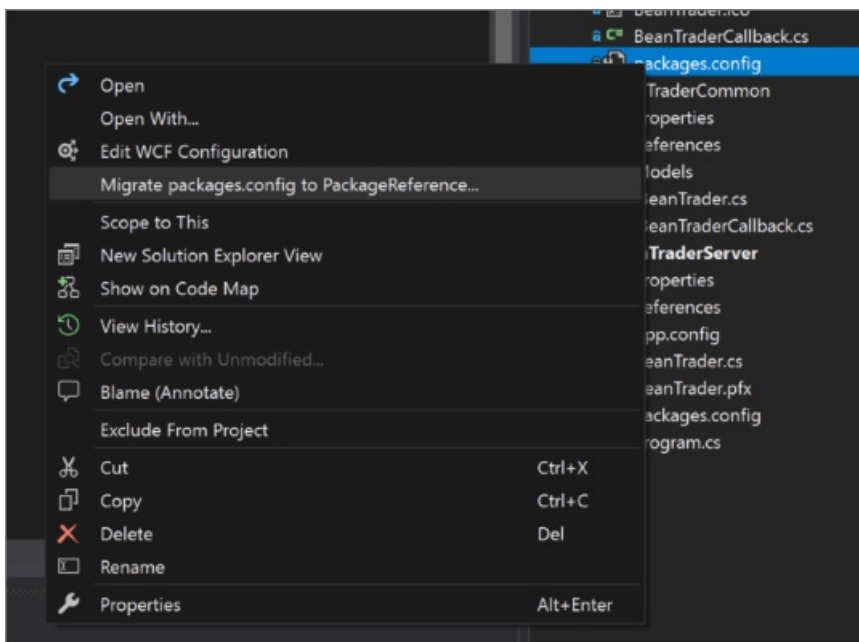
SDK-style projects

.NET 5 uses SDK-style project files. These project files are different from the traditional .NET Framework project files managed by Visual Studio. To migrate your .NET Framework WPF apps to .NET 5, you must convert your projects. For more information, see [Migrate WPF apps to .NET 5](#).

NuGet package references

If your .NET Framework app lists its NuGet dependencies in a *packages.config* file, migrate to the `<PackageReference>` format:

1. In Visual Studio, open the **Solution Explorer** pane.
2. In your WPF project, right-click *packages.config* > **Migrate packages.config to PackageReference**.



A dialog will appear showing calculated top-level NuGet dependencies and asking which other NuGet packages should be promoted to top level. Select **OK** and the *packages.config* file will be removed from the project and `<PackageReference>` elements will be added to the project file.

When your project uses `<PackageReference>`, packages aren't stored locally in a *Packages* folder, they're stored

globally. Open the project file and remove any `<Analyzer>` elements that referred to the *Packages* folder. These analyzers are automatically included with the NuGet package references.

Code Access Security

Code Access Security (CAS) is not supported by .NET 5. All CAS-related functionality is treated under the assumption of full-trust. WPF for .NET 5 removes CAS-related code. The public API surface of these types still exists to ensure that calls into these types succeed.

Publicly defined CAS-related types were moved out of the WPF assemblies and into the Core .NET library assemblies. The WPF assemblies have type-forwarding set to the new location of the moved types.

SOURCE ASSEMBLY	TARGET ASSEMBLY	TYPE
<i>WindowsBase.dll</i>	<i>System.Security.Permissions.dll</i>	MediaPermission MediaPermissionAttribute MediaPermissionAudio MediaPermissionImage MediaPermissionVideo WebBrowserPermission WebBrowserPermissionAttribute WebBrowserPermissionLevel
<i>System.Xaml.dll</i>	<i>System.Security.Permissions.dll</i>	XamlLoadPermission
<i>System.Xaml.dll</i>	<i>System.Windows.Extension.dll</i>	XamlAccessLevel

NOTE

In order to minimize porting friction, the functionality for storing and retrieving information related to the following properties was retained in the `XamlAccessLevel` type.

- `PrivateAccessToTypeName`
- `AssemblyNameString`

Next steps

- [Migrate WPF apps to .NET 5](#)

Migrating WPF apps to .NET Core

4/15/2021 • 26 minutes to read • [Edit Online](#)

This article covers the steps necessary to migrate a Windows Presentation Foundation (WPF) app from .NET Framework to .NET Core 3.0. If you don't have a WPF app on hand to port, but would like to try out the process, you can use the **Bean Trader** sample app available on [GitHub](#). The original app (targeting .NET Framework 4.7.2) is available in the NetFx\BeanTraderClient folder. First we'll explain the steps necessary to port apps in general, and then we'll walk through the specific changes that apply to the **Bean Trader** sample.

TIP

Try the [.NET Upgrade Assistant](#) to help migrate your own project.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

To migrate to .NET Core, you must first:

1. Understand and update NuGet dependencies:
 - a. Upgrade NuGet dependencies to use the `<PackageReference>` format.
 - b. Review top-level NuGet dependencies for .NET Core or .NET Standard compatibility.
 - c. Upgrade NuGet packages to newer versions.
 - d. Use the [.NET Portability Analyzer](#) to understand .NET dependencies.
2. Migrate the project file to the new SDK-style format:
 - a. Choose whether to target both .NET Core and .NET Framework, or only .NET Core.
 - b. Copy relevant project file properties and items to the new project file.
3. Fix build issues:
 - a. Add a reference to the [Microsoft.Windows.Compatibility](#) package.
 - b. Find and fix API-level differences.
 - c. Remove *app.config* sections other than `appSettings` or `connectionStrings`.
 - d. Regenerate generated code, if necessary.
4. Runtime testing:
 - a. Confirm the ported app works as expected.
 - b. Beware of [NotSupportedException](#) exceptions.

About the sample

This article references the [Bean Trader sample app](#) because it uses a variety of dependencies similar to those that real-world WPF apps might have. The app isn't large, but is meant to be a step up from 'Hello World' in terms of complexity. The app demonstrates some issues users may encounter while porting real apps. The app communicates with a WCF service, so for it to run properly, you'll also need to run the BeanTraderServer project (available in the same GitHub repository) and make sure the BeanTraderClient configuration points to the correct endpoint. (By default, the sample assumes the server is running on the same machine at

`http://localhost:8090`, which will be true if you launch BeanTraderServer locally.)

Keep in mind that this sample app is meant to demonstrate .NET Core porting challenges and solutions. It's not meant to demonstrate WPF best practices. In fact, it deliberately includes some anti-patterns to make sure you come across at least a couple of interesting challenges while porting.

Getting ready

The primary challenge of migrating a .NET Framework app to .NET Core is that its dependencies may work differently or not at all. Migration is much easier than it used to be; many NuGet packages now target .NET Standard. Starting with .NET Core 2.0, the .NET Framework and .NET Core surface areas have become similar. Even so, some differences (both in support from NuGet packages and in available .NET APIs) remain. The first step in migrating is to review the app's dependencies and make sure references are in a format that's easily migrated to .NET Core.

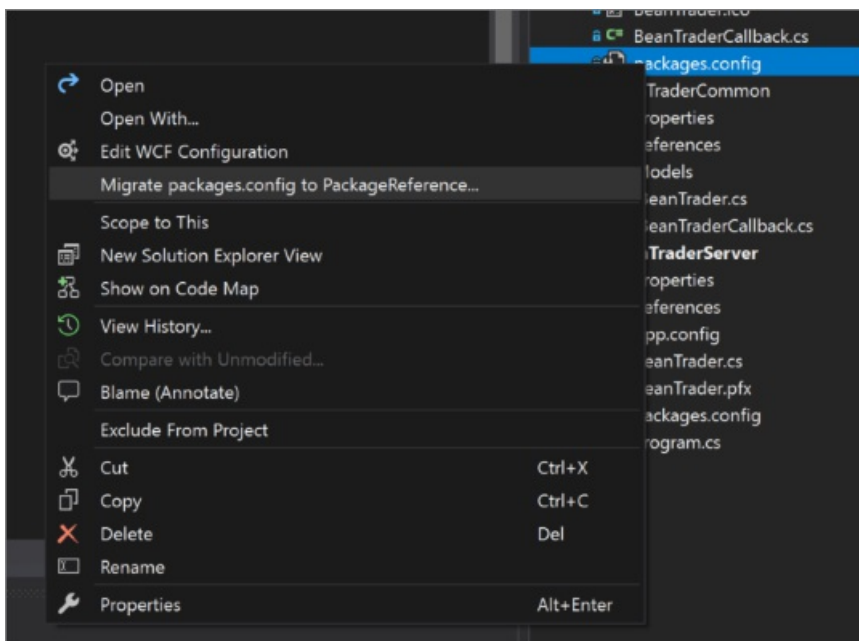
Upgrade to `<PackageReference>` NuGet references

Older .NET Framework projects typically list their NuGet dependencies in a *packages.config* file. The new SDK-style project file format references NuGet packages as `<PackageReference>` elements in the csproj file itself rather than in a separate config file.

When migrating, there are two advantages to using `<PackageReference>`-style references:

- This is the style of NuGet reference that is required for the new .NET Core project file. If you're already using `<PackageReference>`, those project file elements can be copied and pasted directly into the new project.
- Unlike a *packages.config* file, `<PackageReference>` elements only refer to the top-level dependencies that your project depends on directly. All other transitive NuGet packages will be determined at restore time and recorded in the autogenerated *obj\project.assets.json* file. This makes it much easier to determine what dependencies your project has, which is useful when determining whether the necessary dependencies will work on .NET Core or not.

The first step to migrating a .NET Framework app to .NET Core is to update it to use `<PackageReference>` NuGet references. Visual Studio makes this simple. Just right-click the project's *packages.config* file in Visual Studio's **Solution Explorer**, and then select **Migrate packages.config to PackageReference...**



A dialog appears showing calculated top-level NuGet dependencies and asking which other NuGet packages should be promoted to top-level. None of these other packages need to be top-level for the Bean Trader sample, so you can uncheck all of those boxes. Then, click **Ok** and the *packages.config* file is removed and

`<PackageReference>` elements are added to the project file.

`<PackageReference>`-style references don't store NuGet packages locally in a packages folder. Instead, they're stored globally as an optimization. After the migration completes, edit the csproj file and remove any `<Analyzer>` elements referring to the analyzers that previously came from the `..\packages` directory. Don't worry; since you still have the NuGet package references, the analyzers will be included in the project. You just need to clean up the old packages.config-style `<Analyzer>` elements.

Review NuGet packages

Now that you can see the top-level NuGet packages that the project depends on, you can review whether those packages are available on .NET Core. You can determine whether a package supports .NET Core by looking at its dependencies on nuget.org. The community-created fuget.org site shows this information prominently at the top of the package information page.

When targeting .NET Core 3.0, any packages targeting .NET Core or .NET Standard should work (since .NET Core implements the .NET Standard surface area). In some cases, the specific version of a package that's used won't target .NET Core or .NET Standard, but newer versions will. In this case, you should consider upgrading to the latest version of the package.

You can use packages targeting .NET Framework, as well, but that introduces some risk. .NET Core to .NET Framework dependencies are allowed because .NET Core and .NET Framework surface areas are similar enough that such dependencies *often* work. However, if the package tries to use a .NET API that isn't present in .NET Core, you'll encounter a runtime exception. Because of that, you should only reference .NET Framework packages when no other options are available and understand that doing so imposes a test burden.

If there are packages referenced that don't target .NET Core or .NET Standard, you'll have to think about other alternatives:

- Are there other similar packages that can be used instead? Sometimes NuGet authors publish separate '.Core' versions of their libraries specifically targeting .NET Core. Enterprise Library packages are an example of the community publishing ".NetCore" alternatives. In other cases, newer SDKs for a particular service (sometimes with different package names) are available for .NET Standard. If no alternatives are available, you can proceed using the .NET Framework-targeted packages, bearing in mind that you'll need to test them thoroughly once running on .NET Core.

The Bean Trader sample has the following top-level NuGet dependencies:

- [Castle.Windsor, version 4.1.1](#)

This package targets .NET Standard 1.6, so it works on .NET Core.

- [Microsoft.CodeAnalysis.FxCopAnalyzers, version 2.6.3](#)

This is a meta-package, so it's not immediately obvious which platforms it supports, but [documentation](#) indicates that its newest version (2.9.2) will work for both .NET Framework and .NET Core.

- [Nito.AsyncEx, version 4.0.1](#)

This package doesn't target .NET Core, but the newer 5.0 version does. This is common when migrating because many NuGet packages have added .NET Standard support recently, but older project versions will only target .NET Framework. If the version difference is only a minor version difference, it's often easy to upgrade to the newer version. Because this is a major version change, you need to be cautious upgrading since there could be breaking changes in the package. There is a path forward, though, which is good.

- [MahApps.Metro, version 1.6.5](#)

This package also doesn't target .NET Core, but has a newer pre-release (2.0-alpha) that does. Again, you have to look out for breaking changes, but the newer package is encouraging.

The Bean Trader sample's NuGet dependencies all either target .NET Standard/.NET Core or have newer versions that do, so there are unlikely to be any blocking issues here.

Upgrade NuGet packages

If possible, it would be good to upgrade the versions of any packages that only target .NET Core or .NET Standard with more recent versions at this point (with the project still targeting .NET Framework) to discover and address any breaking changes early.

If you would rather not make any material changes to the existing .NET Framework version of the app, this can wait until you have a new project file targeting .NET Core. However, upgrading the NuGet packages to .NET Core-compatible versions ahead of time makes the migration process even easier once you create the new project file and reduces the number of differences between the .NET Framework and .NET Core versions of the app.

With the Bean Trader sample, all of the necessary upgrades can be made easily (using Visual Studio's NuGet package manager) with one exception: upgrading from **MahApps.Metro 1.6.5** to **2.0** reveals breaking changes related to theme and accent management APIs.

Ideally, the app would be updated to use the newer version of the package (since that is more likely to work on .NET Core). In some cases, however, that may not be feasible. In these cases, don't upgrade **MahApps.Metro** because the necessary changes are non-trivial and this tutorial focuses on migrating to .NET Core 3, not to **MahApps.Metro 2**. Also, this is a low-risk .NET Framework dependency because the Bean Trader app only exercises a small part of **MahApps.Metro**. It will, of course, require testing to make sure everything's working once the migration is complete. If this were a real-world scenario, it would be good to file an issue to track the work to move to **MahApps.Metro** version 2.0 since not doing the migration now leaves behind some technical debt.

Once the NuGet packages are updated to recent versions, the `<PackageReference>` item group in the Bean Trader sample's project file should look like this.

```
<ItemGroup>
  <PackageReference Include="Castle.Windsor">
    <Version>4.1.1</Version>
  </PackageReference>
  <PackageReference Include="MahApps.Metro">
    <Version>1.6.5</Version>
  </PackageReference>
  <PackageReference Include="Microsoft.CodeAnalysis.FxCopAnalyzers">
    <Version>2.9.2</Version>
  </PackageReference>
  <PackageReference Include="Nito.AsyncEx">
    <Version>5.0.0</Version>
  </PackageReference>
</ItemGroup>
```

.NET Framework portability analysis

Once you understand the state of your project's NuGet dependencies, the next thing to consider is .NET Framework API dependencies. The [.NET Portability Analyzer](#) tool is useful for understanding which of the .NET APIs your project uses are available on other .NET platforms.

The tool comes as a [Visual Studio plugin](#), a [command-line tool](#), or wrapped in a [simple GUI](#), which simplifies its options. You can read more about using the .NET Portability Analyzer (API Port) using the GUI in the [Porting desktop apps to .NET Core](#) blog post. If you prefer to use the command line, the necessary steps are:

1. Download the [.NET Portability Analyzer](#) if you don't already have it.
2. Make sure the .NET Framework app to be ported builds successfully (this is a good idea prior to migration regardless).

3. Run API Port with a command line like this.

```
ApiPort.exe analyze -f <PathToBeanTraderBinaries> -r html -r excel -t ".NET Core"
```

The `-f` argument specifies the path containing the binaries to analyze. The `-r` argument specifies which output file format you want. The `-t` argument specifies which .NET platform to analyze API usage against. In this case, you want .NET Core.

When you open the HTML report, the first section will list all of the analyzed binaries and what percentage of the .NET APIs they use are available on the targeted platform. The percentage is not meaningful by itself. What's more useful is to see the specific APIs that are missing. To do that, either select an assembly name or scroll down to the reports for individual assemblies.

Focus on the assemblies that you own the source code for. In the Bean Trader ApiPort report, for example, there are many binaries listed, but most of them belong to NuGet packages. `Castle.Windsor` shows that it depends on some System.Web APIs that are missing in .NET Core. This isn't a concern because you previously verified that `Castle.Windsor` supports .NET Core. It is common for NuGet packages to have different binaries for use with different .NET platforms, so whether the .NET Framework version of `Castle.Windsor` uses System.Web APIs or not is irrelevant as long as the package also targets .NET Standard or .NET Core (which it does).

With the Bean Trader sample, the only binary that you need to consider is **BeanTraderClient** and the report shows that only two .NET APIs are missing: `System.ServiceModel.ClientBase<T>.Close` and `System.ServiceModel.ClientBase<T>.Open`.

BeanTraderClient, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null (.NETFramework,Version=v4.7.2)		
Target type	.NET Core,Version=v3.0	Recommended changes
System.ServiceModel.ClientBase`1	✓	
Close	✗	
Open	✗	

These are unlikely to be blocking issues because WCF Client APIs are (mostly) supported on .NET Core, so there must be alternatives available for these central APIs. In fact, looking at `System.ServiceModel`'s .NET Core surface area (using <https://apisof.net>), you see that there are async alternatives in .NET Core instead.

Based on this report and the previous NuGet dependency analysis, it looks like there should be no major issues migrating the Bean Trader sample to .NET Core. You're ready for the next step in which you'll actually start the migration.

Migrating the project file

If your app isn't using the new [SDK-style project file format](#), you'll need a new project file to target .NET Core. You can replace the existing csproj file or, if you prefer to keep the existing project untouched in its current state, you can add a new csproj file targeting .NET Core. You can build versions of the app for .NET Framework and .NET Core with a single SDK-style project file with [multi-targeting](#) (specifying multiple `<TargetFrameworks>` targets).

To create the new project file, you can create a new WPF project in Visual Studio or use the `dotnet new wpf` command in a temporary directory to generate the project file and then copy/rename it to the correct location. There is also a community-created tool, [CsprojToVs2017](#), that can automate some of the project file migration. The tool is helpful but still needs a human to review the results to make sure all the details of the migration are correct. One particular area that the tool doesn't handle optimally is migrating NuGet packages from *packages.config* files. If the tool runs on a project file that still uses a *packages.config* file to reference NuGet packages, it will migrate to `<PackageReference>` elements automatically, but will add `<PackageReference>` elements for *all* of the packages instead of just top-level ones. If you have already migrated to `<PackageReference>` elements with Visual Studio (as you've done in this sample), then the tool can help with the

rest of the conversion. Like Scott Hanselman recommends in [his blog post on migrating csproj files](#), porting by hand is educational and will give better results if you only have a few projects to port. But if you're porting dozens or hundreds of project files, then a tool like [CsprojToVs2017](#) can be a help.

To create a new project file for the Bean Trader sample, run `dotnet new wpf` in a temporary directory and move the generated `.csproj` file into the `BeanTraderClient` folder and rename it `BeanTraderClient.Core.csproj`.

Because the new project file format automatically includes C# files, `resx` files, and XAML files that it finds in or under its directory, the project file is already almost complete! To finish the migration, open the old and new project files side-by-side and look through the old one to see if any information it contains needs to be migrated. In the Bean Trader sample case, the following items should be copied to the new project:

- The `<RootNamespace>`, `<AssemblyName>`, and `<ApplicationIcon>` properties should all be copied.
- You also need to add a `<GenerateAssemblyInfo>false</GenerateAssemblyInfo>` property to the new project file since the Bean Trader sample includes assembly-level attributes (like `[AssemblyTitle]`) in an `AssemblyInfo.cs` file. By default, new SDK-style projects will autogenerate these attributes based on properties in the `csproj` file. Because you don't want that to happen in this case (the autogenerated attributes would conflict with those from `AssemblyInfo.cs`), you disable the autogenerated attributes with `<GenerateAssemblyInfo>`.
- Although `resx` files are automatically included as embedded resources, other `<Resource>` items like images are not. So, copy the `<Resource>` elements for embedding image and icon files. You can simplify the png references to a single line by using the new project file format's support for globbing patterns:
`<Resource Include="***.png" />`.
- Similarly, `<None>` items are included automatically, but they aren't copied to the output directory, by default. Because the Bean Trader project includes a `<None>` item that *is* copied to the output directory (using `PreserveNewest` behaviors), you need to update the automatically populated `<None>` item for that file, like this.

```
<None Update="BeanTrader.pfx">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</None>
```

- The Bean Trader sample includes a XAML file (`Default.Accent.xaml`) as `Content` (rather than as a `Page`) because themes and accents defined in this file are loaded from the file's XAML at runtime, rather than being embedded in the app itself. The new project system automatically includes this file as a `<Page>`, however, since it's a XAML file. So, you need to both remove the XAML file as a page (`<Page Remove="**\Default.Accent.xaml" />`) and add it as content.

```
<Content Include="Resources\Themes\Default.Accent.xaml">
  <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</Content>
```

- Finally, add NuGet references by copying the `<ItemGroup>` with all the `<PackageReference>` elements. If you hadn't previously upgraded the NuGet packages to .NET Core-compatible versions, you could do that now that the package references are in a .NET Core-specific project.

At this point, it should be possible to add the new project to the BeanTrader solution and open it in Visual Studio. The project should look correct in **Solution Explorer**, and `dotnet restore BeanTraderClient.Core.csproj` should successfully restore packages (with two expected warnings related to the MahApps.Metro version you're using targeting .NET Framework).

Although it's possible to keep both project files side-by-side (and may even be desirable if you want to keep

building the old project exactly as it was), it complicates the migration process (the two projects will try to use the same bin and obj folders) and usually isn't necessary. If you want to build for both .NET Core and .NET Framework targets, you can replace the `<TargetFramework>netcoreapp3.0</TargetFramework>` property in the new project file with `<TargetFrameworks>netcoreapp3.0;net472</TargetFrameworks>` instead. For the Bean Trader sample, delete the old project file (BeanTraderClient.csproj) since it's no longer needed. If you prefer to keep both project files, be sure to have them build to different output and intermediate output paths.

Fix build issues

The third step of the porting process is getting the project to build. Some apps will already build successfully once the project file is converted to an SDK-style project. If that's the case for your app, congratulations! You can go on to Step 4. Other apps will need some updates to get them building for .NET Core. If you try to run `dotnet build` on the Bean Trader sample project now, for example, (or build it in Visual Studio), there will be many errors, but you'll get them fixed quickly.

System.ServiceModel references and Microsoft.Windows.Compatibility

A common source of errors is missing references for APIs that are available for .NET Core but not automatically included in the .NET Core app metapackage. To address this, you should reference the `Microsoft.Windows.Compatibility` package. The compatibility package includes a broad set of APIs that are common in Windows desktop apps, such as WCF client, directory services, registry, configuration, ACLs APIs, and more.

With the Bean Trader sample, the majority of the build errors are due to missing `System.ServiceModel` types. These could be addressed by referencing the necessary WCF NuGet packages. WCF client APIs are among those present in the `Microsoft.Windows.Compatibility` package, though, so referencing the compatibility package is an even better solution (since it also addresses any issues related to APIs as well as solutions to the WCF issues that the compatibility package makes available). The `Microsoft.Windows.Compatibility` package helps in most .NET Core 3.0 WPF and WinForms porting scenarios. After adding the NuGet reference to `Microsoft.Windows.Compatibility`, only one build error remains!

Cleaning up unused files

One type of migration issue that comes up often relates to C# and XAML files that weren't previously included in the build getting picked up by the new SDK-style projects that include *all* source automatically.

The next build error you see in the Bean Trader sample refers to a bad interface implementation in `OldUnusedViewModel.cs`. The file name is a hint, but on inspection, you'll find that this source file is incorrect. It didn't cause issues previously because it wasn't included in the original .NET Framework project. Source files that were present on disk but not included in the old *csproj* are included automatically now.

For one-off issues like this, it's easy to compare to the previous *csproj* to confirm that the file isn't needed, and then either `<Compile Remove="" />` it or, if the source file isn't needed anywhere anymore, delete it. In this case, it's safe to just delete `OldUnusedViewModel.cs`.

If you have many source files that would need to be excluded this way, you can disable auto-inclusion of C# files by setting the `<EnableDefaultCompileItems>` property to false in the project file. Then, you can copy `<Compile Include>` items from the old project file to the new one in order to only build sources you intended to include. Similarly, `<EnableDefaultPageItems>` can be used to turn off auto-inclusion of XAML pages and `<EnableDefaultItems>` can control both with a single property.

A brief aside on multi-pass compilers

After removing the offending file from the Bean Trader sample, you can re-build and will get four errors. Didn't you have one before? Why did the number of errors go up? The C# compiler is a [multi-pass compiler](#). This means that it goes through each source file twice. First, the compiler just looks at metadata and declarations in each source file and identifies any declaration-level problems. Those are the errors you've fixed. Then it goes

through the code again to build the C# source into IL; those are the second set of errors that you're seeing now.

NOTE

The C# compiler does [more than just two passes](#), but the end result is that compiler errors for large code changes like this tend to come in two waves.

Third-party dependency fixes (Castle.Windsor)

Another class of issue that comes up in some migration scenarios is API differences between .NET Framework and .NET Core versions of dependencies. Even if a NuGet package targets both .NET Framework and .NET Standard or .NET Core, there may be different libraries for use with different .NET targets. This allows the packages to support many different .NET platforms, which may require different implementations. It also means that there may be small API differences in the libraries when targeting different .NET platforms.

The next set of errors you'll see in the Bean Trader sample are related to `Castle.Windsor` APIs. The .NET Core Bean Trader project uses the same version of `Castle.Windsor` as the .NET Framework-targeted project (4.1.1), but the implementations for those two platforms are slightly different.

In this case, you see the following issues that need to be fixed:

1. `Castle.MicroKernel.Registration.Classes.FromThisAssembly` isn't available on .NET Core. There is, however, the similar API `Classes.FromAssemblyContaining` available, so we can replace both uses of `Classes.FromThisAssembly()` with calls to `Classes.FromAssemblyContaining(t)`, where `t` is the type making the call.
2. Similarly, in *Bootstrapper.cs*, `Castle.Windsor.Installer.FromAssembly` is unavailable on .NET Core. Instead, that call can be replaced with `FromAssembly.Containing(typeof(Bootstrapper))`.

Updating WCF client usage

Having fixed the `Castle.Windsor` differences, the last remaining build error in the .NET Core Bean Trader project is that `BeanTraderServiceClient` (which derives from `DuplexClientBase`) doesn't have an `Open` method. This isn't surprising since this is an API that was highlighted by the .NET Portability Analyzer at the beginning of this migration process. Looking at `BeanTraderServiceClient` draws our attention to a larger issue, though. This WCF client was autogenerated by the [Svcutil.exe](#) tool.

WCF clients generated by Svcutil are meant for use on .NET Framework.

Solutions that use svcutil-generated WCF clients will need to regenerate .NET Standard-compatible clients for use with .NET Core. One of the main reasons the old clients won't work is that they depend on app configuration for defining WCF bindings and endpoints. Because .NET Standard WCF APIs can work cross-platform (where System.Configuration APIs aren't available), WCF clients for .NET Core and .NET Standard scenarios must define bindings and endpoints programmatically instead of in configuration.

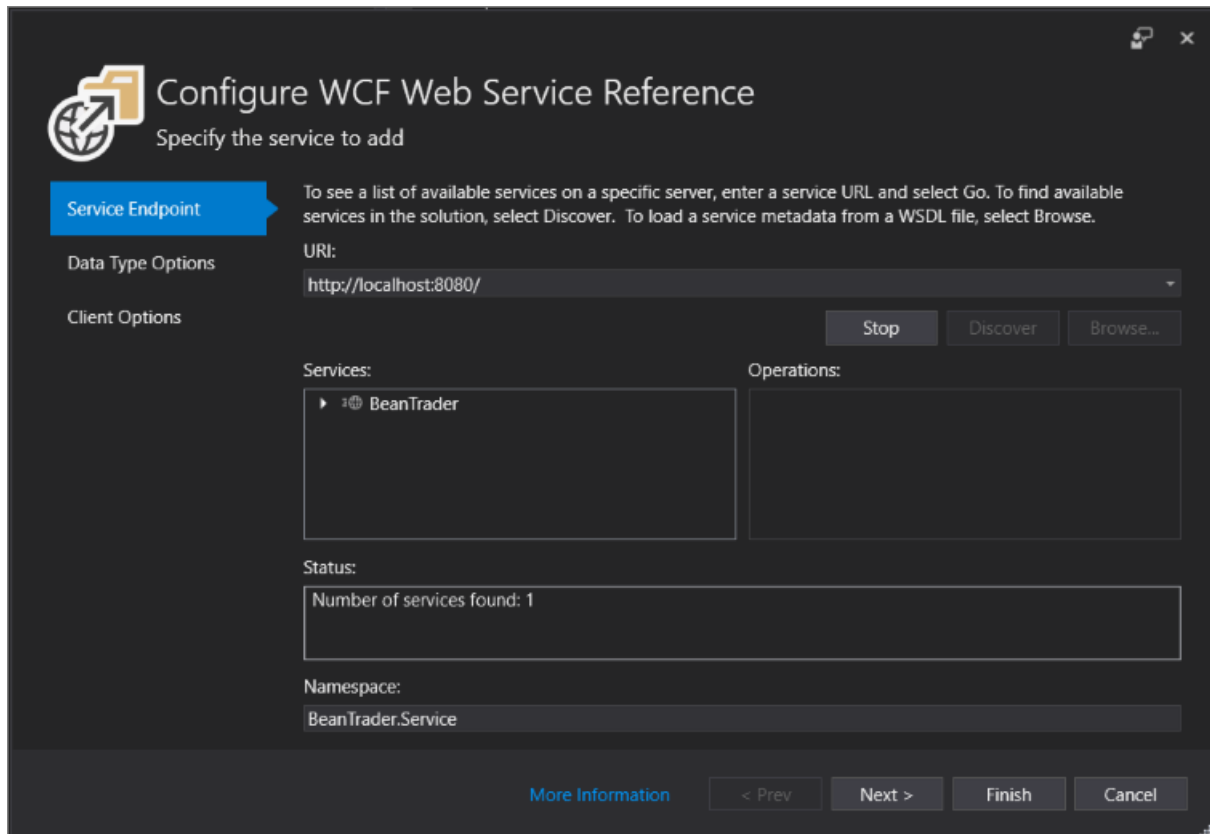
In fact, any WCF client usage that depends on the `<system.serviceModel>` app.config section (whether created with Svcutil or manually) will need to be changed to work on .NET Core.

There are two ways to automatically generate .NET Standard-compatible WCF clients:

- The `dotnet-svcutil` tool is a .NET tool that generates WCF clients in a way that is similar to how Svcutil worked previously.
- Visual Studio can generate WCF clients using the [WCF Web Service Reference](#) option of its Connected Services feature.

Either approach works well. Alternatively, of course, you could write the WCF client code yourself. For this sample, I chose to use the Visual Studio Connected Service feature. To do that, right-click on the *BeanTraderClient.Core* project in Visual Studio's solution explorer and select **Add > Connected Service**. Next,

choose the WCF Web Service Reference Provider. This will bring up a dialog where you can specify the address of the backend Bean Trader web service (`localhost:8080` if you are running the server locally) and the namespace that generated types should use (`BeanTrader.Service`, for example).



After you select the **Finish** button, a new Connected Services node is added to the project and a `Reference.cs` file is added under that node containing the new .NET Standard WCF client for accessing the Bean Trader service. If you look at the `GetEndpointAddress` or `GetBindingForEndpoint` methods in that file, you'll see that bindings and endpoints are now generated programmatically (instead of via app config). The 'Add Connected Services' feature may also add references to some `System.ServiceModel` packages in the project file, which aren't needed since all necessary WCF packages are included via `Microsoft.Windows.Compatibility`. Check the csproj to see if any extra `System.ServiceModel` `<PackageReference>` items have been added, and if so, remove them.

Our project has new WCF client classes now (in `Reference.cs`), but it also still has the old ones (in `BeanTrader.cs`). There are two options at this point:

- If you want to be able to build the original .NET Framework project (alongside the new .NET Core-targeted one), you can use a `<Compile Remove="BeanTrader.cs" />` item in the .NET Core project's csproj file so that the .NET Framework and .NET Core versions of the app use different WCF clients. This has the advantage of leaving the existing .NET Framework project unchanged, but has the disadvantage that code using the generated WCF clients may need to be slightly different in the .NET Core case than it was in the .NET Framework project, so you'll likely need to use `#if` directives to conditionally compile some WCF client usage (creating clients, for example) to work one way when built for .NET Core and another way when built for .NET Framework.
- If, on the other hand, some code churn in the existing .NET Framework project is acceptable, you can remove `BeanTrader.cs` all together. Because the new WCF client is built for .NET Standard, it will work in both .NET Core and .NET Framework scenarios. If you are building for .NET Framework in addition to .NET Core (either by multi-targeting or by having two csproj files), you can use this new `Reference.cs` file for both targets. This approach has the advantage that the code won't need to bifurcate to support two different WCF clients; the same code will be used everywhere. The drawback is that it involves changing the (presumably stable) .NET Framework project.

In the case of the Bean Trader sample, you can make small changes to the original project if it makes migration easier, so follow these steps to reconcile WCF client usage:

1. Add the new Reference.cs file to the .NET Framework *BeanTraderClient.csproj* project using the 'Add existing item' context menu from the solution explorer. Be sure to add 'as link' so that the same file is used by both projects (as opposed to copying the C# file). If you are building for both .NET Core and .NET Framework with a single csproj (using multi-targeting) then this step isn't necessary.
2. Delete *BeanTrader.cs*.
3. The new WCF client is similar to the old one, but a number of namespaces in the generated code are different. Because of this, it is necessary to update the project so that WCF client types are used from *BeanTrader.Service* (or whatever namespace name you chose) instead of *BeanTrader.Model* or without a namespace. Building *BeanTraderClient.Core.csproj* will help to identify where these changes need to be made. Fixes will be needed both in C# and in XAML source files.
4. Finally, you'll discover that there is an error in *BeanTraderServiceClientFactory.cs* because the available constructors for the `BeanTraderServiceClient` type have changed. It used to be possible to supply an `InstanceContext` argument (which was created using a `CallbackHandler` from the `Castle.Windsor` IoC container). The new constructors create new `CallbackHandler`s. There are, however, constructors in `BeanTraderServiceClient`'s base type that match what you want. Since the autogenerated WCF client code all exists in partial classes, you can easily extend it. To do this, create a new file called *BeanTraderServiceClient.cs* and then create a partial class with that same name (using the *BeanTrader.Service* namespace). Then, add one constructor to the partial type as shown here.

```
public BeanTraderServiceClient(System.ServiceModel.InstanceContext callbackInstance) :  
    base(callbackInstance, EndpointConfiguration.NetTcpBinding_BeanTraderService)  
{ }
```

With those changes made, the Bean Trader sample will now be using a new .NET Standard-compatible WCF client and you can make the final fix of changing the `Open` call in *TradingService.cs* to use `await OpenAsync` instead.

With the WCF issues addressed, the .NET Core version of the Bean Trader sample now builds cleanly!

Runtime testing

It's easy to forget that migration work isn't done as soon as the project builds cleanly against .NET Core. It's important to leave time for testing the ported app, too. Once things build successfully, make sure the app runs and works as expected, especially if you are using any packages targeting .NET Framework.

Let's try launching the ported Bean Trader app and see what happens. The app doesn't get far before failing with the following exception.

```
System.Configuration.ConfigurationErrorsException: 'Configuration system failed to initialize'  
  
Inner Exception  
ConfigurationErrorsException: Unrecognized configuration section system.serviceModel.
```

This makes sense, of course. Remember that WCF no longer uses app configuration, so the old `system.serviceModel` section of the `app.config` file needs to be removed. The updated WCF client includes all of the same information in its code, so the config section isn't needed anymore. If you wanted the WCF endpoint to be configurable in `app.config`, you could add it as an app setting and update the WCF client code to retrieve the WCF service endpoint from configuration.

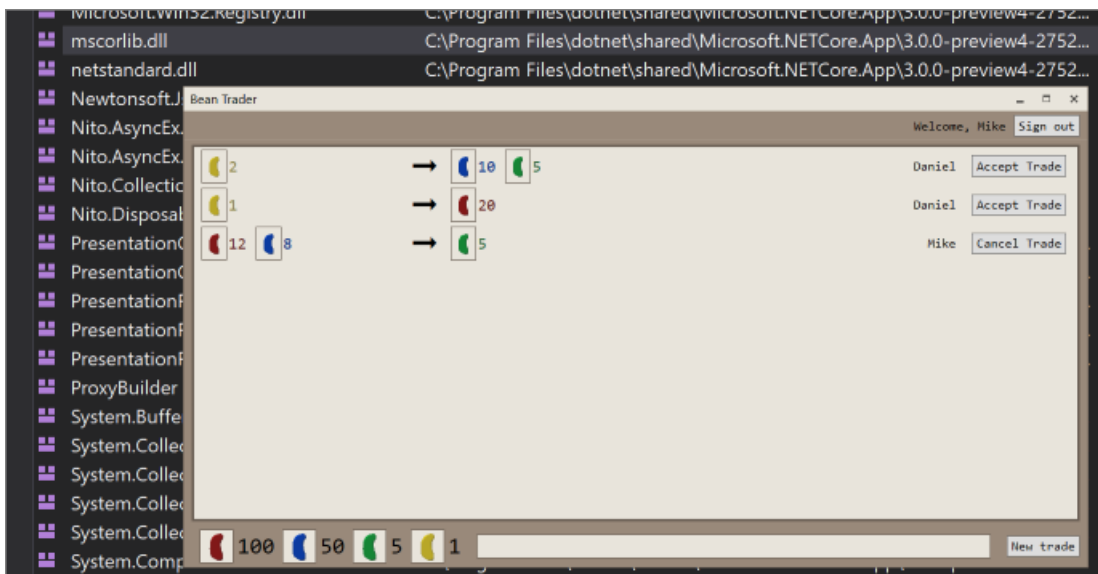
After removing the `system.serviceModel` section of `app.config`, the app launches but fails with another exception when a user signs in.

```
System.PlatformNotSupportedException: 'Operation is not supported on this platform.'
```

The unsupported API is `Func<T>.BeginInvoke`. As explained in [dotnet/corefx#5940](#), .NET Core doesn't support the `BeginInvoke` and `EndInvoke` methods on delegate types due to underlying remoting dependencies. This issue and its fix are explained in more detail in the [Migrating Delegate.BeginInvoke Calls for .NET Core](#) blog post, but the gist is that `BeginInvoke` and `EndInvoke` calls should be replaced with `Task.Run` (or async alternatives, if possible). Applying the general solution here, the `BeginInvoke` call can be replaced with an `Invoke` call launched by `Task.Run`.

```
Task.Run(() =>
{
    return userInfoRetriever.Invoke();
}).ContinueWith(result =>
{
    // BeginInvoke's callback is replaced with ContinueWith
    var task = result.ConfigureAwait(false);
    CurrentTrader = task.GetAwaiter().GetResult();
}, TaskScheduler.Default);
```

After removing the `BeginInvoke` usage, the Bean Trader app runs successfully on .NET Core!



All apps are different, so the specific steps needed to migrate your own apps to .NET Core will vary. But hopefully the Bean Trader sample demonstrates the general workflow and the types of issues that can be expected. And, despite this article's length, the actual changes needed in the Bean Trader sample to make it work on .NET Core were fairly limited. Many apps migrate to .NET Core in this same way; with limited or even no code changes needed.

Overview of WPF windows (WPF .NET)

4/15/2021 • 22 minutes to read • [Edit Online](#)

Users interact with Windows Presentation Foundation (WPF) applications through windows. The primary purpose of a window is to host content that visualizes data and enables users to interact with data. WPF applications provide their own windows by using the [Window](#) class. This article introduces [Window](#) before covering the fundamentals of creating and managing windows in applications.

IMPORTANT

This article uses XAML generated from a C# project. If you're using Visual Basic, the XAML may look slightly different. These differences are typically present on `x:Class` attribute values. C# includes the root namespace for the project while Visual Basic doesn't.

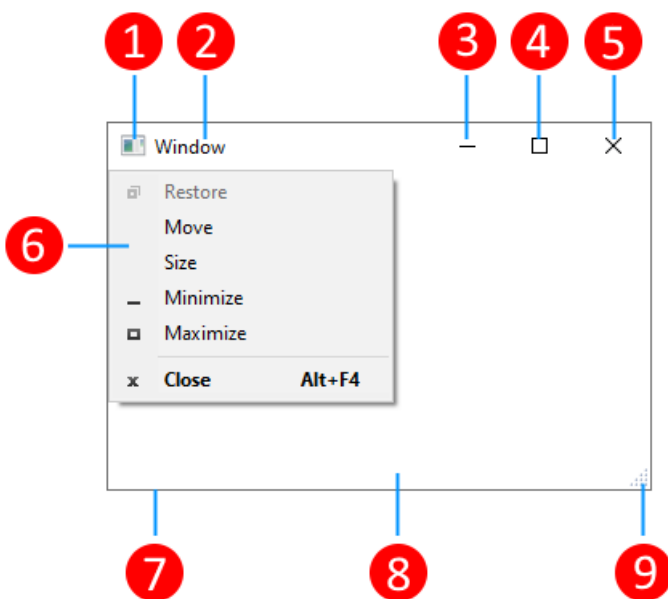
The project templates for C# create an `App` type contained in the `app.xaml` file. In Visual Basic, the type is named `Application` and the file is named `Application.xaml`.

The Window class

In WPF, a window is encapsulated by the [Window](#) class that you use to do the following:

- Display a window.
- Configure the size, position, and appearance of a window.
- Host application-specific content.
- Manage the lifetime of a window.

The following figure illustrates the constituent parts of a window:



A window is divided into two areas: the non-client area and client area.

The *non-client area* of a window is implemented by WPF and includes the parts of a window that are common to most windows, including the following:

- A title bar (1-5).

- An icon (1).
- Title (2).
- Minimize (3), Maximize (4), and Close (5) buttons.
- System menu (6) with menu items. Appears when clicking on the icon (1).
- Border (7).

The *client area* of a window is the area within a window's non-client area and is used by developers to add application-specific content, such as menu bars, tool bars, and controls.

- Client area (8).
- Resize grip (9). This is a control added to the Client area (8).

Implementing a window

The implementation of a typical window includes both appearance and behavior, where *appearance* defines how a window looks to users and *behavior* defines the way a window functions as users interact with it. In WPF, you can implement the appearance and behavior of a window using either code or XAML markup.

In general, however, the appearance of a window is implemented using XAML markup, and its behavior is implemented using code-behind, as shown in the following example.

```
<Window x:Class="WindowsOverview.Window1"
        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:WindowsOverview"
        >

    <!-- Client area containing the content of the window -->

</Window>
```

The following code is the code-behind for the XAML.

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

```
Public Class Window1

End Class
```

To enable a XAML markup file and code-behind file to work together, the following are required:

- In markup, the `Window` element must include the `x:Class` attribute. When the application is built, the existence of `x:Class` attribute causes Microsoft build engine (MSBuild) to generate a `partial` class that derives from `Window` with the name that is specified by the `x:Class` attribute. This requires the addition

of an XML namespace declaration for the XAML schema (

`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`). The generated `partial` class implements the `InitializeComponent` method, which is called to register the events and set the properties that are implemented in markup.

- In code-behind, the class must be a `partial` class with the same name that is specified by the `x:Class` attribute in markup, and it must derive from `Window`. This allows the code-behind file to be associated with the `partial` class that is generated for the markup file when the application is built, for more information, see [Compile a WPF Application](#).
- In code-behind, the `Window` class must implement a constructor that calls the `InitializeComponent` method. `InitializeComponent` is implemented by the markup file's generated `partial` class to register events and set properties that are defined in markup.

NOTE

When you add a new `Window` to your project by using Visual Studio, the `Window` is implemented using both markup and code-behind, and includes the necessary configuration to create the association between the markup and code-behind files as described here.

With this configuration in place, you can focus on defining the appearance of the window in XAML markup and implementing its behavior in code-behind. The following example shows a window with a button that defines an event handler for the `Click` event. This is implemented in the XAML and the handler is implemented in code-behind.

```
<Window x:Class="WindowsOverview.Final"
        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:WindowsOverview"
>

    <!-- Client area containing the content of the window -->

    <Button Click="Button_Click">Click This Button</Button>

</Window>
```

The following code is the code-behind for the XAML.

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Button was clicked.");
        }
    }
}
```

```
Public Class Window1

    Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
        MessageBox.Show("Button was clicked.")
    End Sub

End Class
```

Configuring a window for MSBuild

How you implement your window determines how it's configured for MSBuild. For a window that is defined using both XAML markup and code-behind:

- XAML markup files are configured as MSBuild `Page` items.
- Code-behind files are configured as MSBuild `Compile` items.

.NET SDK projects automatically import the correct `Page` and `Compile` items for you and you don't need to declare these. When the project is configured for WPF, the XAML markup files are automatically imported as `Page` items, and the corresponding code-behind file is imported as `Compile`.

MSBuild projects won't automatically import the types and you must declare them yourself:

```
<Project>
  ...
  <Page Include="MarkupAndCodeBehindWindow.xaml" />
  <Compile Include=" MarkupAndCodeBehindWindow.xaml.cs" />
  ...
</Project>
```

For information about building WPF applications, see [Compile a WPF Application](#).

Window lifetime

As with any class, a window has a lifetime that begins when it's first instantiated, after which it's opened, activated/deactivated, and eventually closed.

Opening a window

To open a window, you first create an instance of it, which is demonstrated in the following example:

```
using System.Windows;

namespace WindowsOverview
{
    public partial class App : Application
    {
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            // Create the window
            Window1 window = new Window1();

            // Open the window
            window.Show();
        }
    }
}
```

```

Class Application

    Private Sub Application_Startup(sender As Object, e As StartupEventArgs)
        ' Create the window
        Dim window As New Window1

        ' Open the window
        window.Show()
    End Sub

End Class

```

In this example `Window1` is instantiated when the application starts, which occurs when the [Startup](#) event is raised. For more information about the startup window, see [How to get or set the main application window](#).

When a window is instantiated, a reference to it's automatically added to a [list of windows](#) that is managed by the [Application](#) object. The first window to be instantiated is automatically set by [Application](#) as the [main application window](#).

The window is finally opened by calling the [Show](#) method as shown in the following image:



A window that is opened by calling [Show](#) is a *modeless* window, and the application doesn't prevent users from interacting with other windows in the application. Opening a window with [ShowDialog](#) opens a window as *modal* and restricts user interaction to the specific window. For more information, see [Dialog Boxes Overview](#).

When [Show](#) is called, a window does initialization work before it's shown to establish infrastructure that allows it to receive user input. When the window is initialized, the [SourceInitialized](#) event is raised and the window is shown.

For more information, see [How to open a window or dialog box](#).

Startup window

The previous example used the `Startup` event to run code that displayed the initial application window. As a shortcut, instead use [StartupUri](#) to specify the path to a XAML file in your application. The application automatically creates and displays the window specified by that property.

```

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

    </Application.Resources>
</Application>

```

Window ownership

A window that is opened by using the [Show](#) method doesn't have an implicit relationship with the window that created it. Users can interact with either window independently of the other, which means that either window can do the following:

- Cover the other (unless one of the windows has its [Topmost](#) property set to `true`).
- Be minimized, maximized, and restored without affecting the other.

Some windows require a relationship with the window that opens them. For example, an Integrated Development Environment (IDE) application may open property windows and tool windows whose typical behavior is to cover the window that creates them. Furthermore, such windows should always close, minimize, maximize, and restore in concert with the window that created them. Such a relationship can be established by making one window *own* another, and is achieved by setting the [Owner](#) property of the *owned window* with a reference to the *owner window*. This is shown in the following example.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create a window and make the current window its owner
    var ownedWindow = new ChildWindow1();
    ownedWindow.Owner = this;
    ownedWindow.Show();
}
```

```
Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
    ' Create a window and make the current window its owner
    Dim ownedWindow As New ChildWindow1
    ownedWindow.Owner = Me
    ownedWindow.Show()
End Sub
```

After ownership is established:

- The owned window can reference its owner window by inspecting the value of its [Owner](#) property.
- The owner window can discover all the windows it owns by inspecting the value of its [OwnedWindows](#) property.

Window activation

When a window is first opened, it becomes the active window. The *active window* is the window that is currently capturing user input, such as key strokes and mouse clicks. When a window becomes active, it raises the [Activated](#) event.

NOTE

When a window is first opened, the [Loaded](#) and [ContentRendered](#) events are raised only after the [Activated](#) event is raised. With this in mind, a window can effectively be considered opened when [ContentRendered](#) is raised.

After a window becomes active, a user can activate another window in the same application, or activate another application. When that happens, the currently active window becomes deactivated and raises the [Deactivated](#) event. Likewise, when the user selects a currently deactivated window, the window becomes active again and [Activated](#) is raised.

One common reason to handle [Activated](#) and [Deactivated](#) is to enable and disable functionality that can only run when a window is active. For example, some windows display interactive content that requires constant user input or attention, including games and video players. The following example is a simplified video player that demonstrates how to handle [Activated](#) and [Deactivated](#) to implement this behavior.

```

<Window x:Class="WindowsOverview.CustomMediaPlayerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Activated="Window_Activated"
        Deactivated="Window_Deactivated"
        Title="CustomMediaPlayerWindow" Height="450" Width="800">
    <Grid>
        <MediaElement x:Name="mediaElement" Stretch="Fill"
                        LoadedBehavior="Manual" Source="numbers.mp4" />
    </Grid>
</Window>

```

The following code is the code-behind for the XAML.

```

using System;
using System.Windows;

namespace WindowsOverview
{
    public partial class CustomMediaPlayerWindow : Window
    {
        public CustomMediaPlayerWindow() =>
            InitializeComponent();

        private void Window_Activated(object sender, EventArgs e)
        {
            // Continue playing media if window is activated
            mediaElement.Play();
        }

        private void Window_Deactivated(object sender, EventArgs e)
        {
            // Pause playing if media is being played and window is deactivated
            mediaElement.Pause();
        }
    }
}

```

```

Public Class CustomMediaPlayerWindow
    Private Sub Window_Activated(sender As Object, e As EventArgs)
        ' Continue playing media if window is activated
        mediaElement.Play()
    End Sub

    Private Sub Window_Deactivated(sender As Object, e As EventArgs)
        ' Pause playing if media is being played and window is deactivated
        mediaElement.Pause()
    End Sub
End Class

```

Other types of applications may still run code in the background when a window is deactivated. For example, a mail client may continue polling the mail server while the user is using other applications. Applications like these often provide different or extra behavior while the main window is deactivated. For a mail program, this may mean both adding the new mail item to the inbox and adding a notification icon to the system tray. A notification icon need only be displayed when the mail window isn't active, which is determined by inspecting the [IsActive](#) property.

If a background task completes, a window may want to notify the user more urgently by calling [Activate](#) method. If the user is interacting with another application activated when [Activate](#) is called, the window's taskbar button flashes. However, if a user is interacting with the current application, calling [Activate](#) will bring the

window to the foreground.

NOTE

You can handle application-scope activation using the [Application.Activated](#) and [Application.Deactivated](#) events.

Preventing window activation

There are scenarios where windows shouldn't be activated when shown, such as conversation windows of a chat application or notification windows of an email application.

If your application has a window that shouldn't be activated when shown, you can set its [ShowActivated](#) property to `false` before calling the [Show](#) method for the first time. As a consequence:

- The window isn't activated.
- The window's [Activated](#) event isn't raised.
- The currently activated window remains activated.

The window will become activated, however, as soon as the user activates it by clicking either the client or non-client area. In this case:

- The window is activated.
- The window's [Activated](#) event is raised.
- The previously activated window is deactivated.
- The window's [Deactivated](#) and [Activated](#) events are then raised as expected in response to user actions.

Closing a window

The life of a window starts coming to an end when a user closes it. Once a window is closed, it can't be reopened. A window can be closed by using elements in the non-client area, including the following:

- The **Close** item of the **System** menu.
- Pressing ALT + F4.
- Pressing the **Close** button.
- Pressing ESC when a button has the [IsCancel](#) property set to `true` on a modal window.

You can provide more mechanisms to the client area to close a window, the more common of which include the following:

- An **Exit** item in the **File** menu, typically for main application windows.
- A **Close** item in the **File** menu, typically on a secondary application window.
- A **Cancel** button, typically on a modal dialog box.
- A **Close** button, typically on a modeless dialog box.

To close a window in response to one of these custom mechanisms, you need to call the [Close](#) method. The following example implements the ability to close a window by choosing **Exit** from a **File** menu.

```
<Window x:Class="WindowsOverview.ClosingWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ClosingWindow" Height="450" Width="800">
    <StackPanel>
        <Menu>
            <MenuItem Header="_File">
                <MenuItem Header="E_xit" Click="fileExitMenuItem_Click" />
            </MenuItem>
        </Menu>
    </StackPanel>
</Window>
```

The following code is the code-behind for the XAML.

```
using System.Windows;

namespace WindowsOverview
{
    public partial class ClosingWindow : Window
    {
        public ClosingWindow() =>
            InitializeComponent();

        private void fileExitMenuItem_Click(object sender, RoutedEventArgs e)
        {
            // Close the current window
            this.Close();
        }
    }
}
```

```
Public Class ClosingWindow
    Private Sub fileExitMenuItem_Click(sender As Object, e As RoutedEventArgs)
        ' Close the current window
        Me.Close()
    End Sub
End Class
```

NOTE

An application can be configured to shut down automatically when either the main application window closes (see [MainWindow](#)) or the last window closes. For more information, see [ShutdownMode](#).

While a window can be explicitly closed through mechanisms provided in the non-client and client areas, a window can also be implicitly closed as a result of behavior in other parts of the application or Windows, including the following:

- A user logs off or shuts down Windows.
- A window's [Owner](#) closes.
- The main application window is closed and [ShutdownMode](#) is [OnMainWindowClose](#).
- [Shutdown](#) is called.

IMPORTANT

A window can't be reopened after it's closed.

Cancel window closure

When a window closes, it raises two events: [Closing](#) and [Closed](#).

[Closing](#) is raised before the window closes, and it provides a mechanism by which window closure can be prevented. One common reason to prevent window closure is if window content contains modified data. In this situation, the [Closing](#) event can be handled to determine whether data is dirty and, if so, to ask the user whether to either continue closing the window without saving the data or to cancel window closure. The following example shows the key aspects of handling [Closing](#).

```
<Window x:Class="WindowsOverview.DataWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataWindow" Height="450" Width="800"
        Closing="Window_Closing">
    <Grid>
        <TextBox x:Name="documentTextBox" TextChanged="documentTextBox_TextChanged" />
    </Grid>
</Window>
```

The following code is the code-behind for the XAML.

```
using System.Windows;
using System.Windows.Controls;

namespace WindowsOverview
{
    public partial class DataWindow : Window
    {
        private bool _isDataDirty;

        public DataWindow() =>
            InitializeComponent();

        private void documentTextBox_TextChanged(object sender, TextChangedEventArgs e) =>
            _isDataDirty = true;

        private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
        {
            // If data is dirty, prompt user and ask for a response
            if (_isDataDirty)
            {
                var result = MessageBox.Show("Document has changed. Close without saving?",
                                             "Question",
                                             MessageBoxButton.YesNo);

                // User doesn't want to close, cancel closure
                if (result == MessageBoxResult.No)
                    e.Cancel = true;
            }
        }
    }
}
```

```

Public Class DataWindow

    Private _isDataDirty As Boolean

    Private Sub documentTextBox_TextChanged(sender As Object, e As TextChangedEventArgs)
        _isDataDirty = True
    End Sub

    Private Sub Window_Closing(sender As Object, e As ComponentModel.CancelEventArgs)

        ' If data is dirty, prompt user and ask for a response
        If _isDataDirty Then
            Dim result = MessageBox.Show("Document has changed. Close without saving?",
                                         "Question",
                                         MessageBoxButton.YesNo)

            ' User doesn't want to close, cancel closure
            If result = MessageBoxResult.No Then
                e.Cancel = True
            End If
        End If

    End Sub

End Class

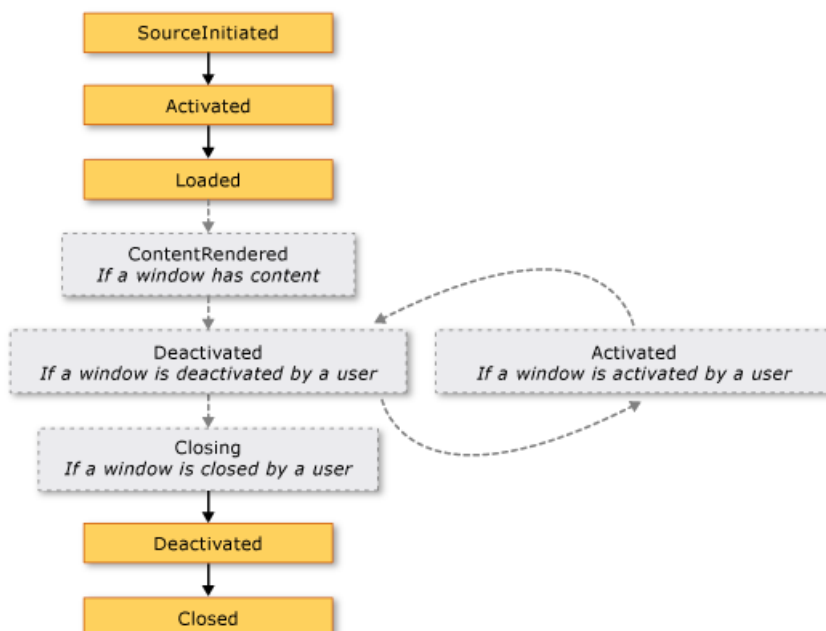
```

The **Closing** event handler is passed a **CancelEventArgs**, which implements the **Cancel** property that you set to `true` to prevent a window from closing.

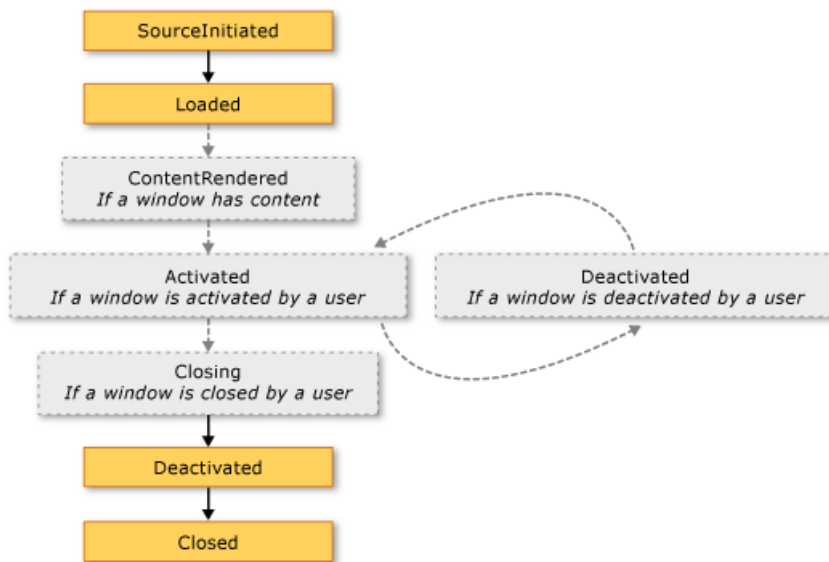
If **Closing** isn't handled, or it's handled but not canceled, the window will close. Just before a window actually closes, **Closed** is raised. At this point, a window can't be prevented from closing.

Window lifetime events

The following illustration shows the sequence of the principal events in the lifetime of a window:



The following illustration shows the sequence of the principal events in the lifetime of a window that is shown without activation (**ShowActivated** is set to `false` before the window is shown):



Window location

While a window is open, it has a location in the x and y dimensions relative to the desktop. This location can be determined by inspecting the [Left](#) and [Top](#) properties, respectively. Set these properties to change the location of the window.

You can also specify the initial location of a [Window](#) when it first appears by setting the [WindowStartupLocation](#) property with one of the following [WindowStartupLocation](#) enumeration values:

- [CenterOwner](#) (default)
- [CenterScreen](#)
- [Manual](#)

If the startup location is specified as [Manual](#), and the [Left](#) and [Top](#) properties have not been set, [Window](#) will ask the operating system for a location to appear in.

Topmost windows and z-order

Besides having an x and y location, a window also has a location in the z dimension, which determines its vertical position with respect to other windows. This is known as the window's z-order, and there are two types: **normal** z-order and **topmost** z-order. The location of a window in the *normal* z-order is determined by whether it's currently active or not. By default, a window is located in the normal z-order. The location of a window in the *topmost* z-order is also determined by whether it's currently active or not. Furthermore, windows in the topmost z-order are always located above windows in the normal z-order. A window is located in the topmost z-order by setting its [Topmost](#) property to `true`.

Within each z-order type, the currently active window appears above all other windows in the same z-order.

Window size

Besides having a desktop location, a window has a size that is determined by several properties, including the various width and height properties and [SizeToContent](#).

[MinWidth](#), [Width](#), and [MaxWidth](#) are used to manage the range of widths that a window can have during its lifetime.

```

<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  MinWidth="300" Width="400" MaxWidth="500">
</Window>
  
```

Window height is managed by [MinHeight](#), [Height](#), and [MaxHeight](#).

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    MinHeight="300" Height="400" MaxHeight="500">
</Window>
```

Because the various width values and height values each specify a range, it's possible for the width and height of a resizable window to be anywhere within the specified range for the respective dimension. To detect its current width and height, inspect [ActualWidth](#) and [ActualHeight](#), respectively.

If you'd like the width and height of your window to have a size that fits to the size of the window's content, you can use the [SizeToContent](#) property, which has the following values:

- [SizeToContent.Manual](#)
No effect (default).
- [SizeToContent.Width](#)
Fit to content width, which has the same effect as setting both [MinWidth](#) and [MaxWidth](#) to the width of the content.
- [SizeToContent.Height](#)
Fit to content height, which has the same effect as setting both [MinHeight](#) and [MaxHeight](#) to the height of the content.
- [SizeToContent.WidthAndHeight](#)
Fit to content width and height, which has the same effect as setting both [MinHeight](#) and [MaxHeight](#) to the height of the content, and setting both [MinWidth](#) and [MaxWidth](#) to the width of the content.

The following example shows a window that automatically sizes to fit its content, both vertically and horizontally, when first shown.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    SizeToContent="WidthAndHeight">
</Window>
```

The following example shows how to set the [SizeToContent](#) property in code to specify how a window resizes to fit its content .

```
// Manually alter window height and width
this.SizeToContent = SizeToContent.Manual;

// Automatically resize width relative to content
this.SizeToContent = SizeToContent.Width;

// Automatically resize height relative to content
this.SizeToContent = SizeToContent.Height;

// Automatically resize height and width relative to content
this.SizeToContent = SizeToContent.WidthAndHeight;
```

```
' Manually alter window height and width
Me.SizeToContent = SizeToContent.Manual

' Automatically resize width relative to content
Me.SizeToContent = SizeToContent.Width

' Automatically resize height relative to content
Me.SizeToContent = SizeToContent.Height

' Automatically resize height and width relative to content
Me.SizeToContent = SizeToContent.WidthAndHeight
```

Order of precedence for sizing properties

Essentially, the various sizes properties of a window combine to define the range of width and height for a resizable window. To ensure a valid range is maintained, [Window](#) evaluates the values of the size properties using the following orders of precedence.

For Height Properties:

1. [FrameworkElement.MinHeight](#)
2. [FrameworkElement.MaxHeight](#)
3. [SizeToContent.Height](#) / [SizeToContent.WidthAndHeight](#)
4. [FrameworkElement.Height](#)

For Width Properties:

1. [FrameworkElement.MinWidth](#)
2. [FrameworkElement.MaxWidth](#)
3. [SizeToContent.Width](#) / [SizeToContent.WidthAndHeight](#)
4. [FrameworkElement.Width](#)

The order of precedence can also determine the size of a window when it's maximized, which is managed with the [WindowState](#) property.

Window state

During the lifetime of a resizable window, it can have three states: normal, minimized, and maximized. A window with a *normal* state is the default state of a window. A window with this state allows a user to move and resize it by using a resize grip or the border, if it's resizable.

A window with a *minimized* state collapses to its task bar button if [ShowInTaskbar](#) is set to `true`; otherwise, it collapses to the smallest possible size it can be and moves itself to the bottom-left corner of the desktop. Neither type of minimized window can be resized using a border or resize grip, although a minimized window that isn't shown in the task bar can be dragged around the desktop.

A window with a *maximized* state expands to the maximum size it can be, which will only be as large as its [MaxWidth](#), [MaxHeight](#), and [SizeToContent](#) properties dictate. Like a minimized window, a maximized window can't be resized by using a resize grip or by dragging the border.

NOTE

The values of the [Top](#), [Left](#), [Width](#), and [Height](#) properties of a window always represent the values for the normal state, even when the window is currently maximized or minimized.

The state of a window can be configured by setting its [WindowState](#) property, which can have one of the following [WindowState](#) enumeration values:

- [Normal](#) (default)
- [Maximized](#)
- [Minimized](#)

The following example shows how to create a window that is shown as maximized when it opens.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowState="Maximized">
</Window>
```

In general, you should set [WindowState](#) to configure the initial state of a window. Once a *resizable* window is shown, users can press the minimize, maximize, and restore buttons on the window's title bar to change the window state.

Window appearance

You change the appearance of the client area of a window by adding window-specific content to it, such as buttons, labels, and text boxes. To configure the non-client area, [Window](#) provides several properties, which include [Icon](#) to set a window's icon and [Title](#) to set its title.

You can also change the appearance and behavior of non-client area border by configuring a window's [resize mode](#), [window style](#), and whether it appears as a button in the desktop task bar.

Resize mode

Depending on the [WindowStyle](#) property, you can control if, and how, users resize the window. The window style affects the following:

- Allow or disallow resizing by dragging the window border with the mouse.
- Whether the **Minimize**, **Maximize**, and **Close** buttons appear on the non-client area.
- Whether the **Minimize**, **Maximize**, and **Close** buttons are enabled.

You can configure how a window resizes by setting its [ResizeMode](#) property, which can be one of the following [ResizeMode](#) enumeration values:

- [NoResize](#)
- [CanMinimize](#)
- [CanResize](#) (default)
- [CanResizeWithGrip](#)

As with [WindowStyle](#), the [resize mode](#) of a window is unlikely to change during its lifetime, which means that you'll most likely set it from XAML markup.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  ResizeMode="CanResizeWithGrip">
</Window>
```

Note that you can detect whether a window is maximized, minimized, or restored by inspecting the [WindowState](#) property.

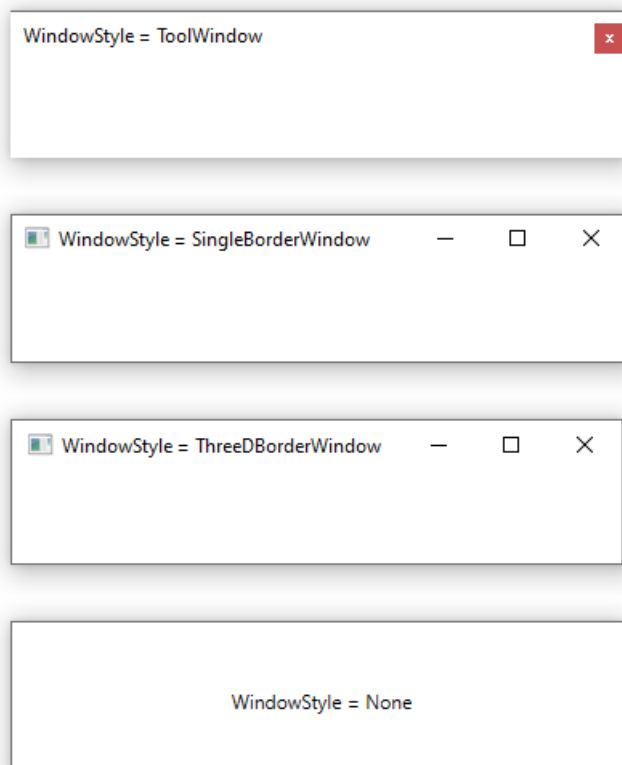
Window style

The border that is exposed from the non-client area of a window is suitable for most applications. However, there are circumstances where different types of borders are needed, or no borders are needed at all, depending on the type of window.

To control what type of border a window gets, you set its [WindowStyle](#) property with one of the following values of the [WindowStyle](#) enumeration:

- [None](#)
- [SingleBorderWindow](#) (default)
- [ThreeDBorderWindow](#)
- [ToolWindow](#)

The effect of applying a window style is illustrated in the following image:



Notice that the image above doesn't show any noticeable difference between [SingleBorderWindow](#) and [ThreeDBorderWindow](#). Back in Windows XP, [ThreeDBorderWindow](#) did affect how the window was drawn, adding a 3D border to the client area. Starting with Windows 7, the differences between the two styles are minimal.

You can set [WindowStyle](#) using either XAML markup or code. Because it's unlikely to change during the lifetime of a window, you'll most likely configure it using XAML markup.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowStyle="ToolWindow">
</Window>
```

Non-rectangular window style

There are also situations where the border styles that [WindowStyle](#) allows you to have aren't sufficient. For example, you may want to create an application with a non-rectangular border, like Microsoft Windows Media Player uses.

For example, consider the speech bubble window shown in the following image:



Greetings!

This type of window can be created by setting the [WindowStyle](#) property to [None](#), and by using special support that [Window](#) has for transparency.

```
<Window x:Class="WindowsOverview.ClippedWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ClippedWindow" SizeToContent="WidthAndHeight"
        WindowStyle="None" AllowsTransparency="True" Background="Transparent">
    <Grid Margin="20">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="20" />
        </Grid.RowDefinitions>

        <Rectangle Stroke="#FF000000" RadiusX="10" RadiusY="10" />
        <Path Fill="White" Stretch="Fill" Stroke="#FF000000" HorizontalAlignment="Left" Margin="15,-
5.597,0,-0.003" Width="30" Grid.Row="1" Data="M22.166642,154.45381 L29.999666,187.66699
40.791059,154.54395" />
        <Rectangle Fill="White" RadiusX="10" RadiusY="10" Margin="1" />

        <TextBlock HorizontalAlignment="Left" VerticalAlignment="Center" FontSize="25" Text="Greetings!"
TextWrapping="Wrap" Margin="5,5,50,5" />
        <Button HorizontalAlignment="Right" VerticalAlignment="Top" Background="Transparent" BorderBrush="
{x:Null}" Foreground="Red" Content="X" FontSize="15" />

        <Grid.Effect>
            <DropShadowEffect BlurRadius="10" ShadowDepth="3" Color="LightBlue" />
        </Grid.Effect>
    </Grid>
</Window>
```

This combination of values instructs the window to render transparent. In this state, the window's non-client area adornment buttons can't be used and you need to provide your own.

Task bar presence

The default appearance of a window includes a taskbar button. Some types of windows don't have a task bar button, such as message boxes, [dialog boxes](#), or windows with the [WindowStyle](#) property set to [ToolWindow](#). You can control whether the task bar button for a window is shown by setting the [ShowInTaskbar](#) property, which is `true` by default.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ShowInTaskbar="False">
</Window>
```

Other types of windows

[NavigationWindow](#) is a window that is designed to host navigable content.

Dialog boxes are windows that are often used to gather information from a user to complete a function. For example, when a user wants to open a file, the **Open File** dialog box is displayed by an application to get the file name from the user. For more information, see [Dialog Boxes Overview](#).

See also

- [Dialog boxes overview](#)
- [How to open a window or dialog box](#)
- [How to open a common dialog box](#)
- [How to open a message box](#)
- [How to close a window or dialog box](#)
- [System.Windows.Window](#)
- [System.Windows.MessageBox](#)
- [System.Windows.Navigation.NavigationWindow](#)
- [System.Windows.Application](#)

Dialog boxes overview (WPF .NET)

4/15/2021 • 10 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) provides ways for you to design your own dialog boxes. Dialog boxes are windows but with a specific intent and user experience. This article discusses how a dialog box works and what types of dialog boxes you can create and use. Dialog boxes are used to:

- Display specific information to users.
- Gather information from users.
- Both display and gather information.
- Display an operating system prompt, such a print window.
- Select a file or folder.

These types of windows are known as *dialog boxes*. A dialog box can be displayed in two ways: modal and modeless.

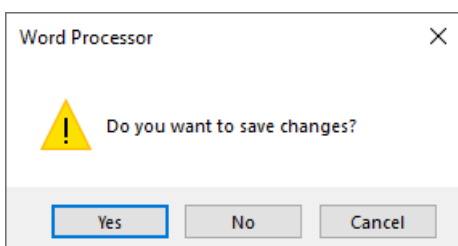
Displaying a *modal* dialog box to the user is a technique with which the application interrupts what it was doing until the user closes the dialog box. This generally comes in the form of a prompt or alert. Other windows in the application can't be interacted with until the dialog box is closed. Once the *modal* dialog box is closed, the application continues. The most common dialog boxes are used to show an open file or save file prompt, displaying the printer dialog, or messaging the user with some status.

A *modeless* dialog box doesn't prevent a user from activating other windows while it's open. For example, if a user wants to find occurrences of a particular word in a document, a main window will often open a dialog box to ask a user what word they're looking for. Since the application doesn't want to prevent the user from editing the document, the dialog box doesn't need to be modal. A modeless dialog box at least provides a **Close** button to close the dialog box. Other buttons may be provided to run specific functions, such as a **Find Next** button to find the next word in a word search.

With WPF you can create several types of dialog boxes, such as message boxes, common dialog boxes, and custom dialog boxes. This article discusses each, and the [Dialog Box Sample](#) provides matching examples.

Message boxes

A *message box* is a dialog box that can be used to display textual information and to allow users to make decisions with buttons. The following figure shows a message box that asks a question and provides the user with three buttons to answer the question.



To create a message box, you use the [MessageBox](#) class. [MessageBox](#) lets you configure the message box text, title, icon, and buttons.

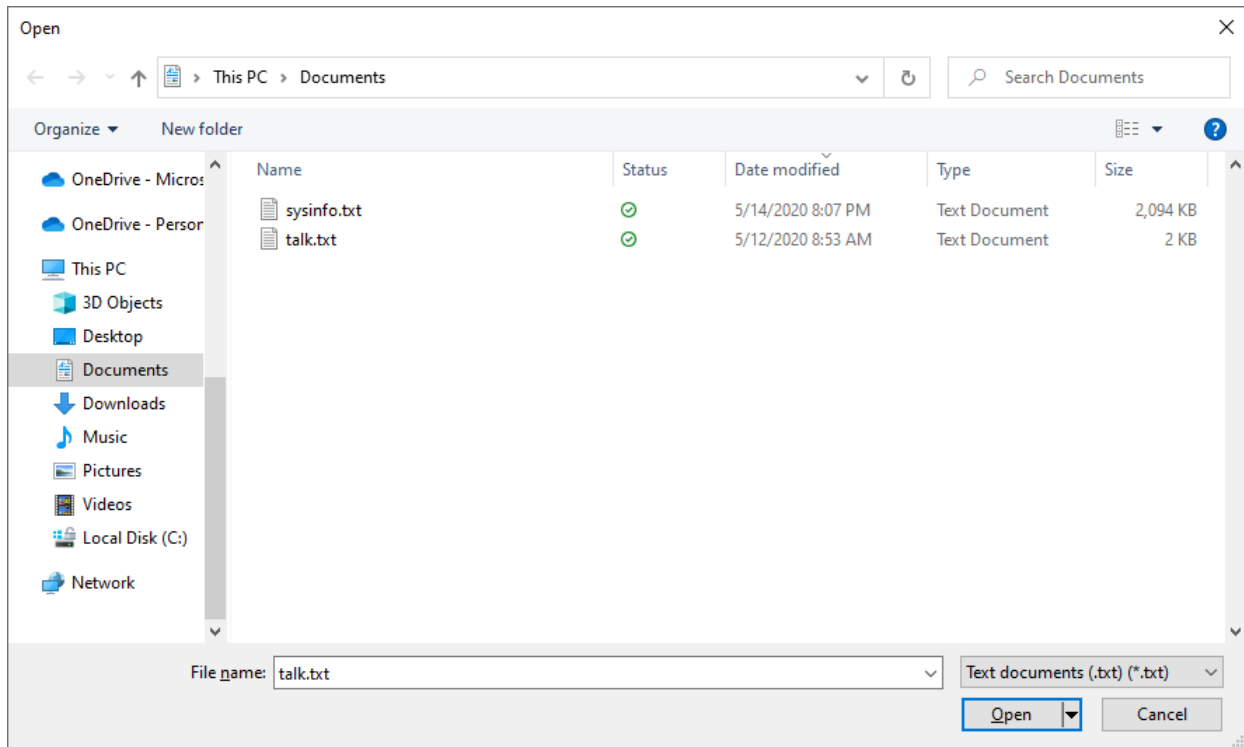
For more information, see [How to open a message box](#).

Common dialog boxes

Windows implements different kinds of reusable dialog boxes that are common to all applications, including dialog boxes for selecting files and printing.

Since these dialog boxes are provided by the operating system, they're shared among all the applications that run on the operating system. These dialog boxes provide a consistent user experience, and are known as *common dialog boxes*. As a user uses a common dialog box in one application, they don't need to learn how to use that dialog box in other applications.

WPF encapsulates the open file, save file, and print common dialog boxes and exposes them as managed classes for you to use in standalone applications.



To learn more about common dialog boxes, see the following articles:

- [How to display a common dialog box](#)
- [Show the Open File dialog box](#)
- [Show the Save File dialog box](#)
- [Show the Print dialog box](#)

Custom dialog boxes

While common dialog boxes are useful, and should be used when possible, they don't support the requirements of domain-specific dialog boxes. In these cases, you need to create your own dialog boxes. As we'll see, a dialog box is a window with special behaviors. [Window](#) implements those behaviors and you use the window to create custom modal and modeless dialog boxes.

There are many design considerations to take into account when you create your own dialog box. Although both an application window and dialog box contain similarities, such as sharing the same base class, a dialog box is used for a specific purpose. Usually a dialog box is required when you need to prompt a user for some sort of information or response. Typically the application will pause while the dialog box (modal) is displayed, restricting access to the rest of the application. Once the dialog box is closed, the application continues. Confining interactions to the dialog box alone, though, isn't a requirement.

When a WPF window is closed, it can't be reopened. Custom dialog boxes are WPF windows and the same rule applies. To learn how to close a window, see [How to close a window or dialog box](#).

Implementing a dialog box

When designing a dialog box, follow these suggestions to create a good user experience:

- ✗ DON'T clutter the dialog window. The dialog experience is for the user to enter some data, or to make a choice.
- ✓ DO provide an **OK** button to close the window.
- ✓ DO set the **OK** button's `IsDefault` property to `true` to allow the user to press the ENTER key to accept and close the window.
- ✓ CONSIDER adding a **Cancel** button so that the user can close the window and indicate that they don't want to continue.
- ✓ DO set the **Cancel** button's `IsCancel` property to `true` to allow the user to press the ESC key to close the window.
- ✓ DO set the title of the window to accurately describe what the dialog represents, or what the user should do with the dialog.
- ✓ DO set minimum width and height values for the window, preventing the user from resizing the window too small.
- ✓ CONSIDER disabling the ability to resize the window if `ShowInTaskbar` is set to `false`. You can disable resizing by setting `ResizeMode` to `NoResize`

The following code demonstrates this configuration.

```

<Window x:Class="Dialogs.Margins"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Change Margins"
        Closing="Window_Closing"
        MinHeight="200"
        MinWidth="300"
        SizeToContent="WidthAndHeight"
        ResizeMode="NoResize"
        ShowInTaskbar="False"
        WindowStartupLocation="CenterOwner"
        FocusManager.FocusedElement="{Binding ElementName=leftMarginTextBox}">
    <Grid Margin="10">
        <Grid.Resources>
            <!-- Default settings for controls -->
            <Style TargetType="{x:Type Label}">
                <Setter Property="Margin" Value="0,3,5,5" />
                <Setter Property="Padding" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type TextBox}">
                <Setter Property="Margin" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type Button}">
                <Setter Property="Width" Value="70" />
                <Setter Property="Height" Value="25" />
                <Setter Property="Margin" Value="5,0,0,0" />
            </Style>
        </Grid.Resources>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <!-- Left,Top,Right,Bottom margins-->
        <Label Grid.Column="0" Grid.Row="0">Left Margin:</Label>
        <TextBox Name="leftMarginTextBox" Grid.Column="1" Grid.Row="0" />

        <Label Grid.Column="0" Grid.Row="1">Top Margin:</Label>
        <TextBox Name="topMarginTextBox" Grid.Column="1" Grid.Row="1"/>

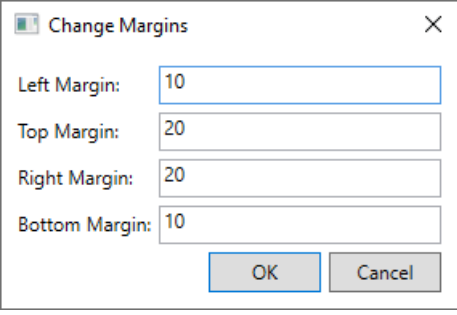
        <Label Grid.Column="0" Grid.Row="2">Right Margin:</Label>
        <TextBox Name="rightMarginTextBox" Grid.Column="1" Grid.Row="2" />

        <Label Grid.Column="0" Grid.Row="3">Bottom Margin:</Label>
        <TextBox Name="bottomMarginTextBox" Grid.Column="1" Grid.Row="3" />

        <!-- Accept or Cancel -->
        <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="4" Orientation="Horizontal"
        HorizontalAlignment="Right">
            <Button Name="okButton" Click="okButton_Click" IsDefault="True">OK</Button>
            <Button Name="cancelButton" IsCancel="True">Cancel</Button>
        </StackPanel>
    </Grid>
</Window>

```

The above XAML creates a window that looks similar to the following image:



UI elements opening a dialog box

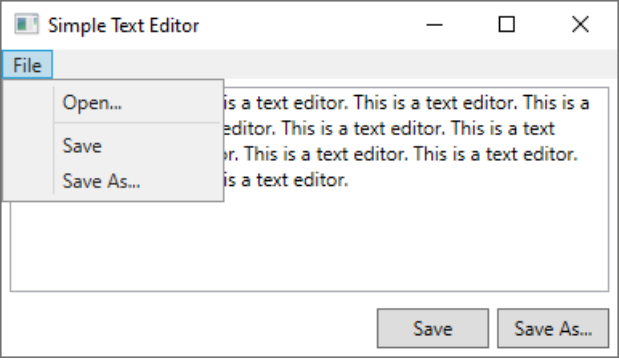
The user experience for a dialog box also extends into the menu bar or the button of the window that opens it. When a menu item or button runs a function that requires user interaction through a dialog box before the function can continue, the control should use an ellipsis at the end of its header text:

```
<MenuItem Header="_Margins..." Click="formatMarginsMenuItem_Click" />
<!-- or -->
<Button Content="_Margins..." Click="formatMarginsButton_Click" />
```

When a menu item or button runs a function that displays a dialog box that **doesn't** require user interaction, such as an *About* dialog box, an ellipsis isn't required.

Menu items

Menu items are a common way to provide users with application actions that are grouped into related themes. You've probably seen the *File* menu on many different applications. In a typical application, the *File* menu item provides ways to save a file, load a file, and print a file. If the action is going to display a modal window, the header typically includes an ellipsis as shown in the following image:

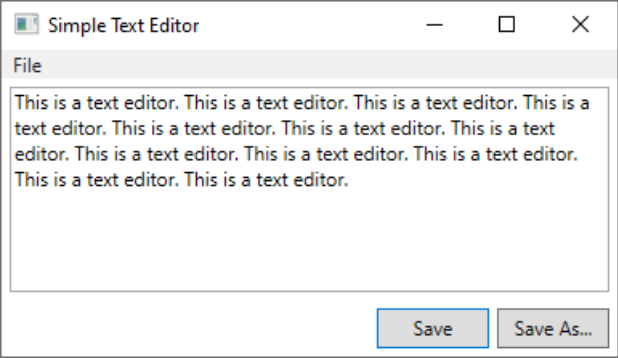


Two of the menu items have an ellipsis: `...`. This helps the user identify that when they select those menu items, a modal window is shown, pausing the application until the user closes it.

This design technique is an easy way for you to communicate to your users what they should expect.

Buttons

You can follow the same principle described in the [Menu items](#) section. Use an ellipsis on the button text to indicate that when the user presses the button, a modal dialog will appear. In the following image, there are two buttons and it's easy to understand which button displays a dialog box:



Return a result

Opening another window, especially a modal dialog box, is a great way to return status and information to calling code.

Modal dialogs

When a dialog box is shown by calling `ShowDialog()`, the code that opened the dialog box waits until the `ShowDialog` method returns. When the method returns, the code that called it needs to decide whether to continue processing or stop processing. The user generally indicates this by pressing an **OK** or **Cancel** button on the dialog box.

When the **OK** button is pressed, `ShowDialog` should be designed to return `true`, and the **Cancel** button to return `false`. This is achieved by setting the `DialogResult` property when the button is pressed.

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

```
Private Sub okButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = True
End Sub

Private Sub cancelButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = False
End Sub
```

The `DialogResult` property can only be set if the dialog box was displayed with `ShowDialog()`. When the `DialogResult` property is set, the dialog box closes.

If a button's `IsCancel` property is set to `true`, and the window is opened with `ShowDialog()`, the ESC key will close the window and set `DialogResult` to `false`.

For more information about closing dialog boxes, see [How to close a window or dialog box](#).

Processing the response

The `ShowDialog()` returns a boolean value to indicate whether the user accepted or canceled the dialog box. If you're alerting the user to something, but not requiring they make a decision or provide data, you can ignore the response. The response can also be inspected by checking the `DialogResult` property. The following code shows how to process the response:

```

var dialog = new Margins();

// Display the dialog box and read the response
bool? result = dialog.ShowDialog();

if (result == true)
{
    // User accepted the dialog box
    MessageBox.Show("Your request will be processed.");
}
else
{
    // User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.");
}

```

```

Dim marginsWindow As New Margins

Dim result As Boolean? = marginsWindow.ShowDialog()

If result = True Then
    ' User accepted the dialog box
    MessageBox.Show("Your request will be processed.")
Else
    ' User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.")
End If

marginsWindow.Show()

```

Modeless dialog

To show a dialog box modeless, call [Show\(\)](#). The dialog box should at least provide a **Close** button. Other buttons and interactive elements can be provided to run a specific function, such as a **Find Next** button to find the next word in a word search.

Because a modeless dialog box doesn't block the calling code from continuing, you must provide a different way of returning a result. You can do one of the following:

- Expose a data object property on the window.
- Handle the [Window.Closed](#) event in the calling code.
- Create events on the window that are raised when the user selects an object or presses a specific button.

The following example uses the [Window.Closed](#) event to display a message box to the user when the dialog box closes. The message displayed references a property of the closed dialog box. For more information about closing dialogs boxes, see [How to close a window or dialog box](#).

```

var marginsWindow = new Margins();

marginsWindow.Closed += (sender, eventArgs) =>
{
    MessageBox.Show($"You closed the margins window! It had the title of {marginsWindow.Title}");
};

marginsWindow.Show();

```



```
Dim marginsWindow As New Margins

AddHandler marginsWindow.Closed, Sub(sender As Object, e As EventArgs)
                                   MessageBox.Show($"You closed the margins window! It had the title of
{marginsWindow.Title}")
                                   End Sub

marginsWindow.Show()
```

See also

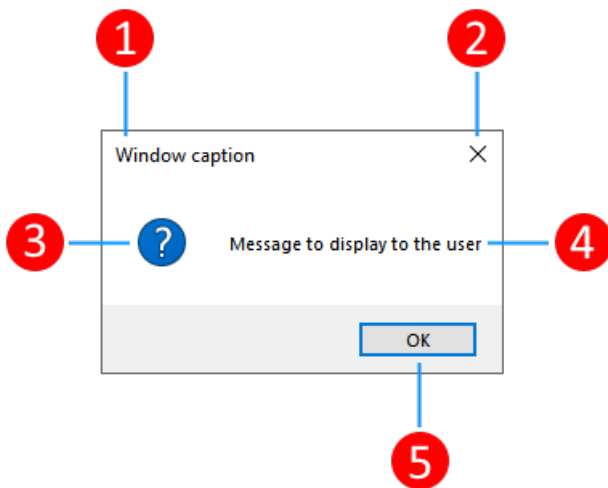
- [Overview of WPF windows](#)
- [How to open a window or dialog box](#)
- [How to open a common dialog box](#)
- [How to open a message box](#)
- [How to close a window or dialog box](#)
- [Dialog Box Sample](#)
- [System.Windows.Window](#)
- [System.Windows.MessageBox](#)

How to open a message box (WPF .NET)

4/15/2021 • 3 minutes to read • [Edit Online](#)

A *message box* is a dialog box that is used to quickly display information and optionally allow users to make decisions. Access to the message box is provided by the [MessageBox](#) class. A message box is displayed *modally*. And the code that displays the message box is paused until the user closes the message box either with the close button or a response button.

The following illustration demonstrates the parts of a message box:



- A title bar with a caption (1).
- A close button (2).
- Icon (3).
- Message displayed to the user (4).
- Response buttons (5).

For presenting or gathering complex data, a dialog box might be more suitable than a message box. For more information, see [Dialog boxes overview](#).

Display a message box

To create a message box, you use the [MessageBox](#) class. The [MessageBox.Show](#) method lets you configure the message box text, title, icon, and buttons, shown in the following code:

```
string messageBoxText = "Do you want to save changes?";
string caption = "Word Processor";
MessageBoxButton button = MessageBoxButton.YesNoCancel;
MessageBoxImage icon = MessageBoxImage.Warning;
MessageBoxResult result;

result = MessageBox.Show(messageBoxText, caption, button, icon, MessageBoxResult.Yes);
```

```
Dim messageBoxText As String = "Do you want to save changes?"
Dim caption As String = "Word Processor"
Dim Button As MessageBoxButton = MessageBoxButton.YesNoCancel
Dim Icon As MessageBoxImage = MessageBoxImage.Warning
Dim result As MessageBoxResult

result = MessageBox.Show(messageBoxText, caption, Button, Icon, MessageBoxResult.Yes)
```

The [MessageBox.Show](#) method overloads provide ways to configure the message box. These options include:

- Title bar **caption**
- Message **icon**
- Message **text**
- Response **buttons**

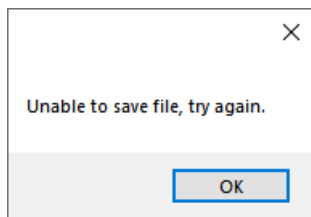
Here are some more examples of using a message box.

- Display an alert.

```
MessageBox.Show("Unable to save file, try again.");
```

```
MessageBox.Show("Unable to save file, try again.")
```

The previous code displays a message box like the following image:

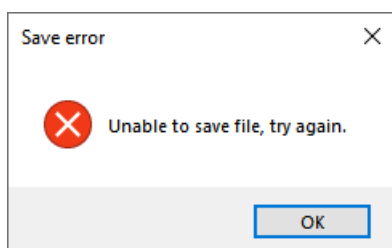


It's a good idea to use the options provided by the message box class. Using the same alert as before, set more options to make it more visually appealing:

```
MessageBox.Show("Unable to save file, try again.", "Save error", MessageBoxButton.OK,
    MessageBoxImage.Error);
```

```
MessageBox.Show("Unable to save file, try again.", "Save error", MessageBoxButton.OK,
    MessageBoxImage.Error)
```

The previous code displays a message box like the following image:

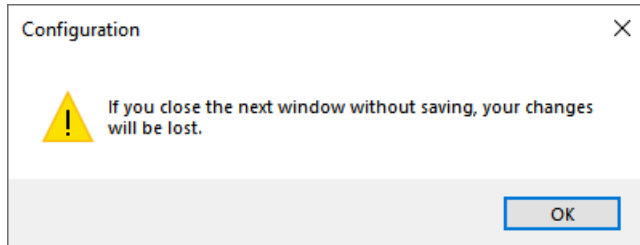


- Display a warning.

```
MessageBox.Show("If you close the next window without saving, your changes will be lost.",  
"Configuration", MessageBoxButtons.OK, MessageBoxIcon.Warning);
```

```
MessageBox.Show("If you close the next window without saving, your changes will be lost.",  
"Configuration", MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

The previous code displays a message box like the following image:

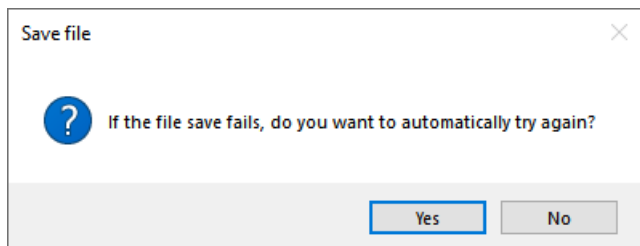


- Ask the user a question.

```
if (MessageBox.Show("If the file save fails, do you want to automatically try again?",  
"Save file",  
MessageBoxButton.YesNo,  
MessageBoxImage.Question) == DialogResult.Yes)  
{  
    // Do something here  
}
```

```
If MessageBox.Show("If the file save fails, do you want to automatically try again?",  
"Save file",  
MessageBoxButton.YesNo,  
MessageBoxImage.Question) = DialogResult.Yes Then  
  
    ' Do something here  
  
End If
```

The previous code displays a message box like the following image:



Handle a message box response

The `MessageBox.Show` method displays the message box and returns a result. The result indicates how the user closed the message box:

```

result = MessageBox.Show(messageBoxText, caption, button, icon, MessageBoxResult.Yes);

switch (result)
{
    case MessageBoxResult.Cancel:
        // User pressed Cancel
        break;
    case MessageBoxResult.Yes:
        // User pressed Yes
        break;
    case MessageBoxResult.No:
        // User pressed No
        break;
}

```

```

result = MessageBox.Show(messageBoxText, caption, Button, Icon, MessageBoxResult.Yes)

Select Case result
    Case MessageBoxResult.Cancel
        ' User pressed Cancel
    Case MessageBoxResult.Yes
        ' User pressed Yes
    Case MessageBoxResult.No
        ' User pressed No
End Select

```

When a user presses the buttons at the bottom of the message box, the corresponding [MessageBoxResult](#) is returned. However, if the user presses the ESC key or presses the **Close** button (#2 in the [message box illustration](#)), the result of the message box varies based on the button options:

BUTTON OPTIONS	ESC OR CLOSE BUTTON RESULT
	
	
	ESC keyboard shortcut and Close button disabled. User must press Yes or No .
	

For more information on using message boxes, see [MessageBox](#) and the [MessageBox sample](#).

See also

- [Overview of WPF windows](#)
- [Dialog boxes overview](#)
- [How to display a common dialog box](#)
- [MessageBox sample](#)
- [System.Windows.MessageBox](#)
- [System.Windows.MessageBox.Show](#)
- [System.Windows.MessageBoxResult](#)

How to open a window or dialog box (WPF .NET)

4/15/2021 • 2 minutes to read • [Edit Online](#)

You can create your own windows and display them in Windows Presentation Foundation (WPF). In this article, you'll learn how to display modal and modeless windows and dialogs.

Conceptually, a window and a dialog box are the same thing: they're displayed to a user to provide information or interaction. They're both "window" objects. The design of the window and the way it's used, is what makes a dialog box. A dialog box is generally small in size and requires the user to respond to it. For more information, see [Overview of WPF windows](#) and [Dialog boxes overview](#).

If you're interested in opening operating system dialog boxes, see [How to open a common dialog box](#).

Open as modal

When a modal window is opened, it generally represents a dialog box. WPF restricts interaction to the modal window, and the code that opened the window pauses until the window closes. This mechanism provides an easy way for you to prompt the user with data and wait for their response.

Use the [ShowDialog](#) method to open a window. The following code instantiates the window, and opens it modally. The code opening the window pauses, waiting for the window to be closed:

```
var window = new Margins();

window.Owner = this;
window.ShowDialog();
```

```
Dim myWindow As New Margins()

myWindow.Owner = Me
myWindow.ShowDialog()
```

IMPORTANT

Once a window is closed, the same object instance can't be used to reopen the window.

For more information about how to handle the user response to a dialog box, see [Dialog boxes overview: Processing the response](#).

Open as modeless

Opening a window modeless means displaying it as a normal window. The code that opens the window continues to run as the window becomes visible. You can focus and interact with all modeless windows displayed by your application, without restriction.

Use the [Show](#) method to open a window. The following code instantiates the window, and opens it modeless. The code opening the window continues to run:

```
var window = new Margins();
```

```
window.Owner = this;  
window.Show();
```

```
Dim myWindow As New Margins()
```

```
myWindow.Owner = Me  
myWindow.Show()
```

IMPORTANT

Once a window is closed, the same object instance can't be used to reopen the window.

See also

- [Overview of WPF windows](#)
- [Dialog boxes overview](#)
- [How to close a window or dialog box](#)
- [How to open a common dialog box](#)
- [How to open a message box](#)
- [System.Windows.Window](#)
- [System.Windows.Window.DialogResult](#)
- [System.Windows.Window.Show\(\)](#)
- [System.Windows.Window.ShowDialog\(\)](#)

How to close a window or dialog box (WPF .NET)

4/15/2021 • 3 minutes to read • [Edit Online](#)

In this article, you'll learn about the different ways to close a window or dialog box. A user can close a window by using the elements in the non-client area, including the following:

- The **Close** item of the **System** menu.
- Pressing ALT + F4.
- Pressing the **Close** button.
- Pressing ESC when a button has the **IsCancel** property set to `true` on a modal window.

When designing a window, provide more mechanisms to the client area to close a window. Some of the common design elements on a window that are used to close it include the following:

- An **Exit** item in the **File** menu, typically for main application windows.
- A **Close** item in the **File** menu, typically on a secondary application window.
- A **Cancel** button, typically on a modal dialog box.
- A **Close** button, typically on a modeless dialog box.

IMPORTANT

Once a window is closed, the same object instance can't be used to reopen the window.

For more information about the life of a window, see [Overview of WPF windows: Window lifetime](#).

Close a modal window

When closing a window that was opened with the [ShowDialog](#) method, set the [DialogResult](#) property to either `true` or `false` to indicate an "accepted" or "canceled" state, respectively. As soon as the [DialogResult](#) property is set to a value, the window closes. The following code demonstrates setting the [DialogResult](#) property:

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

```
Private Sub okButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = True
End Sub

Private Sub cancelButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = False
End Sub
```

You can also call the [Close](#) method. If the [Close](#) method is used, the [DialogResult](#) property is set to `false`.

Once a window has been closed, it can't be reopened with the same object instance. If you try to show the same window, a [InvalidOperationException](#) is thrown. Instead, create a new instance of the window and open it.

Close a modeless window

When closing a window that was opened with the [Show](#) method, use the [Close](#) method. The following code demonstrates closing a modeless window:

```
private void closeButton_Click(object sender, RoutedEventArgs e) =>
    Close();
```

```
Private Sub closeButton_Click(sender As Object, e As RoutedEventArgs)
    Close()
End Sub
```

Close with IsCancel

The [Button.IsCancel](#) property can be set to `true` to enable the ESC key to automatically close the window. This only works when the window is opened with [ShowDialog](#) method.

```
<Button Name="cancelButton" IsCancel="True">Cancel</Button>
```

Hide a window

Instead of closing a window, a window can be hidden with the [Hide](#) method. A hidden window can be reopened, unlike a window that has been closed. If you're going to reuse a window object instance, hide the window instead of closing it. The following code demonstrates hiding a window:

```
private void saveButton_Click(object sender, RoutedEventArgs e) =>
    Hide();
```

```
Private Sub saveButton_Click(sender As Object, e As RoutedEventArgs)
    Hide()
End Sub
```

Cancel close and hide

If you've designed your buttons to hide a window instead of closing it, the user can still bypass this and close the window. The **Close** item of the system menu and the **Close** button of the non-client area of the window, will close the window instead of hiding it. Consider this scenario when your intent is to hide a window instead of closing it.

Caution

If a window is shown modally with [ShowDialog](#), the [DialogResult](#) property will be set to `null` when the window is hidden. You'll need to communicate state back to the calling code by adding your own property to the window.

When a window is closed, the [Closing](#) event is raised. The handler is passed a [CancelEventArgs](#), which implements the [Cancel](#) property. Set that property to `true` to prevent a window from closing. The following code demonstrates how to cancel the closure and instead hide the window:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    // Cancel the closure
    e.Cancel = true;

    // Hide the window
    Hide();
}
```

```
Private Sub Window_Closing(sender As Object, e As ComponentModel.CancelEventArgs)
    ' Cancel the closure
    e.Cancel = True

    ' Hide the window
    Hide()
End Sub
```

There may be times where you don't want to hide a window, but actually prevent the user from closing it. For more information, see [Overview of WPF windows: Cancel window closure](#).

See also

- [Overview of WPF windows](#)
- [Dialog boxes overview](#)
- [How to open a window or dialog box](#)
- [System.Windows.Window.Close\(\)](#)
- [System.Windows.Window.Closing](#)
- [System.Windows.Window.DialogResult](#)
- [System.Windows.Window.Hide\(\)](#)
- [System.Windows.Window.Show\(\)](#)
- [System.Windows.Window.ShowDialog\(\)](#)

How to open a common dialog box (WPF .NET)

4/15/2021 • 3 minutes to read • [Edit Online](#)

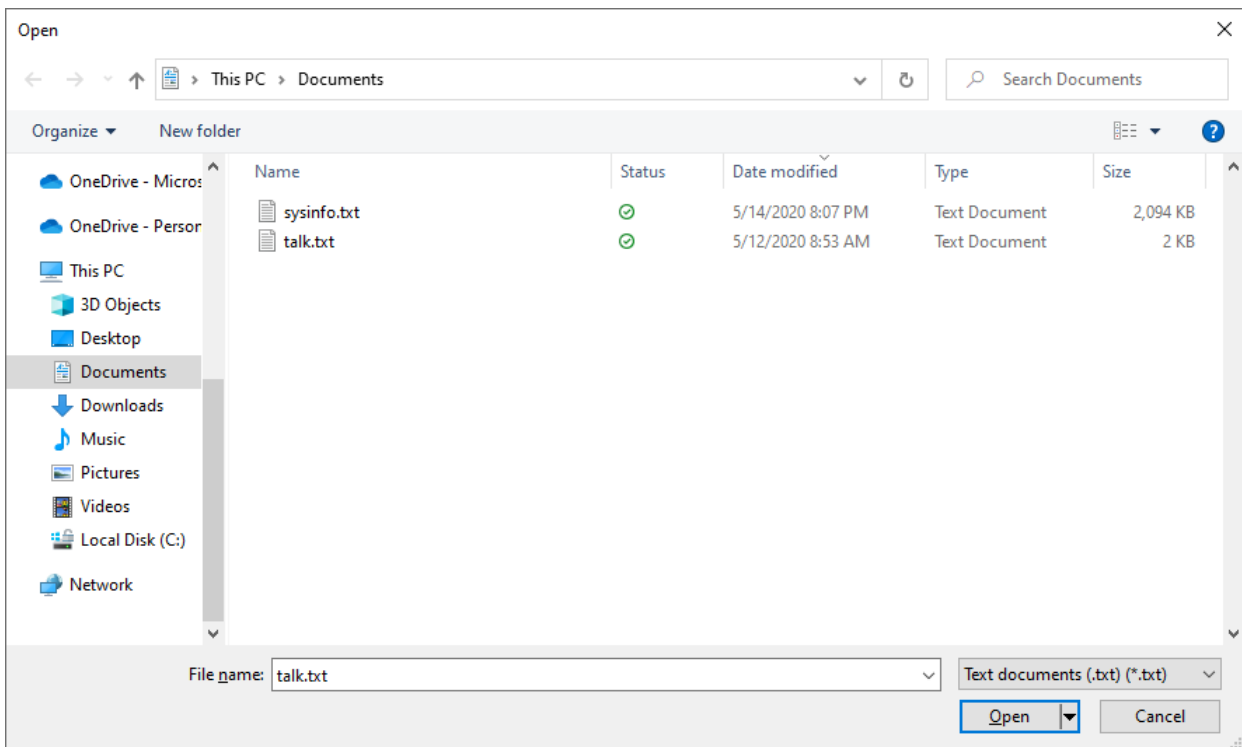
This article demonstrates how you can display a common system dialog box in Windows Presentation Foundation (WPF). Windows implements different kinds of reusable dialog boxes that are common to all applications, including dialog boxes for selecting files and printing.

Since these dialog boxes are provided by the operating system, they're shared among all the applications that run on the operating system. These dialog boxes provide a consistent user experience, and are known as *common dialog boxes*. As a user uses a common dialog box in one application, they don't need to learn how to use that dialog box in other applications.

A message box is another common dialog box. For more information, see [How to open a message box](#).

Open File dialog box

The open file dialog box is used by file opening functionality to retrieve the name of a file to open.



The common open file dialog box is implemented as the [OpenFileDialog](#) class and is located in the [Microsoft.Win32](#) namespace. The following code shows how to create, configure, and show one, and how to process the result.

```
// Configure open file dialog box
var dialog = new Microsoft.Win32.OpenFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show open file dialog box
bool? result = dialog.ShowDialog();

// Process open file dialog box results
if (result == true)
{
    // Open document
    string filename = dialog.FileName;
}
```

```
' Configure open file dialog box
Dim dialog As New Microsoft.Win32.OpenFileDialog()
dialog.FileName = "Document" ' Default file name
dialog.DefaultExt = ".txt" ' Default file extension
dialog.Filter = "Text documents (.txt)|*.txt" ' Filter files by extension

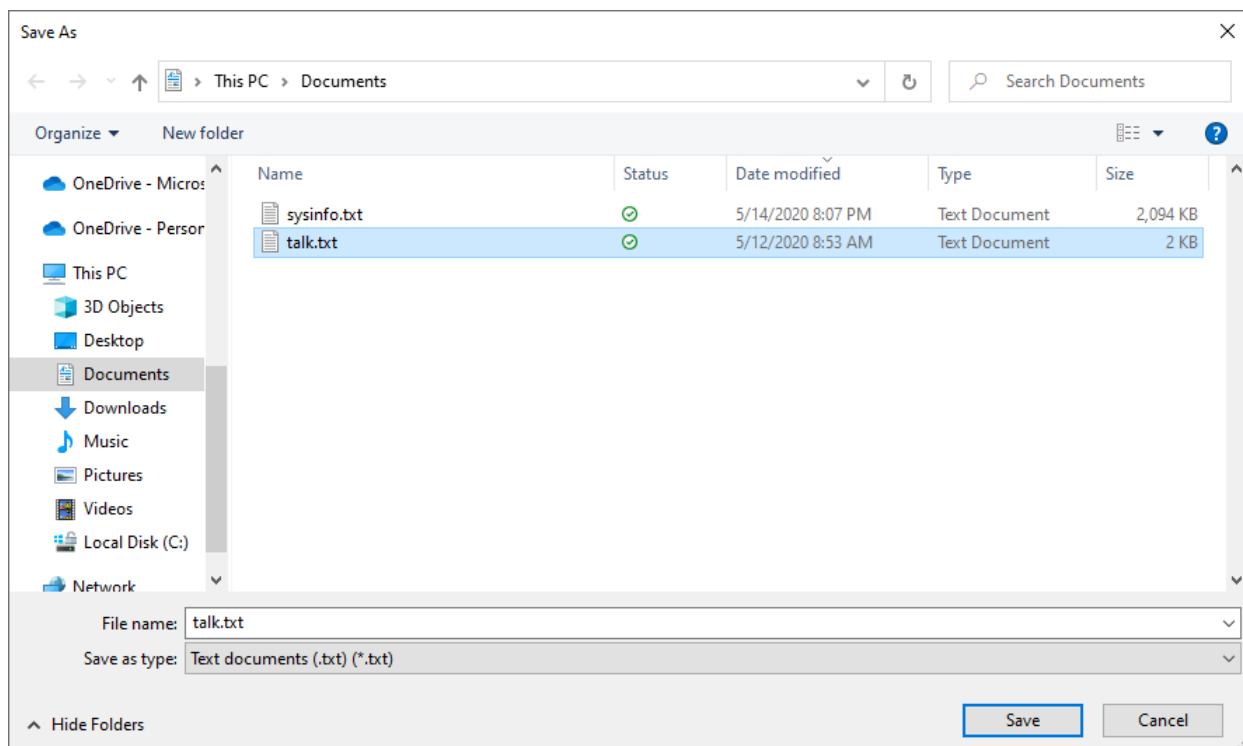
' Show open file dialog box
Dim result As Boolean? = dialog.ShowDialog()

' Process open file dialog box results
If result = True Then
    ' Open document
    Dim filename As String = dialog.FileName
End If
```

For more information on the open file dialog box, see [Microsoft.Win32.OpenFileDialog](#).

Save File dialog box

The save file dialog box is used by file saving functionality to retrieve the name of a file to save.



The common save file dialog box is implemented as the [SaveFileDialog](#) class, and is located in the

[Microsoft.Win32](#) namespace. The following code shows how to create, configure, and show one, and how to process the result.

```
// Configure save file dialog box
var dialog = new Microsoft.Win32.SaveFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show save file dialog box
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Save document
    string filename = dialog.FileName;
}
```

```
' Configure save file dialog box
Dim dialog As New Microsoft.Win32.SaveFileDialog()
dialog.FileName = "Document" ' Default file name
dialog.DefaultExt = ".txt" ' Default file extension
dialog.Filter = "Text documents (.txt)|*.txt" ' Filter files by extension

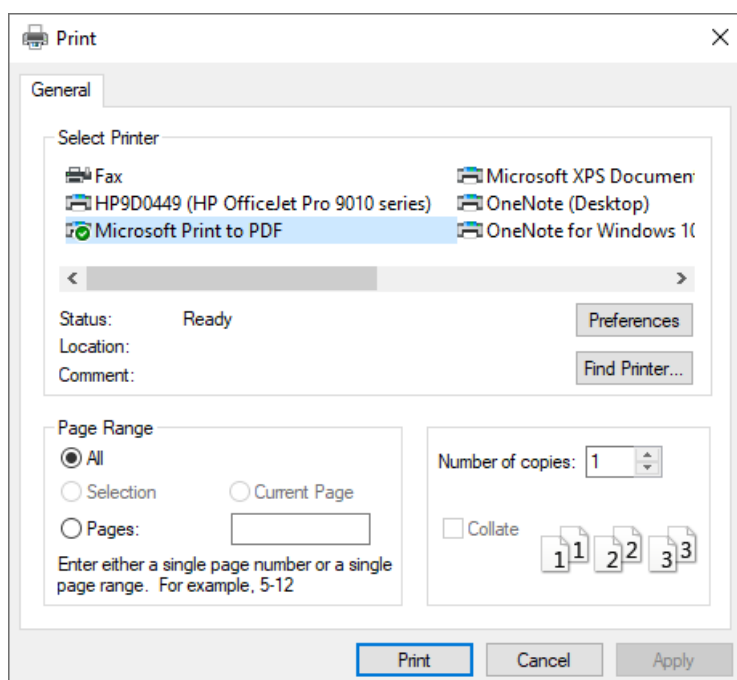
' Show save file dialog box
Dim result As Boolean? = dialog.ShowDialog()

' Process save file dialog box results
If result = True Then
    ' Save document
    Dim filename As String = dialog.FileName
End If
```

For more information on the save file dialog box, see [Microsoft.Win32.SaveFileDialog](#).

Print dialog box

The print dialog box is used by printing functionality to choose and configure the printer that a user wants to print data to.



The common print dialog box is implemented as the [PrintDialog](#) class, and is located in the [System.Windows.Controls](#) namespace. The following code shows how to create, configure, and show one.

```
// Configure printer dialog box
var dialog = new System.Windows.Controls.PrintDialog();
dialog.PageRangeSelection = System.Windows.Controls.PageRangeSelection.AllPages;
dialog.UserPageRangeEnabled = true;

// Show save file dialog box
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Document was printed
}
```

```
' Configure printer dialog box
Dim dialog As New System.Windows.Controls.PrintDialog()
dialog.PageRangeSelection = System.Windows.Controls.PageRangeSelection.AllPages
dialog.UserPageRangeEnabled = True

' Show save file dialog box
Dim result As Boolean? = dialog.ShowDialog()

' Process save file dialog box results
If result = True Then
    ' Document was printed
End If
```

For more information on the print dialog box, see [System.Windows.Controls.PrintDialog](#). For detailed discussion of printing in WPF, see [Printing overview](#).

See also

- [How to open a message box](#)
- [Dialog boxes overview](#)
- [Overview of WPF windows](#)
- [Microsoft.Win32.OpenFileDialog](#)
- [Microsoft.Win32.SaveFileDialog](#)
- [System.Windows.Controls.PrintDialog](#)

How to get or set the main application window (WPF .NET)

4/15/2021 • 2 minutes to read • [Edit Online](#)

This article teaches you how to get or set the main application window for Windows Presentation Foundation (WPF). The first [Window](#) that is instantiated within a WPF application is automatically set by [Application](#) as the main application window. The main window is referenced with the [Application.MainWindow](#) property.

Much of the time a project template will set the [Application.StartupUri](#) to a XAML file within your application, such as `_Window1.xaml_`. This is the first window instantiated and shown by your application, and it becomes the main window.

TIP

The default behavior for an application is to shutdown when the last window is closed. This behavior is controlled by the [Application.ShutdownMode](#) property. Instead, you can configure the application to shutdown if the [MainWindow](#) is closed. Set [Application.ShutdownMode](#) to [OnMainWindowClose](#) to enable this behavior.

Set the main window in XAML

The templates that generate your WPF application typically set the [Application.StartupUri](#) property to a XAML file. This property is helpful because:

1. It's easily changeable to a different XAML file in your project.
2. Automatically instantiates and displays the specified window.
3. The specified window becomes the [Application.MainWindow](#).

```
<Application x:Class="MainApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:MainApp"
    StartupUri="Window1.xaml">

</Application>
```

Instead of using [Application.StartupUri](#), you can set the [Application.MainWindow](#) to a XAML-declared window. However, the window specified here won't be displayed and you must set its visibility.

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

    <Application.MainWindow>
        <local:Window2 Visibility="Visible" />
    </Application.MainWindow>
</Application>
```

Caution

If you set both the [Application.StartupUri](#) and the [Application.MainWindow](#) properties, you'll display both windows when your application starts.

Also, you can use the [Application.Startup](#) event to open a window. For more information, see [Use the Startup event to open a window](#).

Set the main window in code

The first window instantiated by your application automatically becomes the main window and is set to the [Application.MainWindow](#) property. To set a different main window, change this property to a window:

```
Application.Current.MainWindow = new Window2();

Application.Current.MainWindow.Show();
```

```
Application.Current.MainWindow = New Window2()

Application.Current.MainWindow.Show()
```

If your application has never created an instance of a window, the following code is functionally equivalent to the previous code:

```
var appWindow = new Window2();

appWindow.Show();
```

```
Dim appWindow As New Window2()

appWindow.Show()
```

As soon as the window object instance is created, it's assigned to [Application.MainWindow](#).

Get the main window

You can access the window chosen as the main window by inspecting the [Application.MainWindow](#) property. The following code displays a message box with the title of the main window when a button is clicked:

```
private void Button_Click(object sender, RoutedEventArgs e) =>
    MessageBox.Show($"The main window's title is: {Application.Current.MainWindow.Title}");
```

```
Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
    MessageBox.Show($"The main window's title is: {Application.Current.MainWindow.Title}")
End Sub
```

See also

- [Overview of WPF windows](#)
- [Use the Startup event to open a window](#)
- [How to open a window or dialog box](#)
- [System.Windows.Application](#)
- [System.Windows.Application.MainWindow](#)
- [System.Windows.Application.StartupUri](#)
- [System.Windows.Application.ShutdownMode](#)

Styles and templates (WPF .NET)

4/15/2021 • 12 minutes to read • [Edit Online](#)

Windows Presentation Foundation (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, you 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.

This overview focuses on the styling and templating aspects of the app and doesn't discuss any data-binding concepts. For information about data binding, see [Data Binding Overview](#).

It's important to understand resources, which are what enable styles and templates to be reused. For more information about resources, see [Overview of XAML resources](#).

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Sample

The sample code provided in this overview is based on a [simple photo browsing application](#) shown in the following illustration.



This simple photo sample uses styling and templating to create a visually compelling user experience. The sample has two [TextBlock](#) elements and a [ListBox](#) control that is bound to a list of images.

For the complete sample, see [Introduction to Styling and Templating Sample](#).

Styles

You can think of a [Style](#) as a convenient way to apply a set of property values to multiple elements. You 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. Because styles are resources, they obey the same scoping rules that apply to all resources. Put simply, where you declare a style affects where the style can be applied. For example, if you declare the style in the root element of your app definition XAML file, the style can be used anywhere in your app.

For example, the following XAML code declares two styles for a `TextBlock`, one automatically applied to all `TextBlock` elements, and another that must be explicitly referenced.

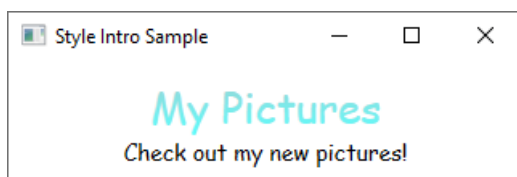
```
<Window.Resources>
  <!-- .... other resources .... -->

  <!--A Style that affects all TextBlocks-->
  <Style TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
  </Style>

  <!--A Style that extends the previous TextBlock Style with an x:Key of TitleText-->
  <Style BasedOn="{StaticResource {x:Type TextBlock}}"
    TargetType="TextBlock"
    x:Key="TitleText">
    <Setter Property="FontSize" Value="26"/>
    <Setter Property="Foreground">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <LinearGradientBrush.GradientStops>
            <GradientStop Offset="0.0" Color="#90DDDD" />
            <GradientStop Offset="1.0" Color="#5BFFFF" />
          </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Here is an example of the styles declared above being used.

```
<StackPanel>
  <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My Pictures</TextBlock>
  <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```



For more information, see [Create a style for a control](#).

ControlTemplates

In WPF, the [ControlTemplate](#) of a control defines the appearance of the control. You can change the structure and appearance of a control by defining a new [ControlTemplate](#) and assigning it to a control. In many cases, templates give you enough flexibility so that you do not have to write your own custom controls.

Each control has a default template assigned to the [Control.Template](#) property. The template connects the visual presentation of the control with the control's capabilities. Because you define a template in XAML, you can change the control's appearance without writing any code. Each template is designed for a specific control, such as a [Button](#).

Commonly you declare a template as a resource on the `Resources` section of a XAML file. As with all resources, scoping rules apply.

Control templates are a lot more involved than a style. This is because the control template rewrites the visual appearance of the entire control, while a style simply applies property changes to the existing control. However, since the template of a control is applied by setting the [Control.Template](#) property, you can use a style to define or set a template.

Designers generally allow you to create a copy of an existing template and modify it. For example, in the Visual Studio WPF designer, select a `CheckBox` control, and then right-click and select **Edit template > Create a copy**. This command generates a *style that defines a template*.

```
<Style x:Key="CheckBoxStyle1" TargetType="{x:Type CheckBox}">
  <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual1}"/>
  <Setter Property="Background" Value="{StaticResource OptionMark.Static.Background1}"/>
  <Setter Property="BorderBrush" Value="{StaticResource OptionMark.Static.Border1}"/>
  <Setter Property="Foreground" Value="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"/>
  <Setter Property="BorderThickness" Value="1"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type CheckBox}">
        <Grid x:Name="templateRoot" Background="Transparent" SnapsToDevicePixels="True">
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*/>
          </Grid.ColumnDefinitions>
          <Border x:Name="checkBoxBorder" Background="{TemplateBinding Background}"
            BorderThickness="{TemplateBinding BorderThickness}" BorderBrush="{TemplateBinding BorderBrush}"
            HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="1" VerticalAlignment="
            {TemplateBinding VerticalContentAlignment}">
            <Grid x:Name="markGrid">
              <Path x:Name="optionMark" Data="F1 M 9.97498,1.22334L 4.6983,9.09834L
                4.52164,9.09834L 0,5.19331L 1.27664,3.52165L 4.255,6.08833L 8.33331,1.52588e-005L 9.97498,1.22334 Z " Fill="
                {StaticResource OptionMark.Static.Glyph1}" Margin="1" Opacity="0" Stretch="None"/>
              <Rectangle x:Name="indeterminateMark" Fill="{StaticResource
                OptionMark.Static.Glyph1}" Margin="2" Opacity="0"/>
            </Grid>
          </Border>
          <ContentPresenter x:Name="contentPresenter" Grid.Column="1" Focusable="False"
            HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding Padding}"
            RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding SnapsToDevicePixels}" VerticalAlignment="
            {TemplateBinding VerticalContentAlignment}"/>
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property="HasContent" Value="true">
            <Setter Property="FocusVisualStyle" Value="{StaticResource
              OptionMarkFocusVisual1}"/>
            <Setter Property="Padding" Value="4,-1,0,0"/>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

... content removed to save space ...

Editing a copy of a template is a great way to learn how templates work. Instead of creating a new blank template, it's easier to edit a copy and change a few aspects of the visual presentation.

For an example, see [Create a template for a control](#).

TemplateBinding

You may have noticed that the template resource defined in the previous section uses the [TemplateBinding Markup Extension](#). A `TemplateBinding` is an optimized form of a binding for template scenarios, analogous to a binding constructed with `{Binding RelativeSource={RelativeSource TemplatedParent}}`. `TemplateBinding` is useful for binding parts of the template to properties of the control. For example, each control has a [BorderThickness](#) property. Use a `TemplateBinding` to manage which element in the template is affected by this control setting.

ContentControl and ItemsControl

If a [ContentPresenter](#) is declared in the [ControlTemplate](#) of a [ContentControl](#), the [ContentPresenter](#) will

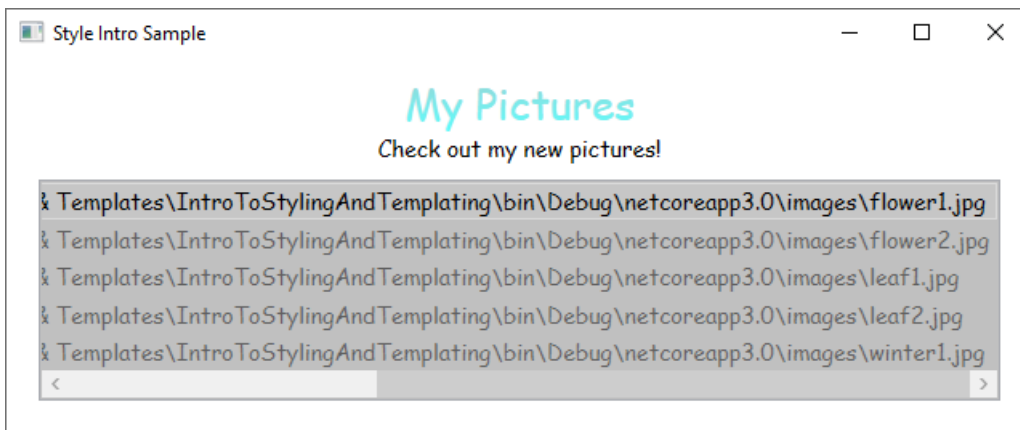
automatically bind to the [ContentTemplate](#) and [Content](#) properties. Likewise, an [ItemsPresenter](#) that is in the [ControlTemplate](#) of an [ItemsControl](#) will automatically bind to the [ItemTemplate](#) and [Items](#) properties.

DataTemplates

In this sample app, there is a [ListBox](#) control that is bound to a list of photos.

```
<ListBox ItemsSource="{Binding Source={StaticResource MyPhotos}}"
        Background="Silver" Width="600" Margin="10" SelectedIndex="0"/>
```

This [ListBox](#) currently looks like the following.



Most controls have some type of content, and that content often comes from data that you are binding to. In this sample, the data is the list of photos. In WPF, you use a [DataTemplate](#) to define the visual representation of data. Basically, what you put into a [DataTemplate](#) determines what the data looks like in the rendered app.

In our sample app, each custom `Photo` object has a `Source` property of type string that specifies the file path of the image. Currently, the photo objects appear as file paths.

```
public class Photo
{
    public Photo(string path)
    {
        Source = path;
    }

    public string Source { get; }

    public override string ToString() => Source;
}
```

```
Public Class Photo
    Sub New(ByVal path As String)
        Source = path
    End Sub

    Public ReadOnly Property Source As String

    Public Overrides Function ToString() As String
        Return Source
    End Function
End Class
```

For the photos to appear as images, you create a [DataTemplate](#) as a resource.

```

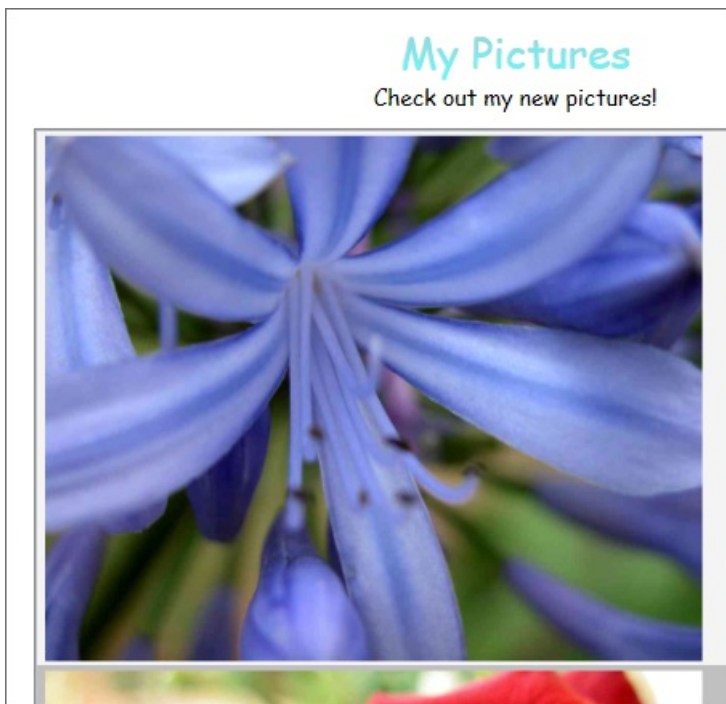
<Window.Resources>
    <!-- .... other resources .... -->

    <!--DataTemplate to display Photos as images
    instead of text strings of Paths-->
    <DataTemplate DataType="{x:Type local:Photo}">
        <Border Margin="3">
            <Image Source="{Binding Source}"/>
        </Border>
    </DataTemplate>
</Window.Resources>

```

Notice that the [DataType](#) property is similar to the [TargetType](#) property of the [Style](#). If your [DataTemplate](#) is in the resources section, when you specify the [DataType](#) property to a type and omit an `x:Key`, the [DataTemplate](#) is applied whenever that type appears. You always have the option to assign the [DataTemplate](#) with an `x:Key` and then set it as a `StaticResource` for properties that take [DataTemplate](#) types, such as the [ItemTemplate](#) property or the [ContentTemplate](#) property.

Essentially, the [DataTemplate](#) in the above example defines that whenever there is a `Photo` object, it should appear as an [Image](#) within a [Border](#). With this [DataTemplate](#), our app now looks like this.



The data templating model provides other features. For example, if you are displaying collection data that contains other collections using a [HeaderedItemsControl](#) type such as a [Menu](#) or a [TreeView](#), there is the [HierarchicalDataTemplate](#). Another data templating feature is the [DataTemplateSelector](#), which allows you to choose a [DataTemplate](#) to use based on custom logic. For more information, see [Data Templating Overview](#), which provides a more in-depth discussion of the different data templating features.

Triggers

A trigger sets properties or starts actions, such as an animation, when a property value changes or when an event is raised. [Style](#), [ControlTemplate](#), and [DataTemplate](#) all have a `Triggers` property that can contain a set of triggers. There are several types of triggers.

PropertyTriggers

A [Trigger](#) that sets property values or starts actions based on the value of a property is called a property trigger.

To demonstrate how to use property triggers, you can make each [ListBoxItem](#) partially transparent unless it is

selected. The following style sets the [Opacity](#) value of a [ListBoxItem](#) to `0.5`. When the [IsSelected](#) property is `true`, however, the [Opacity](#) is set to `1.0`.

```
<Window.Resources>
  <!-- .... other resources .... -->

  <Style TargetType="ListBoxItem">
    <Setter Property="Opacity" Value="0.5" />
    <Setter Property="MaxHeight" Value="75" />
    <Style.Triggers>
      <Trigger Property="IsSelected" Value="True">
        <Trigger.Setters>
          <Setter Property="Opacity" Value="1.0" />
        </Trigger.Setters>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

This example uses a [Trigger](#) to set a property value, but note that the [Trigger](#) class also has the [EnterActions](#) and [ExitActions](#) properties that enable a trigger to perform actions.

Notice that the [MaxHeight](#) property of the [ListBoxItem](#) is set to `75`. In the following illustration, the third item is the selected item.



EventTriggers and Storyboards

Another type of trigger is the [EventTrigger](#), which starts a set of actions based on the occurrence of an event. For example, the following [EventTrigger](#) objects specify that when the mouse pointer enters the [ListBoxItem](#), the [MaxHeight](#) property animates to a value of `90` over a `0.2` second period. When the mouse moves away from the item, the property returns to the original value over a period of `1` second. Note how it is not necessary to specify a [To](#) value for the [MouseLeave](#) animation. This is because the animation is able to keep track of the original value.

```

<Style.Triggers>
  <Trigger Property="IsSelected" Value="True">
    <Trigger.Setters>
      <Setter Property="Opacity" Value="1.0" />
    </Trigger.Setters>
  </Trigger>
  <EventTrigger RoutedEvent="Mouse.MouseEnter">
    <EventTrigger.Actions>
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Duration="0:0:0.2"
            Storyboard.TargetProperty="MaxHeight"
            To="90" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
  <EventTrigger RoutedEvent="Mouse.MouseLeave">
    <EventTrigger.Actions>
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Duration="0:0:1"
            Storyboard.TargetProperty="MaxHeight" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
</Style.Triggers>

```

For more information, see the [Storyboards overview](#).

In the following illustration, the mouse is pointing to the third item.



MultiTriggers, DataTriggers, and MultiDataTriggers

In addition to [Trigger](#) and [EventTrigger](#), there are other types of triggers. [MultiTrigger](#) allows you to set property values based on multiple conditions. You use [DataTrigger](#) and [MultiDataTrigger](#) when the property of your condition is data-bound.

Visual States

Controls are always in a specific **state**. For example, when the mouse moves over the surface of a control, the control is considered to be in a common state of `MouseOver`. A control without a specific state is considered to be in the common `Normal` state. States are broken into groups, and the previously mentioned states are part of the state group `CommonStates`. Most controls have two state groups: `CommonStates` and `FocusStates`. Of each state group applied to a control, a control is always in one state of each group, such as `CommonStates.MouseOver` and `FocusStates.Unfocused`. However, a control can't be in two different states within the same group, such as `CommonStates.Normal` and `CommonStates.Disabled`. Here is a table of states most controls recognize and use.

VISUALSTATE NAME	VISUALSTATEGROUP NAME	DESCRIPTION
Normal	CommonStates	The default state.
MouseOver	CommonStates	The mouse pointer is positioned over the control.
Pressed	CommonStates	The control is pressed.
Disabled	CommonStates	The control is disabled.
Focused	FocusStates	The control has focus.
Unfocused	FocusStates	The control does not have focus.

By defining a [System.Windows.VisualStateManager](#) on the root element of a control template, you can trigger animations when a control enters a specific state. The `VisualStateManager` declares which combinations of [VisualStateGroup](#) and [VisualState](#) to watch. When the control enters a watched state, the animation defined by the `VisualStateManager` is started.

For example, the following XAML code watches the `CommonStates.MouseOver` state to animate the fill color of the element named `backgroundElement`. When the control returns to the `CommonStates.Normal` state, the fill color of the element named `backgroundElement` is restored.

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup Name="CommonStates">
        <VisualState Name="Normal">
          <ColorAnimation Storyboard.TargetName="backgroundElement"
                        Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
                        To="{TemplateBinding Background}"
                        Duration="0:0:0.3"/>
        </VisualState>
        <VisualState Name="MouseOver">
          <ColorAnimation Storyboard.TargetName="backgroundElement"
                        Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
                        To="Yellow"
                        Duration="0:0:0.3"/>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>

    ...
  </Grid>
</ControlTemplate>
```

For more information about storyboards, see [Storyboards Overview](#).

Shared resources and themes

A typical WPF app might have multiple UI resources that are applied throughout the app. Collectively, this set of resources can be considered the theme for the app. WPF provides support for packaging UI resources as a theme by using a resource dictionary that is encapsulated as the [ResourceDictionary](#) class.

WPF themes are defined by using the styling and templating mechanism that WPF exposes for customizing the visuals of any element.

WPF theme resources are stored in embedded resource dictionaries. These resource dictionaries must be embedded within a signed assembly, and can either be embedded in the same assembly as the code itself or in a

side-by-side assembly. For `PresentationFramework.dll`, the assembly that contains WPF controls, theme resources are in a series of side-by-side assemblies.

The theme becomes the last place to look when searching for the style of an element. Typically, the search will begin by walking up the element tree searching for an appropriate resource, then look in the app resource collection and finally query the system. This gives app developers a chance to redefine the style for any object at the tree or app level before reaching the theme.

You can define resource dictionaries as individual files that enable you to reuse a theme across multiple apps. You can also create swappable themes by defining multiple resource dictionaries that provide the same types of resources but with different values. Redefining these styles or other resources at the app level is the recommended approach for skinning an app.

To share a set of resources, including styles and templates, across apps, you can create a XAML file and define a [ResourceDictionary](#) that includes reference to a `shared.xaml` file.

```
<ResourceDictionary.MergedDictionaries>
  <ResourceDictionary Source="Shared.xaml" />
</ResourceDictionary.MergedDictionaries>
```

It is the sharing of `shared.xaml`, which itself defines a [ResourceDictionary](#) that contains a set of style and brush resources, that enables the controls in an app to have a consistent look.

For more information, see [Merged resource dictionaries](#).

If you are creating a theme for your custom control, see the **Defining resources at the theme level** section of the [Control authoring overview](#).

See also

- [Pack URIs in WPF](#)
- [How to: Find ControlTemplate-Generated Elements](#)
- [Find DataTemplate-Generated Elements](#)

How to create a style for a control (WPF .NET)

4/15/2021 • 6 minutes to read • [Edit Online](#)

With Windows Presentation Foundation (WPF), you can customize an existing control's appearance with your own reusable style. Styles can be applied globally to your app, windows and pages, or directly to controls.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Create a style

You can think of a [Style](#) as a convenient way to apply a set of property values to one or more elements. You 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. Because styles are resources, they obey the same scoping rules that apply to all resources. Put simply, where you declare a style affects where the style can be applied. For example, if you declare the style in the root element of your app definition XAML file, the style can be used anywhere in your app.

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

      <Style x:Key="Header1" TargetType="TextBlock">
        <Setter Property="FontSize" Value="15" />
        <Setter Property="FontWeight" Value="ExtraBold" />
      </Style>

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

If you declare the style in one of the app's XAML files, the style can be used only in that XAML file. For more information about scoping rules for resources, see [Overview of XAML resources](#).

```

<Window x:Class="IntroToStylingAndTemplating.WindowSingleResource"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="WindowSingleResource" Height="450" Width="800">
    <Window.Resources>

        <Style x:Key="Header1" TargetType="TextBlock">
            <Setter Property="FontSize" Value="15" />
            <Setter Property="FontWeight" Value="ExtraBold" />
        </Style>

    </Window.Resources>
</Grid />
</Window>

```

A style is made up of `<Setter>` child elements that set properties on the elements the style is applied to. In the example above, notice that the style is set to apply to `TextBlock` types through the `TargetType` attribute. The style will set the `FontSize` to `15` and the `FontWeight` to `ExtraBold`. Add a `<Setter>` for each property the style changes.

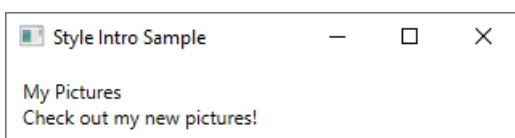
Apply a style implicitly

A `Style` is a convenient way to apply a set of property values to more than one element. For example, consider the following `TextBlock` elements and their default appearance in a window.

```

<StackPanel>
    <TextBlock>My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>

```



You can change the default appearance by setting properties, such as `FontSize` and `FontFamily`, on each `TextBlock` element directly. However, if you want your `TextBlock` elements to share some properties, you can create a `Style` in the `Resources` section of your XAML file, as shown here.

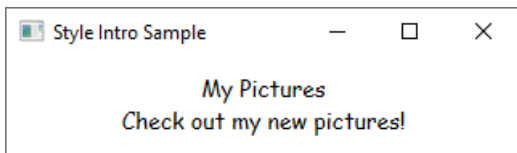
```

<Window.Resources>
    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>

```

When you set the `TargetType` of your style to the `TextBlock` type and omit the `x:Key` attribute, the style is applied to all the `TextBlock` elements scoped to the style, which is generally the XAML file itself.

Now the `TextBlock` elements appear as follows.



Apply a style explicitly

If you add an `x:Key` attribute with value to the style, the style is no longer implicitly applied to all elements of `TargetType`. Only elements that explicitly reference the style will have the style applied to them.

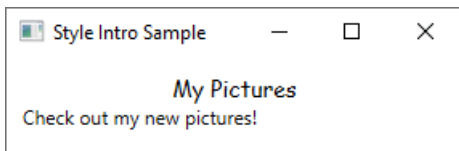
Here is the style from the previous section, but declared with the `x:Key` attribute.

```
<Window.Resources>
  <Style x:Key="TitleText" TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
  </Style>
</Window.Resources>
```

To apply the style, set the `Style` property on the element to the `x:Key` value, using a [StaticResource markup extension](#), as shown here.

```
<StackPanel>
  <TextBlock Style="{StaticResource TitleText}">My Pictures</TextBlock>
  <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Notice that the first `TextBlock` element has the style applied to it while the second `TextBlock` element remains unchanged. The implicit style from the previous section was changed to a style that declared the `x:Key` attribute, meaning, the only element affected by the style is the one that referenced the style directly.



Once a style is applied, explicitly or implicitly, it becomes sealed and can't be changed. If you want to change a style that has been applied, create a new style to replace the existing one. For more information, see the [IsSealed](#) property.

You can create an object that chooses a style to apply based on custom logic. For an example, see the example provided for the [StyleSelector](#) class.

Apply a style programmatically

To assign a named style to an element programmatically, get the style from the resources collection and assign it to the element's `Style` property. The items in a resources collection are of type `Object`. Therefore, you must cast the retrieved style to a `System.Windows.Style` before assigning it to the `style` property. For example, the following code sets the style of a `TextBlock` named `textblock1` to the defined style `TitleText`.

```
textblock1.Style = (Style)Resources["TitleText"];
```

```
textblock1.Style = CType(Resources("TitleText"), Windows.Style)
```

Extend a style

Perhaps you want your two [TextBlock](#) elements to share some property values, such as the [FontFamily](#) and the centered [HorizontalAlignment](#). But you also want the text **My Pictures** to have some additional properties. You can do that by creating a new style that is based on the first style, as shown here.

```
<Window.Resources>
    <!-- .... other resources .... -->

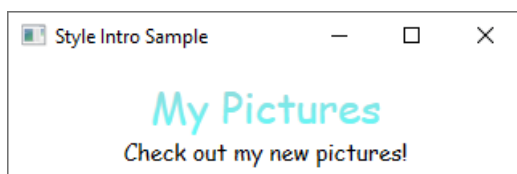
    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>

    <!--A Style that extends the previous TextBlock Style with an x:Key of TitleText-->
    <Style BasedOn="{StaticResource {x:Type TextBlock}}"
        TargetType="TextBlock"
        x:Key="TitleText">
        <Setter Property="FontSize" Value="26"/>
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="#90DDDD" />
                        <GradientStop Offset="1.0" Color="#5BFFFF" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
```

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

This `TextBlock` style is now centered, uses a `Comic Sans MS` font with a size of `26`, and the foreground color set to the [LinearGradientBrush](#) shown in the example. Notice that it overrides the [FontSize](#) value of the base style. If there's more than one [Setter](#) pointing to the same property in a [Style](#), the [Setter](#) that is declared last takes precedence.

The following shows what the [TextBlock](#) elements now look like:



This `TitleText` style extends the style that has been created for the [TextBlock](#) type, referenced with `BasedOn="{StaticResource {x:Type TextBlock}}"`. You can also extend a style that has an `x:Key` by using the `x:Key` of the style. For example, if there was a style named `Header1` and you wanted to extend that style, you would use `BasedOn="{StaticResource Header1}"`.

Relationship of the TargetType property and the x:Key attribute

As previously shown, setting the [TargetType](#) property to `TextBlock` without assigning the style an `x:Key` causes the style to be applied to all `TextBlock` elements. In this case, the `x:Key` is implicitly set to `{x:Type TextBlock}`. This means that if you explicitly set the `x:Key` value to anything other than `{x:Type TextBlock}`, the [Style](#) isn't applied to all `TextBlock` elements automatically. Instead, you must apply the style (by using the `x:Key` value) to the `TextBlock` elements explicitly. If your style is in the resources section and you don't set the `TargetType` property on your style, then you must set the `x:Key` attribute.

In addition to providing a default value for the `x:Key`, the `TargetType` property specifies the type to which setter properties apply. If you don't specify a `TargetType`, you must qualify the properties in your [Setter](#) objects with a class name by using the syntax `Property="ClassName.Property"`. For example, instead of setting `Property="FontSize"`, you must set [Property](#) to `"TextBlock.FontSize"` or `"Control.FontSize"`.

Also note that many WPF controls consist of a combination of other WPF controls. If you create a style that applies to all controls of a type, you might get unexpected results. For example, if you create a style that targets the `TextBlock` type in a [Window](#), the style is applied to all `TextBlock` controls in the window, even if the `TextBlock` is part of another control, such as a [ListBox](#).

See also

- [How to create a template for a control](#)
- [Overview of XAML resources](#)
- [XAML overview](#)

How to create a template for a control (WPF.NET)

4/15/2021 • 7 minutes to read • [Edit Online](#)

With Windows Presentation Foundation (WPF), you can customize an existing control's visual structure and behavior with your own reusable template. Templates can be applied globally to your application, windows and pages, or directly to controls. Most scenarios that require you to create a new control can be covered by instead creating a new template for an existing control.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

In this article, you'll explore creating a new [ControlTemplate](#) for the [Button](#) control.

When to create a ControlTemplate

Controls have many properties, such as [Background](#), [Foreground](#), and [FontFamily](#). These properties control different aspects of the control's appearance, but the changes that you can make by setting these properties are limited. For example, you can set the [Foreground](#) property to blue and [FontStyle](#) to italic on a [CheckBox](#). When you want to customize the control's appearance beyond what setting the other properties on the control can do, you create a [ControlTemplate](#).

In most user interfaces, a button has the same general appearance: a rectangle with some text. If you wanted to create a rounded button, you could create a new control that inherits from the button or recreates the functionality of the button. In addition, the new user control would provide the circular visual.

You can avoid creating new controls by customizing the visual layout of an existing control. With a rounded button, you create a [ControlTemplate](#) with the desired visual layout.

On the other hand, if you need a control with new functionality, different properties, and new settings, you would create a new [UserControl](#).

Prerequisites

Create a new WPF application and in *MainWindow.xaml* (or another window of your choice) set the following properties on the **<Window>** element:

PROPERTY	VALUE
Title	<code>Template Intro Sample</code>
SizeToContent	<code>WidthAndHeight</code>
MinWidth	<code>250</code>

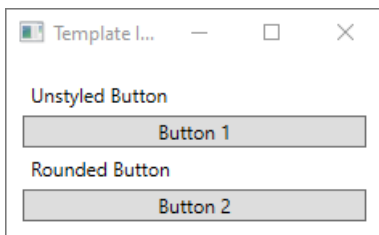
Set the content of the **<Window>** element to the following XAML:

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

In the end, the *MainWindow.xaml* file should look similar to the following:

```
<Window x:Class="IntroToStylingAndTemplating.Window1"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight" MinWidth="250">
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

If you run the application, it looks like the following:



Create a ControlTemplate

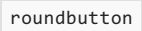

The most common way to declare a [ControlTemplate](#) is as a resource in the `Resources` section in a XAML file. Because templates are resources, they obey the same scoping rules that apply to all resources. Put simply, where you declare a template affects where the template can be applied. For example, if you declare the template in the root element of your application definition XAML file, the template can be used anywhere in your application. If you define the template in a window, only the controls in that window can use the template.

To start with, add a `Window.Resources` element to your *MainWindow.xaml* file:


```
<Window x:Class="IntroToStylingAndTemplating.Window2"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight" MinWidth="250">
    <Window.Resources>

    </Window.Resources>
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

Create a new **<ControlTemplate>** with the following properties set:

PROPERTY	VALUE
x:Key	
TargetType	

This control template will be simple:

- a root element for the control, a [Grid](#)
- an [Ellipse](#) to draw the rounded appearance of the button
- a [ContentPresenter](#) to display the user-specified button content

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

TemplateBinding

When you create a new [ControlTemplate](#), you still might want to use the public properties to change the control's appearance. The [TemplateBinding](#) markup extension binds a property of an element that is in the [ControlTemplate](#) to a public property that is defined by the control. When you use a [TemplateBinding](#), you enable properties on the control to act as parameters to the template. That is, when a property on a control is set, that value is passed on to the element that has the [TemplateBinding](#) on it.

Ellipse

Notice that the **Fill** and **Stroke** properties of the **<Ellipse>** element are bound to the control's [Foreground](#) and [Background](#) properties.

ContentPresenter

A **<ContentPresenter>** element is also added to the template. Because this template is designed for a button, take into consideration that the button inherits from [ContentControl](#). The button presents the content of the element. You can set anything inside of the button, such as plain text or even another control. Both of the following are valid buttons:

```

<Button>My Text</Button>

<!-- and -->

<Button>
    <CheckBox>Checkbox in a button</CheckBox>
</Button>

```

In both of the previous examples, the text and the checkbox are set as the [Button.Content](#) property. Whatever is set as the content can be presented through a [ContentPresenter](#), which is what the template does.

If the [ControlTemplate](#) is applied to a [ContentControl](#) type, such as a [Button](#), a [ContentPresenter](#) is searched for in the element tree. If the [ContentPresenter](#) is found, the template automatically binds the control's [Content](#) property to the [ContentPresenter](#).

Use the template

Find the buttons that were declared at the start of this article.

```

<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>

```

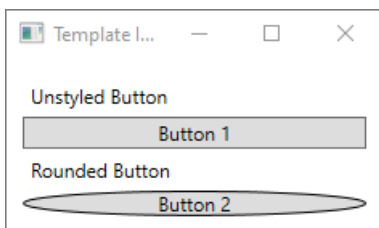
Set the second button's [Template](#) property to the [roundbutton](#) resource:

```

<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}">Button 2</Button>
</StackPanel>

```

If you run the project and look at the result, you'll see that the button has a rounded background.

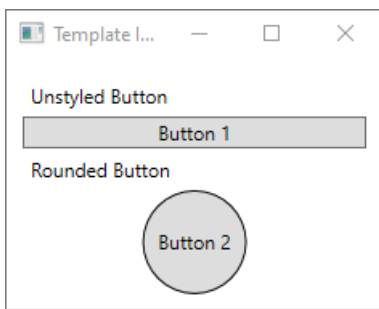


You may have noticed that the button isn't a circle but is skewed. Because of the way the [Ellipse](#) element works, it always expands to fill the available space. Make the circle uniform by changing the button's **width** and **height** properties to the same value:

```

<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}" Width="65" Height="65">Button 2</Button>
</StackPanel>

```



Add a Trigger

Even though a button with a template applied looks different, it behaves the same as any other button. If you press the button, the [Click](#) event fires. However, you may have noticed that when you move your mouse over the button, the button's visuals don't change. These visual interactions are all defined by the template.

With the dynamic event and property systems that WPF provides, you can watch a specific property for a value and then restyle the template when appropriate. In this example, you'll watch the button's [IsMouseOver](#) property. When the mouse is over the control, style the `<Ellipse>` with a new color. This type of trigger is known as a *PropertyTrigger*.

For this to work, you'll need to add a name to the `<Ellipse>` that you can reference. Give it the name of **backgroundElement**.

```
<Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
```

Next, add a new [Trigger](#) to the [ControlTemplate.Triggers](#) collection. The trigger will watch the `IsMouseOver` event for the value `true`.

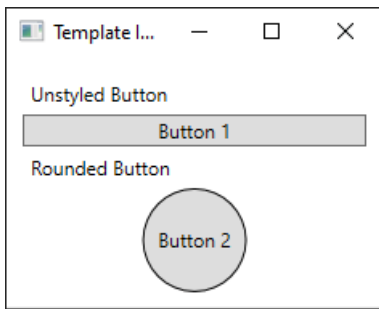
```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="true">

            </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

Next, add a `<Setter>` to the `<Trigger>` that changes the `Fill` property of the `<Ellipse>` to a new color.

```
<Trigger Property="IsMouseOver" Value="true">
    <Setter Property="Fill" TargetName="backgroundElement" Value="AliceBlue"/>
</Trigger>
```

Run the project. Notice that when you move the mouse over the button, the color of the `<Ellipse>` changes.



Use a VisualState

Visual states are defined and triggered by a control. For example, when the mouse is moved on top of the control, the `CommonStates.MouseOver` state is triggered. You can animate property changes based on the current state of the control. In the previous section, a `<PropertyTrigger>` was used to change the foreground of the button to `AliceBlue` when the `IsMouseOver` property was `true`. Instead, create a visual state that animates the change of this color, providing a smooth transition. For more information about *VisualStates*, see [Styles and templates in WPF](#).

To convert the `<PropertyTrigger>` to an animated visual state, First, remove the `<ControlTemplate.Triggers>` element from your template.

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

Next, in the `<Grid>` root of the control template, add the `<VisualStateManager.VisualStateGroups>` element with a `<VisualStateGroup>` for `CommonStates`. Define two states, `Normal` and `MouseOver`.

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
                    </VisualState>
                <VisualState Name="MouseOver">
                    </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

Any animations defined in a `<VisualState>` are applied when that state is triggered. Create animations for each state. Animations are put inside of a `<Storyboard>` element. For more information about storyboards, see [Storyboards Overview](#).

- Normal

This state animates the ellipse fill, restoring it to the control's `Background` color.

```

<Storyboard>
  <ColorAnimation Storyboard.TargetName="backgroundElement"
    Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
    To="{TemplateBinding Background}"
    Duration="0:0:0.3"/>
</Storyboard>

```

- MouseOver

This state animates the ellipse `Background` color to a new color: `Yellow`.

```

<Storyboard>
  <ColorAnimation Storyboard.TargetName="backgroundElement"
    Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
    To="Yellow"
    Duration="0:0:0.3"/>
</Storyboard>

```

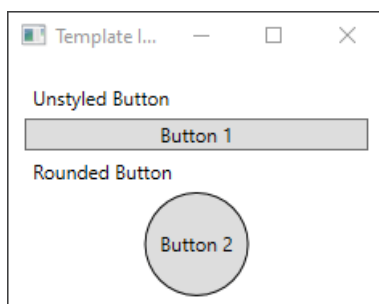
The `<ControlTemplate>` should now look like the following.

```

<ControlTemplate x:Key="roundbutton" TargetType="Button">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup Name="CommonStates">
        <VisualState Name="Normal">
          <Storyboard>
            <ColorAnimation Storyboard.TargetName="backgroundElement"
              Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
              To="{TemplateBinding Background}"
              Duration="0:0:0.3"/>
          </Storyboard>
        </VisualState>
        <VisualState Name="MouseOver">
          <Storyboard>
            <ColorAnimation Storyboard.TargetName="backgroundElement"
              Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
              To="Yellow"
              Duration="0:0:0.3"/>
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Ellipse Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
    <ContentPresenter x:Name="contentPresenter" HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
</ControlTemplate>

```

Run the project. Notice that when you move the mouse over the button, the color of the `<Ellipse>` animates.



Next steps

- [Create a style for a control](#)
- [Styles and templates](#)
- [Overview of XAML resources](#)

Data binding overview (WPF .NET)

5/1/2021 • 39 minutes to read • [Edit Online](#)

Data binding in Windows Presentation Foundation (WPF) provides a simple and consistent way for apps to present and interact with data. Elements can be bound to data from different kinds of data sources in the form of .NET objects and XML. Any [ContentControl](#) such as [Button](#) and any [ItemsControl](#), such as [ListBox](#) and [ListView](#), have built-in functionality to enable flexible styling of single data items or collections of data items. Sort, filter, and group views can be generated on top of the data.

The data binding in WPF has several advantages over traditional models, including inherent support for data binding by a broad range of properties, flexible UI representation of data, and clean separation of business logic from UI.

This article first discusses concepts fundamental to WPF data binding and then covers the usage of the [Binding](#) class and other features of data binding.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

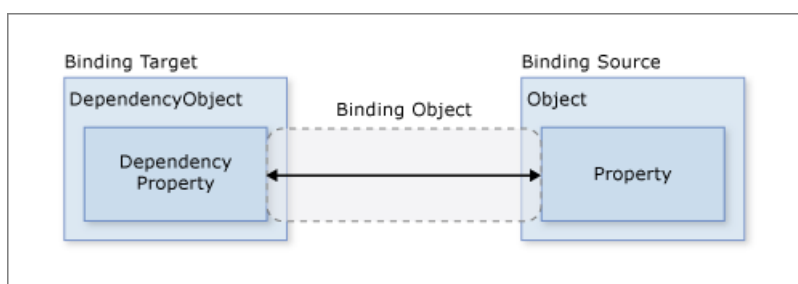
What is data binding?

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 different kinds of data sources. In WPF, dependency properties of elements can be bound to .NET objects (including ADO.NET objects or objects associated with Web Services and Web properties) and XML data.

Basic data binding concepts

Regardless of what element you're binding and the nature of your data source, each binding always follows the model illustrated by the following figure.



As the figure shows, data binding is essentially the bridge between your binding target and your binding source. The figure demonstrates the following fundamental WPF data binding concepts:

- 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.

For example, if you bound the content of a `TextBox` to the `Employee.Name` property, you would set up your binding like the following table:

SETTING	VALUE
Target	<code>TextBox</code>
Target property	<code>Text</code>
Source object	<code>Employee</code>
Source object value path	<code>Name</code>

- The target property must be a dependency property.

Most `UIElement` properties are dependency properties, and most dependency properties, except read-only ones, support data binding by default. Only types derived from `DependencyObject` can define dependency properties. All `UIElement` types derive from `DependencyObject`.

- Binding sources aren't restricted to custom .NET objects.

Although not shown in the figure, it should be noted that the binding source object isn't restricted to being a custom .NET object. WPF data binding supports data in the form of .NET objects, XML, and even XAML element objects. To provide some examples, your binding source may be a `UIElement`, any list object, an ADO.NET or Web Services object, or an `XmlNode` that contains your XML data. For more information, see [Binding sources overview](#).

It's important to remember that when you're establishing a binding, you're binding a binding target *to* a binding source. For example, if you're displaying some underlying XML data in a `ListBox` using data binding, you're binding your `ListBox` to the XML data.

To establish a binding, you use the `Binding` object. The rest of this article discusses many of the concepts associated with and some of the properties and usage of the `Binding` object.

Data context

When data binding is declared on XAML elements, they resolve data binding by looking at their immediate `DataContext` property. The data context is typically the **binding source object** for the **binding source value path** evaluation. You can override this behavior in the binding and set a specific **binding source object** value. If the `DataContext` property for the object hosting the binding isn't set, the parent element's `DataContext` property is checked, and so on, up until the root of the XAML object tree. In short, the data context used to resolve binding is inherited from the parent unless explicitly set on the object.

Bindings can be configured to resolve with a specific object, as opposed to using the data context for binding resolution. Specifying a source object directly is used when, for example, you bind the foreground color of an object to the background color of another object. Data context isn't needed since the binding is resolved between those two objects. Inversely, bindings that aren't bound to specific source objects use data-context resolution.

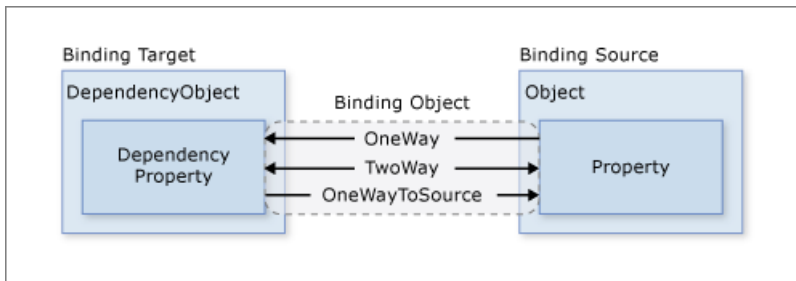
When the `DataContext` property changes, all bindings that could be affected by the data context are reevaluated.

Direction of the data flow

As indicated by the arrow in the previous figure, the data flow of a binding can go from the binding target to the binding source (for example, the source value changes when a user edits the value of a `TextBox`) and/or from the binding source to the binding target (for example, your `TextBox` content is updated with changes in the binding source) if the binding source provides the proper notifications.

You may want your app to enable users to change the data and propagate it back to the source object. Or you may not want to enable users to update the source data. You can control the flow of data by setting the [Binding.Mode](#).

This figure illustrates the different types of data flow:



- **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. For instance, you may bind to a source such as a stock ticker, or perhaps your target property has no control interface provided for making changes, such as a data-bound background color of a table. If there's no need to monitor the changes of the target property, using the **OneWay** binding mode avoids the overhead of the **TwoWay** binding mode.
- **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. A programmatic way to determine whether a dependency property binds one-way or two-way by default is to get the property metadata with `DependencyProperty.GetMetadata` and then check the Boolean value of the `FrameworkPropertyMetadata.BindsTwoWayByDefault` property.
- **OneWayToSource** is the reverse of **OneWay** binding; it updates the source property when the target property changes. One example scenario is if you only need to reevaluate the source value from the UI.
- Not illustrated in the figure is **OneTime** binding, which causes the source property to initialize the target property but doesn't propagate subsequent changes. If the data context changes or the object in the data context changes, the change is *not* reflected in the target property. This type of binding is appropriate if either a snapshot of the current state is appropriate or the data is truly static. This type of binding is also useful if you want to initialize your target property with some value from a source property and the data context isn't known in advance. This mode is essentially a simpler form of **OneWay** binding that provides better performance in cases where the source value doesn't change.

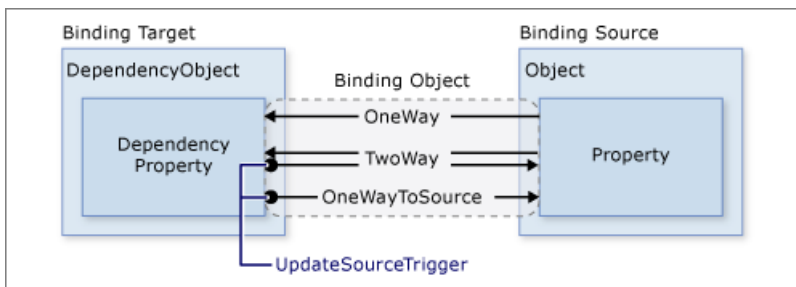
To detect source changes (applicable to **OneWay** and **TwoWay** bindings), the source must implement a suitable property change notification mechanism such as `INotifyPropertyChanged`. See [How to: Implement property change notification \(.NET Framework\)](#) for an example of an `INotifyPropertyChanged` implementation.

The [Binding.Mode](#) property provides more information about binding modes and an example of how to specify the direction of a binding.

What triggers source updates

Bindings that are **TwoWay** or **OneWayToSource** listen for changes in the target property and propagate them back to the source, known as updating the source. For example, you may edit the text of a `TextBox` to change the underlying source value.

However, is your source value updated while you're editing the text or after you finish editing the text and the control loses focus? The [Binding.UpdateSourceTrigger](#) property determines what triggers the update of the source. The dots of the right arrows in the following figure illustrate the role of the [Binding.UpdateSourceTrigger](#) property.



If the `UpdateSourceTrigger` value is [UpdateSourceTrigger.PropertyChanged](#), then the value pointed to by the right arrow of `TwoWay` or the `OneWayToSource` bindings is updated as soon as the target property changes. However, if the `UpdateSourceTrigger` value is [LostFocus](#), then that value only is updated with the new value when the target property loses focus.

Similar to the [Mode](#) property, different dependency properties have different default `UpdateSourceTrigger` values. The default value for most dependency properties is [PropertyChanged](#), which causes the source property's value to instantly change when the target property value is changed. Instant changes are fine for [CheckBox](#) and other simple controls. However, for text fields, updating after every keystroke can diminish performance and denies the user the usual opportunity to backspace and fix typing errors before committing to the new value. For example, the `TextBox.Text` property defaults to the `UpdateSourceTrigger` value of [LostFocus](#), which causes the source value to change only when the control element loses focus, not when the `TextBox.Text` property is changed. See the [UpdateSourceTrigger](#) property page for information about how to find the default value of a dependency property.

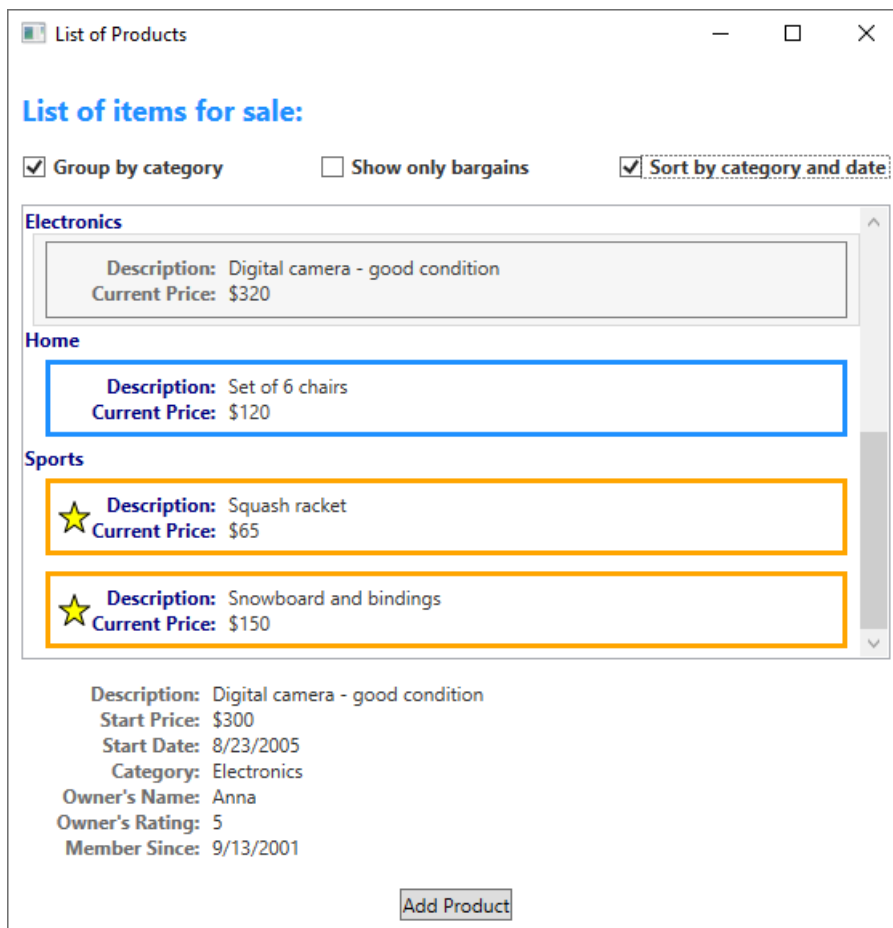
The following table provides an example scenario for each `UpdateSourceTrigger` value using the `TextBox` as an example.

UPDATESOURCETRIGGER VALUE	WHEN THE SOURCE VALUE IS UPDATED	EXAMPLE SCENARIO FOR TEXTBOX
<code>LostFocus</code> (default for <code>TextBox.Text</code>)	When the <code>TextBox</code> control loses focus.	A <code>TextBox</code> that is associated with validation logic (see Data Validation below).
<code>PropertyChanged</code>	As you type into the <code>TextBox</code> .	<code>TextBox</code> controls in a chat room window.
<code>Explicit</code>	When the app calls <code>UpdateSource</code> .	<code>TextBox</code> controls in an editable form (updates the source values only when the user presses the submit button).

For an example, see [How to: Control when the TextBox text updates the source \(.NET Framework\)](#).

Example of data binding

For an example of data binding, take a look at the following app UI from the [Data Binding Demo](#), which displays a list of auction items.



The app demonstrates the following features of data binding:

- The content of the `ListBox` is bound to a collection of *AuctionItem* objects. An *AuctionItem* object has properties such as *Description*, *StartPrice*, *StartDate*, *Category*, and *SpecialFeatures*.
- The data (*AuctionItem* objects) displayed in the `ListBox` is templated so that the description and the current price are shown for each item. The template is created by using a [DataTemplate](#). In addition, the appearance of each item depends on the *SpecialFeatures* value of the *AuctionItem* being displayed. If the *SpecialFeatures* value of the *AuctionItem* is *Color*, the item has a blue border. If the value is *Highlight*, the item has an orange border and a star. The [Data Templating](#) section provides information about data templating.
- The user can group, filter, or sort the data using the `CheckBoxes` provided. In the image above, the **Group by category** and **Sort by category and date** `CheckBoxes` are selected. You may have noticed that the data is grouped based on the category of the product, and the category name is in alphabetical order. It's difficult to notice from the image but the items are also sorted by the start date within each category. Sorting is done using a *collection view*. The [Binding to collections](#) section discusses collection views.
- When the user selects an item, the `ContentControl` displays the details of the selected item. This experience is called the *Master-detail scenario*. The [Master-detail scenario](#) section provides information about this type of binding.
- The type of the *StartDate* property is `DateTime`, which returns a date that includes the time to the millisecond. In this app, a custom converter has been used so that a shorter date string is displayed. The [Data conversion](#) section provides information about converters.

When the user selects the *Add Product* button, the following form comes up.

Add Product Listing

Item for sale:

Item Description:

Start Price:

Start Date:

Category:

Special Features:

★ **Description:** Brand New Computer
Current Price: \$650

Description: Brand New Computer
Start Price: \$650
Start Date: 9/28/2019
Category: Computers
Owner's Name: John
Owner's Rating: 12
Member Since: 4/20/2003

The user can edit the fields in the form, preview the product listing using the short or detailed preview panes, and select to add the new product listing. Any existing grouping, filtering and sorting settings will apply to the new entry. In this particular case, the item entered in the above image will be displayed as the second item within the *Computer* category.

Not shown in this image is the validation logic provided in the *Start Date* [TextBox](#). If the user enters an invalid date (invalid formatting or a past date), the user will be notified with a [ToolTip](#) and a red exclamation point next to the [TextBox](#). The [Data Validation](#) section discusses how to create validation logic.

Before going into the different features of data binding outlined above, we will first discuss the fundamental concepts that are critical to understanding WPF data binding.

Create a binding

To restate some of the concepts discussed in the previous sections, you establish a binding using the [Binding](#) object, and each binding usually has four components: a binding target, a target property, a binding source, and a path to the source value to use. This section discusses how to set up a binding.

Binding sources are tied to the active [DataContext](#) for the element. Elements automatically inherit their if they've not explicitly defined one.

Consider the following example, in which the binding source object is a class named *MyData* that is defined in the *SDKSample* namespace. For demonstration purposes, *MyData* has a string property named *ColorName* whose value is set to "Red". Thus, this example generates a button with a red background.

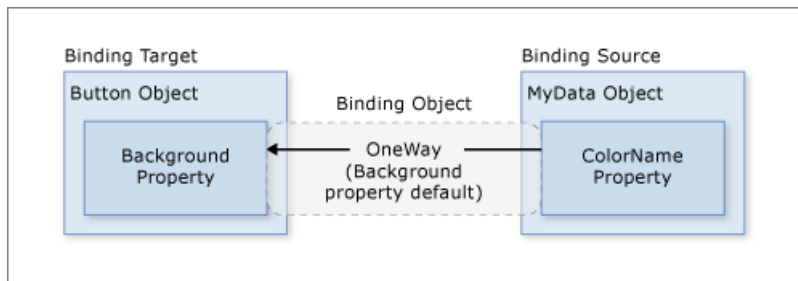
```

<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
    <DockPanel.DataContext>
        <Binding Source="{StaticResource myDataSource}"/>
    </DockPanel.DataContext>
    <Button Background="{Binding Path=ColorName}"
            Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>

```

For more information on the binding declaration syntax and examples of how to set up a binding in code, see [Binding declarations overview](#).

If we apply this example to our basic diagram, the resulting figure looks like the following. This figure describes a [OneWay](#) binding because the Background property supports [OneWay](#) binding by default.



You may wonder why this binding works even though the *ColorName* property is of type string while the [Background](#) property is of type [Brush](#). This binding uses default type conversion, which is discussed in the [Data conversion](#) section.

Specifying the binding source

Notice that in the previous example, the binding source is specified by setting the [DockPanel.DataContext](#) property. The [Button](#) then inherits the [DataContext](#) value from the [DockPanel](#), which is its parent element. To reiterate, the binding source object is one of the four necessary components of a binding. So, without the binding source object being specified, the binding would do nothing.

There are several ways to specify the binding source object. Using the [DataContext](#) property on a parent element is useful when you're binding multiple properties to the same source. However, sometimes it may be more appropriate to specify the binding source on individual binding declarations. For the previous example, instead of using the [DataContext](#) property, you can specify the binding source by setting the [Binding.Source](#) property directly on the binding declaration of the button, as in the following example.

```

<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
    <Button Background="{Binding Source={StaticResource myDataSource}, Path=ColorName}"
            Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>

```

Other than setting the [DataContext](#) property on an element directly, inheriting the [DataContext](#) value from an

ancestor (such as the button in the first example), and explicitly specifying the binding source by setting the [Binding.Source](#) property on the binding (such as the button the last example), you can also use the [Binding.ElementName](#) property or the [Binding.RelativeSource](#) property to specify the binding source. The [ElementName](#) property is useful when you're binding to other elements in your app, such as when you're using a slider to adjust the width of a button. The [RelativeSource](#) property is useful when the binding is specified in a [ControlTemplate](#) or a [Style](#). For more information, see [Binding sources overview](#).

Specifying the path to the value

If your binding source is an object, you use the [Binding.Path](#) property to specify the value to use for your binding. If you're binding to XML data, you use the [Binding.XPath](#) property to specify the value. In some cases, it may be applicable to use the [Path](#) property even when your data is XML. For example, if you want to access the Name property of a returned XmlNode (as a result of an XPath query), you should use the [Path](#) property in addition to the [XPath](#) property.

For more information, see the [Path](#) and [XPath](#) properties.

Although we have emphasized that the [Path](#) to the value to use is one of the four necessary components of a binding, in the scenarios that you want to bind to an entire object, the value to use would be the same as the binding source object. In those cases, it's applicable to not specify a [Path](#). Consider the following example.

```
<ListBox ItemsSource="{Binding}"
        IsSynchronizedWithCurrentItem="true"/>
```

The above example uses the empty binding syntax: {Binding}. In this case, the [ListBox](#) inherits the DataContext from a parent DockPanel element (not shown in this example). When the path isn't specified, the default is to bind to the entire object. In other words, in this example, the path has been left out because we are binding the [ItemsSource](#) property to the entire object. (See the [Binding to collections](#) section for an in-depth discussion.)

Other than binding to a collection, this scenario is also useful when you want to bind to an entire object instead of just a single property of an object. For example, if your source object is of type [String](#), you may simply want to bind to the string itself. Another common scenario is when you want to bind an element to an object with several properties.

You may need to apply custom logic so that the data is meaningful to your bound target property. The custom logic may be in the form of a custom converter if default type conversion doesn't exist. See [Data conversion](#) for information about converters.

Binding and BindingExpression

Before getting into other features and usages of data binding, it's useful to introduce the [BindingExpression](#) class. As you have seen in previous sections, the [Binding](#) class is the high-level class for the declaration of a binding; it provides many properties that allow you to specify the characteristics of a binding. A related class, [BindingExpression](#), is the underlying object that maintains the connection between the source and the target. A binding contains all the information that can be shared across several binding expressions. A [BindingExpression](#) is an instance expression that cannot be shared and contains all the instance information of the [Binding](#).

Consider the following example, where `myDataObject` is an instance of the `MyData` class, `myBinding` is the source [Binding](#) object, and `MyData` is a defined class that contains a string property named `ColorName`. This example binds the text content of `myText`, an instance of [TextBlock](#), to `ColorName`.

```
// Make a new source
var myDataObject = new MyData();
var myBinding = new Binding("ColorName")
{
    Source = myDataObject
};

// Bind the data source to the TextBox control's Text dependency property
myText.SetBinding(TextBox.TextProperty, myBinding);
```

```
' Make a New source
Dim myDataObject As New MyData
Dim myBinding As New Binding("ColorName")
myBinding.Source = myDataObject

' Bind the data source to the TextBox control's Text dependency property
myText.SetBinding(TextBox.TextProperty, myBinding)
```

You can use the same *myBinding* object to create other bindings. For example, you can use the *myBinding* object to bind the text content of a check box to *ColorName*. In that scenario, there will be two instances of [BindingExpression](#) sharing the *myBinding* object.

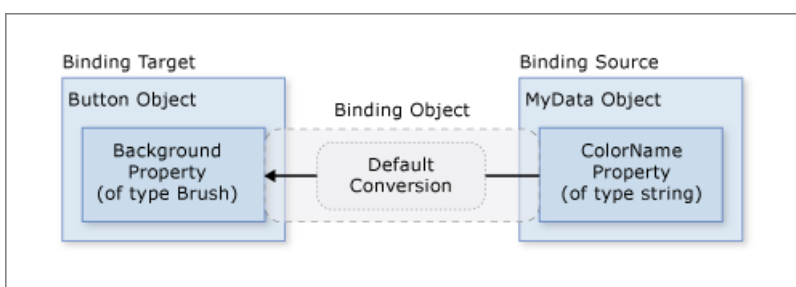
A [BindingExpression](#) object is returned by calling [GetBindingExpression](#) on a data-bound object. The following articles demonstrate some of the usages of the [BindingExpression](#) class:

- [Get the binding object from a bound target property \(.NET Framework\)](#)
- [Control When the TextBox text updates the source \(.NET Framework\)](#)

Data conversion

In the [Create a binding](#) section, the button is red because its [Background](#) property is bound to a string property with the value "Red". This string value works because a type converter is present on the [Brush](#) type to convert the string value to a [Brush](#).

Adding this information to the figure in the [Create a binding](#) section looks like this.



However, what if instead of having a property of type string your binding source object has a *Color* property of type [Color](#)? In that case, in order for the binding to work you would need to first turn the *Color* property value into something that the [Background](#) property accepts. You would need to create a custom converter by implementing the [IValueConverter](#) interface, as in the following example.

```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]
public class ColorBrushConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

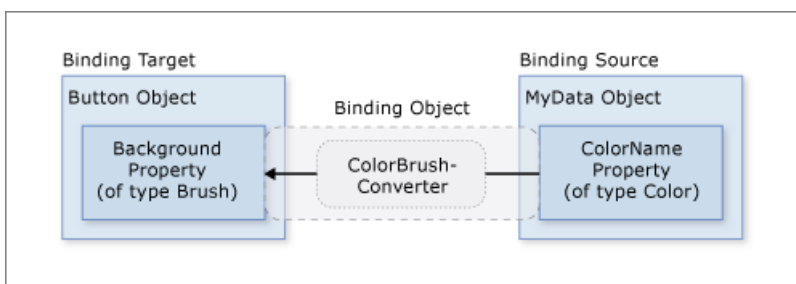
    public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        return null;
    }
}
```

```
<ValueConversion(GetType(Color), GetType(SolidColorBrush))>
Public Class ColorBrushConverter
    Implements IValueConverter
    Public Function Convert(ByVal value As Object, ByVal targetType As Type, ByVal parameter As Object,
        ByVal culture As System.Globalization.CultureInfo) As Object Implements IValueConverter.Convert
        Dim color As Color = CType(value, Color)
        Return New SolidColorBrush(color)
    End Function

    Public Function ConvertBack(ByVal value As Object, ByVal targetType As Type, ByVal parameter As Object,
        ByVal culture As System.Globalization.CultureInfo) As Object Implements IValueConverter.ConvertBack
        Return Nothing
    End Function
End Class
```

See [IValueConverter](#) for more information.

Now the custom converter is used instead of default conversion, and our diagram looks like this.



To reiterate, default conversions may be available because of type converters that are present in the type being bound to. This behavior will depend on which type converters are available in the target. If in doubt, create your own converter.

The following are some typical scenarios where it makes sense to implement a data converter:

- Your data should be displayed differently, depending on culture. For instance, you might want to implement a currency converter or a calendar date/time converter based on the conventions used in a particular culture.
- The data being used isn't necessarily intended to change the text value of a property, but is instead intended to change some other value, such as the source for an image, or the color or style of the display text. Converters can be used in this instance by converting the binding of a property that might not seem to be appropriate, such as binding a text field to the Background property of a table cell.
- More than one control or multiple properties of controls are bound to the same data. In this case, the

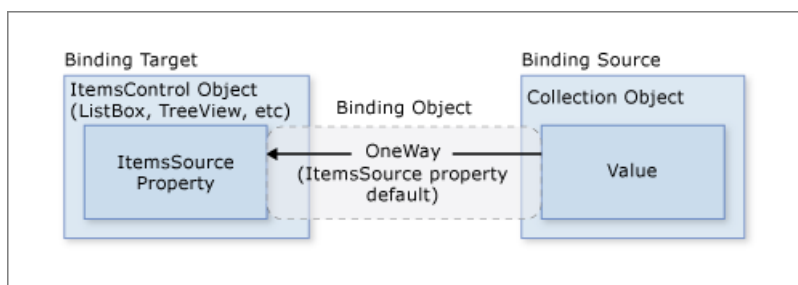
primary binding might just display the text, whereas other bindings handle specific display issues but still use the same binding as source information.

- A target property has a collection of bindings, which is termed [MultiBinding](#). For [MultiBinding](#), you use a custom [IMultiValueConverter](#) to produce a final value from the values of the bindings. For example, color may be computed from red, blue, and green values, which can be values from the same or different binding source objects. See [MultiBinding](#) for examples and information.

Binding to collections

A binding source object can be treated either as a single object whose properties contain data or as a data collection of polymorphic objects that are often grouped together (such as the result of a query to a database). So far we've only discussed binding to single objects. However, binding to a data collection is a common scenario. For example, a common scenario is to use an [ItemsControl](#) such as a [ListBox](#), [ListView](#), or [TreeView](#) to display a data collection, such as in the app shown in the [What is data binding](#) section.

Fortunately, our basic diagram still applies. If you're binding an [ItemsControl](#) to a collection, the diagram looks like this.



As shown in this diagram, to bind an [ItemsControl](#) to a collection object, [ItemsControl.ItemsSource](#) property is the property to use. You can think of `ItemsSource` as the content of the [ItemsControl](#). The binding is [OneWay](#) because the `ItemsSource` property supports `OneWay` binding by default.

How to implement collections

You can enumerate over any collection that implements the [IEnumerable](#) interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the [INotifyCollectionChanged](#) interface. This interface exposes an event that should be raised whenever the underlying collection changes.

WPF provides the [ObservableCollection<T>](#) class, which is a built-in implementation of a data collection that exposes the [INotifyCollectionChanged](#) interface. To fully support transferring data values from source objects to targets, each object in your collection that supports bindable properties must also implement the [INotifyPropertyChanged](#) interface. For more information, see [Binding sources overview](#).

Before implementing your own collection, consider using [ObservableCollection<T>](#) or one of the existing collection classes, such as [List<T>](#), [Collection<T>](#), and [BindingList<T>](#), among many others. If you have an advanced scenario and want to implement your own collection, consider using [IList](#), which provides a non-generic collection of objects that can be individually accessed by the index, and thus provides the best performance.

Collection views

Once your [ItemsControl](#) is bound to a data collection, you may want to sort, filter, or group the data. To do that, you use collection views, which are classes that implement the [ICollectionView](#) interface.

What Are collection views?

A collection view is a layer on top of a binding source collection that allows you to navigate and display the source collection based on sort, filter, and group queries, without having to change the underlying source collection itself. A collection view also maintains a pointer to the current item in the collection. If the source

collection implements the [INotifyCollectionChanged](#) interface, the changes raised by the [CollectionChanged](#) event are propagated to the views.

Because views do not change the underlying source collections, each source collection can have multiple views associated with it. For example, you may have a collection of *Task* objects. With the use of views, you can display that same data in different ways. For example, on the left side of your page you may want to show tasks sorted by priority, and on the right side, grouped by area.

How to create a view

One way to create and use a view is to instantiate the view object directly and then use it as the binding source. For example, consider the [Data binding demo](#) app shown in the [What is data binding](#) section. The app is implemented such that the [ListBox](#) binds to a view over the data collection instead of the data collection directly. The following example is extracted from the [Data binding demo](#) app. The [CollectionViewSource](#) class is the XAML proxy of a class that inherits from [CollectionView](#). In this particular example, the [Source](#) of the view is bound to the *AuctionItems* collection (of type [ObservableCollection<T>](#)) of the current app object.

```
<Window.Resources>
  <CollectionViewSource
    Source="{Binding Source={x:Static Application.Current}, Path=AuctionItems}"
    x:Key="listingDataView" />
</Window.Resources>
```

The resource *listingDataView* then serves as the binding source for elements in the app, such as the [ListBox](#).

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
  ItemsSource="{Binding Source={StaticResource listingDataView}}" />
```

To create another view for the same collection, you can create another [CollectionViewSource](#) instance and give it a different `x:Key` name.

The following table shows what view data types are created as the default collection view or by [CollectionViewSource](#) based on the source collection type.

SOURCE COLLECTION TYPE	COLLECTION VIEW TYPE	NOTES
IEnumerable	An internal type based on CollectionView	Cannot group items.
IList	ListCollectionView	Fastest.
IBindingList	BindingListCollectionView	

Using a default view

Specifying a collection view as a binding source is one way to create and use a collection view. WPF also creates a default collection view for every collection used as a binding source. If you bind directly to a collection, WPF binds to its default view. This default view is shared by all bindings to the same collection, so a change made to a default view by one bound control or code (such as sorting or a change to the current item pointer, discussed later) is reflected in all other bindings to the same collection.

To get the default view, you use the [GetDefaultView](#) method. For an example, see [Get the default view of a data collection \(.NET Framework\)](#).

Collection views with ADO.NET DataTables

To improve performance, collection views for ADO.NET [DataTable](#) or [DataView](#) objects delegate sorting and filtering to the [DataView](#), which causes sorting and filtering to be shared across all collection views of the data

source. To enable each collection view to sort and filter independently, initialize each collection view with its own [DataView](#) object.

Sorting

As mentioned before, views can apply a sort order to a collection. As it exists in the underlying collection, your data may or may not have a relevant, inherent order. The view over the collection allows you to impose an order, or change the default order, based on comparison criteria that you supply. Because it's a client-based view of the data, a common scenario is that the user might want to sort columns of tabular data per the value that the column corresponds to. Using views, this user-driven sort can be applied, again without making any changes to the underlying collection or even having to requery for the collection content. For an example, see [Sort a GridView column when a header is clicked \(.NET Framework\)](#).

The following example shows the sorting logic of the "Sort by category and date" [CheckBox](#) of the app UI in the [What is data binding](#) section.

```
private void AddSortCheckBox_Checked(object sender, RoutedEventArgs e)
{
    // Sort the items first by Category and then by StartDate
    listingDataView.SortDescriptions.Add(new SortDescription("Category", ListSortDirection.Ascending));
    listingDataView.SortDescriptions.Add(new SortDescription("StartDate", ListSortDirection.Ascending));
}
```

```
Private Sub AddSortCheckBox_Checked(sender As Object, e As RoutedEventArgs)
    ' Sort the items first by Category And then by StartDate
    listingDataView.SortDescriptions.Add(New SortDescription("Category", ListSortDirection.Ascending))
    listingDataView.SortDescriptions.Add(New SortDescription("StartDate", ListSortDirection.Ascending))
End Sub
```

Filtering

Views can also apply a filter to a collection, so that the view shows only a certain subset of the full collection. You might filter on a condition in the data. For instance, as is done by the app in the [What is data binding](#) section, the "Show only bargains" [CheckBox](#) contains logic to filter out items that cost \$25 or more. The following code is executed to set *ShowOnlyBargainsFilter* as the [Filter](#) event handler when that [CheckBox](#) is selected.

```
private void AddFilteringCheckBox_Checked(object sender, RoutedEventArgs e)
{
    if (((CheckBox)sender).IsChecked == true)
        listingDataView.Filter += ListingDataView_Filter;
    else
        listingDataView.Filter -= ListingDataView_Filter;
}
```

```
Private Sub AddFilteringCheckBox_Checked(sender As Object, e As RoutedEventArgs)
    Dim checkBox = DirectCast(sender, CheckBox)

    If checkBox.IsChecked = True Then
        AddHandler listingDataView.Filter, AddressOf ListingDataView_Filter
    Else
        RemoveHandler listingDataView.Filter, AddressOf ListingDataView_Filter
    End If
End Sub
```

The *ShowOnlyBargainsFilter* event handler has the following implementation.

```
private void ListingDataView_Filter(object sender, FilterEventArgs e)
{
    // Start with everything excluded
    e.Accepted = false;

    // Only include items with a price less than 25
    if (e.Item is AuctionItem product && product.CurrentPrice < 25)
        e.Accepted = true;
}
```

```
Private Sub ListingDataView_Filter(sender As Object, e As FilterEventArgs)

    ' Start with everything excluded
    e.Accepted = False

    Dim product As AuctionItem = TryCast(e.Item, AuctionItem)

    If product IsNot Nothing Then

        ' Only include products with prices lower than 25
        If product.CurrentPrice < 25 Then e.Accepted = True

    End If

End Sub
```

If you're using one of the [CollectionView](#) classes directly instead of [CollectionViewSource](#), you would use the [Filter](#) property to specify a callback. For an example, see [Filter Data in a View \(.NET Framework\)](#).

Grouping

Except for the internal class that views an [IEnumerable](#) collection, all collection views support *grouping*, which allows the user to partition the collection in the collection view into logical groups. The groups can be explicit, where the user supplies a list of groups, or implicit, where the groups are generated dynamically depending on the data.

The following example shows the logic of the "Group by category" [CheckBox](#).

```
// This groups the items in the view by the property "Category"
var groupDescription = new PropertyGroupDescription();
groupDescription.PropertyName = "Category";
listingDataView.GroupDescriptions.Add(groupDescription);
```

```
' This groups the items in the view by the property "Category"
Dim groupDescription = New PropertyGroupDescription()
groupDescription.PropertyName = "Category"
listingDataView.GroupDescriptions.Add(groupDescription)
```

For another grouping example, see [Group Items in a ListView That Implements a GridView \(.NET Framework\)](#).

Current item pointers

Views also support the notion of a current item. You can navigate through the objects in a collection view. As you navigate, you're moving an item pointer that allows you to retrieve the object that exists at that particular location in the collection. For an example, see [Navigate through the objects in a data CollectionView \(.NET Framework\)](#).

Because WPF binds to a collection only by using a view (either a view you specify, or the collection's default view), all bindings to collections have a current item pointer. When binding to a view, the slash ("/") character in a `Path` value designates the current item of the view. In the following example, the data context is a collection

view. The first line binds to the collection. The second line binds to the current item in the collection. The third line binds to the `Description` property of the current item in the collection.

```
<Button Content="{Binding }" />
<Button Content="{Binding Path=}" />
<Button Content="{Binding Path=/Description}" />
```

The slash and property syntax can also be stacked to traverse a hierarchy of collections. The following example binds to the current item of a collection named `Offices`, which is a property of the current item of the source collection.

```
<Button Content="{Binding /Offices/}" />
```

The current item pointer can be affected by any sorting or filtering that is applied to the collection. Sorting preserves the current item pointer on the last item selected, but the collection view is now restructured around it. (Perhaps the selected item was at the beginning of the list before, but now the selected item might be somewhere in the middle.) Filtering preserves the selected item if that selection remains in view after the filtering. Otherwise, the current item pointer is set to the first item of the filtered collection view.

Master-detail binding scenario

The notion of a current item is useful not only for navigation of items in a collection, but also for the master-detail binding scenario. Consider the app UI in the [What is data binding](#) section again. In that app, the selection within the `ListBox` determines the content shown in the `ContentControl`. To put it in another way, when a `ListBox` item is selected, the `ContentControl` shows the details of the selected item.

You can implement the master-detail scenario simply by having two or more controls bound to the same view. The following example from the [Data binding demo](#) shows the markup of the `ListBox` and the `ContentControl` you see on the app UI in the [What is data binding](#) section.

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
    ItemsSource="{Binding Source={StaticResource listingDataView}}" />
<ContentControl Name="Detail" Grid.Row="3" Grid.ColumnSpan="3"
    Content="{Binding Source={StaticResource listingDataView}}"
    ContentTemplate="{StaticResource detailsProductListingTemplate}"
    Margin="9,0,0,0"/>
```

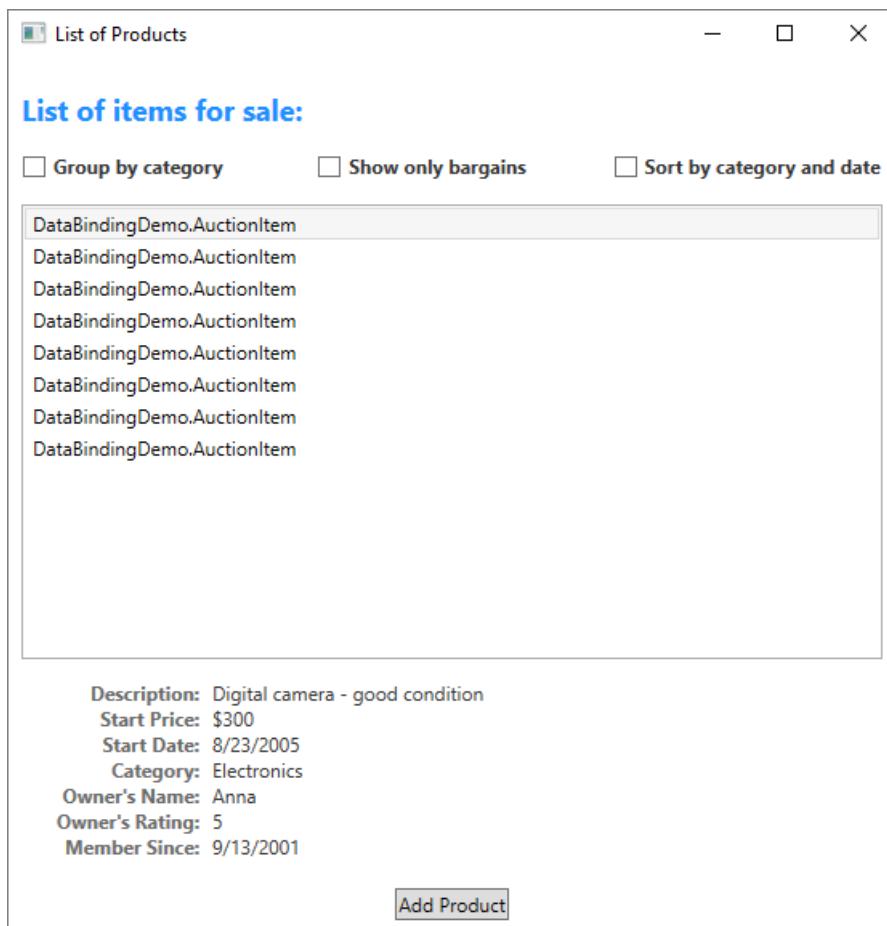
Notice that both of the controls are bound to the same source, the `listingDataView` static resource (see the definition of this resource in the [How to create a view section](#)). This binding works because when a singleton object (the `ContentControl` in this case) is bound to a collection view, it automatically binds to the `CurrentItem` of the view. The `CollectionViewSource` objects automatically synchronize currency and selection. If your list control isn't bound to a `CollectionViewSource` object as in this example, then you would need to set its `IsSynchronizedWithCurrentItem` property to `true` for this to work.

For other examples, see [Bind to a collection and display information based on selection \(.NET Framework\)](#) and [Use the master-detail pattern with hierarchical data \(.NET Framework\)](#).

You may have noticed that the above example uses a template. In fact, the data would not be displayed the way we wish without the use of templates (the one explicitly used by the `ContentControl` and the one implicitly used by the `ListBox`). We now turn to data templating in the next section.

Data templating

Without the use of data templates, our app UI in the [What is data binding](#) section would look like the following.



As shown in the example in the previous section, both the [ListBox](#) control and the [ContentControl](#) are bound to the entire collection object (or more specifically, the view over the collection object) of *AuctionItems*. Without specific instructions of how to display the data collection, the [ListBox](#) displays the string representation of each object in the underlying collection, and the [ContentControl](#) displays the string representation of the object it's bound to.

To solve that problem, the app defines [DataTemplates](#). As shown in the example in the previous section, the [ContentControl](#) explicitly uses the *detailsProductListingTemplate* data template. The [ListBox](#) control implicitly uses the following data template when displaying the *AuctionItem* objects in the collection.

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
  <Border BorderThickness="1" BorderBrush="Gray"
    Padding="7" Name="border" Margin="3" Width="500">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="20"/>
        <ColumnDefinition Width="86"/>
        <ColumnDefinition Width="*/>
      </Grid.ColumnDefinitions>

      <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
        Fill="Yellow" Stroke="Black" StrokeThickness="1"
        StrokeLineJoin="Round" Width="20" Height="20"
        Stretch="Fill"
        Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7 7,7"
        Visibility="Hidden" Name="star"/>

      <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
        Name="descriptionText" Text="{Binding Description}"
        TextWrapping="Wrap" Width="480"/>
```

```

        Name="descriptionTitle"
        Style="{StaticResource smallTitleStyle}">Description:</TextBlock>

<TextBlock Name="DescriptionDTDataType" Grid.Row="0" Grid.Column="2"
    Text="{Binding Path=Description}"
    Style="{StaticResource textStyleTextBlock}"/>

<TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
    Name="currentPriceTitle"
    Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>

<StackPanel Grid.Row="1" Grid.Column="2" Orientation="Horizontal">
    <TextBlock Text="$" Style="{StaticResource textStyleTextBlock}"/>
    <TextBlock Name="CurrentPriceDTDataType"
        Text="{Binding Path=CurrentPrice}"
        Style="{StaticResource textStyleTextBlock}"/>
</StackPanel>
</Grid>
</Border>
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Color</src:SpecialFeatures>
        </DataTrigger.Value>
        <DataTrigger.Setters>
            <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
            <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
            <Setter Property="BorderThickness" Value="3" TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger.Setters>
    </DataTrigger>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Highlight</src:SpecialFeatures>
        </DataTrigger.Value>
        <Setter Property="BorderBrush" Value="Orange" TargetName="border" />
        <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
        <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
        <Setter Property="Visibility" Value="Visible" TargetName="star" />
        <Setter Property="BorderThickness" Value="3" TargetName="border" />
        <Setter Property="Padding" Value="5" TargetName="border" />
    </DataTrigger>
</DataTemplate.Triggers>
</DataTemplate>

```

With the use of those two DataTemplates, the resulting UI is the one shown in the [What is data binding](#) section. As you can see from that screenshot, in addition to letting you place data in your controls, DataTemplates allow you to define compelling visuals for your data. For example, [DataTriggers](#) are used in the above [DataTemplate](#) so that *AuctionItems* with *SpecialFeatures* value of *HighLight* would be displayed with an orange border and a star.

For more information about data templates, see the [Data templating overview \(.NET Framework\)](#).

Data validation

Most app that take user input need to have validation logic to ensure that the user has entered the expected information. The validation checks can be based on type, range, format, or other app-specific requirements. This section discusses how data validation works in WPF.

Associating validation rules with a binding

The WPF data binding model allows you to associate [ValidationRules](#) with your [Binding](#) object. For example, the following example binds a [TextBox](#) to a property named `StartPrice` and adds a [ExceptionValidationRule](#) object to the [Binding.ValidationRules](#) property.

```

<TextBox Name="StartPriceEntryForm" Grid.Row="2"
        Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5" Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>

```

A [ValidationRule](#) object checks whether the value of a property is valid. WPF has two types of built-in [ValidationRule](#) objects:

- A [ExceptionValidationRule](#) checks for exceptions thrown during the update of the binding source property. In the previous example, `StartPrice` is of type integer. When the user enters a value that cannot be converted to an integer, an exception is thrown, causing the binding to be marked as invalid. An alternative syntax to setting the [ExceptionValidationRule](#) explicitly is to set the [ValidatesOnExceptions](#) property to `true` on your [Binding](#) or [MultiBinding](#) object.
- A [DataErrorValidationRule](#) object checks for errors that are raised by objects that implement the [IDataErrorInfo](#) interface. For an example of using this validation rule, see [DataErrorValidationRule](#). An alternative syntax to setting the [DataErrorValidationRule](#) explicitly is to set the [ValidatesOnDataErrors](#) property to `true` on your [Binding](#) or [MultiBinding](#) object.

You can also create your own validation rule by deriving from the [ValidationRule](#) class and implementing the [Validate](#) method. The following example shows the rule used by the *Add Product Listing* "Start Date" [TextBox](#) from the [What is data binding](#) section.

```

public class FutureDateRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        // Test if date is valid
        if (DateTime.TryParse(value.ToString(), out DateTime date))
        {
            // Date is not in the future, fail
            if (DateTime.Now > date)
                return new ValidationResult(false, "Please enter a date in the future.");
        }
        else
        {
            // Date is not a valid date, fail
            return new ValidationResult(false, "Value is not a valid date.");
        }

        // Date is valid and in the future, pass
        return ValidationResult.ValidResult;
    }
}

```



```

Public Class FutureDateRule
    Inherits ValidationRule

    Public Overrides Function Validate(value As Object, cultureInfo As CultureInfo) As ValidationResult

        Dim inputDate As Date

        ' Test if date is valid
        If Date.TryParse(value.ToString, inputDate) Then

            ' Date is not in the future, fail
            If Date.Now > inputDate Then
                Return New ValidationResult(False, "Please enter a date in the future.")
            End If

        Else
            ' // Date Is Not a valid date, fail
            Return New ValidationResult(False, "Value is not a valid date.")
        End If

        ' Date is valid and in the future, pass
        Return ValidationResult.ValidResult

    End Function

End Class

```

The *StartDateEntryForm* [TextBox](#) uses this *FutureDateRule*, as shown in the following example.

```

<TextBox Name="StartDateEntryForm" Grid.Row="3"
    Validation.ErrorTemplate="{StaticResource validationTemplate}"
    Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5" Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged"
            Converter="{StaticResource dateConverter}" >
            <Binding.ValidationRules>
                <src:FutureDateRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>

```

Because the [UpdateSourceTrigger](#) value is [PropertyChanged](#), the binding engine updates the source value on every keystroke, which means it also checks every rule in the [ValidationRules](#) collection on every keystroke. We discuss this further in the Validation Process section.

Providing visual feedback

If the user enters an invalid value, you may want to provide some feedback about the error on the app UI. One way to provide such feedback is to set the [Validation.ErrorTemplate](#) attached property to a custom [ControlTemplate](#). As shown in the previous subsection, the *StartDateEntryForm* [TextBox](#) uses an [ErrorTemplate](#) called *validationTemplate*. The following example shows the definition of *validationTemplate*.

```

<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>

```

The [AdornedElementPlaceholder](#) element specifies where the control being adorned should be placed.

In addition, you may also use a [ToolTip](#) to display the error message. Both the *StartDateEntryForm* and the *StartPriceEntryForm* [TextBoxes](#) use the style *textStyleTextBox*, which creates a [ToolTip](#) that displays the error message. The following example shows the definition of *textStyleTextBox*. The attached property [Validation.HasError](#) is `true` when one or more of the bindings on the properties of the bound element are in error.

```
<Style x:Key="textStyleTextBox" TargetType="TextBox">
  <Setter Property="Foreground" Value="#333333" />
  <Setter Property="MaxLength" Value="40" />
  <Setter Property="Width" Value="392" />
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="ToolTip"
        Value="{Binding (Validation.Errors).CurrentItem.ErrorContent, RelativeSource=
{RelativeSource Self}}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

With the custom [ErrorTemplate](#) and the [ToolTip](#), the *StartDateEntryForm* [TextBox](#) looks like the following when there's a validation error.

If your [Binding](#) has associated validation rules but you do not specify an [ErrorTemplate](#) on the bound control, a default [ErrorTemplate](#) will be used to notify users when there's a validation error. The default [ErrorTemplate](#) is a control template that defines a red border in the adorning layer. With the default [ErrorTemplate](#) and the [ToolTip](#), the UI of the *StartPriceEntryForm* [TextBox](#) looks like the following when there's a validation error.

For an example of how to provide logic to validate all controls in a dialog box, see the Custom Dialog Boxes section in the [Dialog boxes overview](#).

Validation process

Validation usually occurs when the value of a target is transferred to the binding source property. This transfer occurs on [TwoWay](#) and [OneWayToSource](#) bindings. To reiterate, what causes a source update depends on the value of the [UpdateSourceTrigger](#) property, as described in the [What triggers source updates](#) section.

The following items describe the *validation* process. If a validation error or other type of error occurs at any time during this process, the process is halted:

1. The binding engine checks if there are any custom [ValidationRule](#) objects defined whose [ValidationStep](#) is set to [RawProposedValue](#) for that [Binding](#), in which case it calls the [Validate](#) method on each [ValidationRule](#) until one of them runs into an error or until all of them pass.
2. The binding engine then calls the converter, if one exists.
3. If the converter succeeds, the binding engine checks if there are any custom [ValidationRule](#) objects defined whose [ValidationStep](#) is set to [ConvertedProposedValue](#) for that [Binding](#), in which case it calls the [Validate](#) method on each [ValidationRule](#) that has [ValidationStep](#) set to [ConvertedProposedValue](#) until one of them runs into an error or until all of them pass.
4. The binding engine sets the source property.

5. The binding engine checks if there are any custom [ValidationRule](#) objects defined whose [ValidationStep](#) is set to [UpdatedValue](#) for that [Binding](#), in which case it calls the [Validate](#) method on each [ValidationRule](#) that has [ValidationStep](#) set to [UpdatedValue](#) until one of them runs into an error or until all of them pass. If a [DataErrorValidationRule](#) is associated with a binding and its [ValidationStep](#) is set to the default, [UpdatedValue](#), the [DataErrorValidationRule](#) is checked at this point. At this point any binding that has the [ValidatesOnDataErrors](#) set to `true` is checked.
6. The binding engine checks if there are any custom [ValidationRule](#) objects defined whose [ValidationStep](#) is set to [CommittedValue](#) for that [Binding](#), in which case it calls the [Validate](#) method on each [ValidationRule](#) that has [ValidationStep](#) set to [CommittedValue](#) until one of them runs into an error or until all of them pass.

If a [ValidationRule](#) doesn't pass at any time throughout this process, the binding engine creates a [ValidationError](#) object and adds it to the [Validation.Errors](#) collection of the bound element. Before the binding engine runs the [ValidationRule](#) objects at any given step, it removes any [ValidationError](#) that was added to the [Validation.Errors](#) attached property of the bound element during that step. For example, if a [ValidationRule](#) whose [ValidationStep](#) is set to [UpdatedValue](#) failed, the next time the validation process occurs, the binding engine removes that [ValidationError](#) immediately before it calls any [ValidationRule](#) that has [ValidationStep](#) set to [UpdatedValue](#).

When [Validation.Errors](#) isn't empty, the [Validation.HasError](#) attached property of the element is set to `true`. Also, if the [NotifyOnValidationError](#) property of the [Binding](#) is set to `true`, then the binding engine raises the [Validation.Error](#) attached event on the element.

Also note that a valid value transfer in either direction (target to source or source to target) clears the [Validation.Errors](#) attached property.

If the binding either has an [ExceptionValidationRule](#) associated with it, or had the [ValidatesOnExceptions](#) property is set to `true` and an exception is thrown when the binding engine sets the source, the binding engine checks to see if there's a [UpdateSourceExceptionFilter](#). You can use the [UpdateSourceExceptionFilter](#) callback to provide a custom handler for handling exceptions. If an [UpdateSourceExceptionFilter](#) isn't specified on the [Binding](#), the binding engine creates a [ValidationError](#) with the exception and adds it to the [Validation.Errors](#) collection of the bound element.

Debugging mechanism

You can set the attached property [PresentationTraceSources.TraceLevel](#) on a binding-related object to receive information about the status of a specific binding.

See also

- [Data binding demo](#)
- [Binding declarations overview](#)
- [Binding sources overview](#)
- [DataErrorValidationRule](#)

Binding declarations overview (WPF .NET)

5/1/2021 • 8 minutes to read • [Edit Online](#)

Typically, developers declare the bindings directly in the XAML markup of the UI elements they want to bind data to. However, you can also declare bindings in code. This article describes how to declare bindings in both XAML and in code.

Prerequisites

Before reading this article, it's important that you're familiar with the concept and usage of markup extensions. For more information about markup extensions, see [Markup Extensions and WPF XAML](#).

This article doesn't cover data binding concepts. For a discussion of data binding concepts, see [Data binding overview](#).

Declare a binding in XAML

[Binding](#) is a markup extension. When you use the binding extension to declare a binding, the declaration consists of a series of clauses following the `Binding` keyword and separated by commas (.). The clauses in the binding declaration can be in any order and there are many possible combinations. The clauses are *Name=Value* pairs, where *Name* is the name of the [Binding](#) property and *Value* is the value you're setting for the property.

When creating binding declaration strings in markup, they must be attached to the specific dependency property of a target object. The following example shows how to bind the `TextBox.Text` property using the binding extension, specifying the [Source](#) and [Path](#) properties.

```
<TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=Name}"/>
```

You can specify most of the properties of the [Binding](#) class this way. For more information about the binding extension and for a list of [Binding](#) properties that cannot be set using the binding extension, see the [Binding Markup Extension \(.NET Framework\)](#) overview.

Object element syntax

Object element syntax is an alternative to creating the binding declaration. In most cases, there's no particular advantage to using either the markup extension or the object element syntax. However, when the markup extension doesn't support your scenario, such as when your property value is of a non-string type for which no type conversion exists, you need to use the object element syntax.

The previous section demonstrated how to bind with a XAML extension. The following example demonstrates doing the same binding but uses object element syntax:

```
<TextBlock>
  <TextBlock.Text>
    <Binding Source="{StaticResource myDataSource}" Path="Name"/>
  </TextBlock.Text>
</TextBlock>
```

For more information about the different terms, see [XAML Syntax In Detail \(.NET Framework\)](#).

MultiBinding and PriorityBinding

[MultiBinding](#) and [PriorityBinding](#) don't support the XAML extension syntax. That's why you must use the object element syntax if you're declaring a [MultiBinding](#) or a [PriorityBinding](#) in XAML.

Create a binding in code

Another way to specify a binding is to set properties directly on a [Binding](#) object in code, and then assign the binding to a property. The following example shows how to create a [Binding](#) object in code.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Make a new data source object
    var personDetails = new Person()
    {
        Name = "John",
        Birthdate = DateTime.Parse("2001-02-03")
    };

    // New binding object using the path of 'Name' for whatever source object is used
    var nameBindingObject = new Binding("Name");

    // Configure the binding
    nameBindingObject.Mode = BindingMode.OneWay;
    nameBindingObject.Source = personDetails;
    nameBindingObject.Converter = NameConverter.Instance;
    nameBindingObject.ConverterCulture = new CultureInfo("en-US");

    // Set the binding to a target object. The TextBlock.Name property on the NameBlock UI element
    BindingOperations.SetBinding(NameBlock, TextBlock.TextProperty, nameBindingObject);
}
```

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)

    ' Make a new data source object
    Dim personDetails As New Person() With {
        .Name = "John",
        .Birthdate = Date.Parse("2001-02-03")
    }

    ' New binding object using the path of 'Name' for whatever source object is used
    Dim nameBindingObject As New Binding("Name")

    ' Configure the binding
    nameBindingObject.Mode = BindingMode.OneWay
    nameBindingObject.Source = personDetails
    nameBindingObject.Converter = NameConverter.Instance
    nameBindingObject.ConverterCulture = New CultureInfo("en-US")

    ' Set the binding to a target object. The TextBlock.Name property on the NameBlock UI element
    BindingOperations.SetBinding(NameBlock, TextBlock.TextProperty, nameBindingObject)

End Sub
```

The previous code set the following on the binding:

- A path of the property on the data source object.
- The mode of the binding.
- The data source, in this case, a simple object instance representing a person.
- An optional converter that processes the value coming in from the data source object before it's assigned to the target property.

When the object you're binding is a [FrameworkElement](#) or a [FrameworkContentElement](#), you can call the

`SetBinding` method on your object directly instead of using [BindingOperations.SetBinding](#). For an example, see [How to: Create a Binding in Code](#).

The previous example uses a simple data object type of `Person`. The following is the code for that object:

```
class Person
{
    public string Name { get; set; }
    public DateTime Birthdate { get; set; }
}
```

```
Public Class Person

    Public Property Name As String
    Public Property Birthdate As DateTime

End Class
```

Binding path syntax

Use the [Path](#) property to specify the source value you want to bind to:

- In the simplest case, the [Path](#) property value is the name of the property of the source object to use for the binding, such as `Path=PropertyName`.
- Subproperties of a property can be specified by a similar syntax as in C#. For instance, the clause `Path=ShoppingCart.Order` sets the binding to the subproperty `Order` of the object or property `ShoppingCart`.
- To bind to an attached property, place parentheses around the attached property. For example, to bind to the attached property [DockPanel.Dock](#), the syntax is `Path=(DockPanel.Dock)`.
- Indexers of a property can be specified within square brackets following the property name where the indexer is applied. For instance, the clause `Path=ShoppingCart[0]` sets the binding to the index that corresponds to how your property's internal indexing handles the literal string "0". Nested indexers are also supported.
- Indexers and subproperties can be mixed in a `Path` clause; for example, `Path=ShoppingCart.ShippingInfo[MailingAddress,Street]`.
- Inside indexers. You can have multiple indexer parameters separated by commas (,). The type of each parameter can be specified with parentheses. For example, you can have `Path="[(sys:Int32)42,(sys:Int32)24]"`, where `sys` is mapped to the `System` namespace.
- When the source is a collection view, the current item can be specified with a slash (/). For example, the clause `Path= /` sets the binding to the current item in the view. When the source is a collection, this syntax specifies the current item of the default collection view.
- Property names and slashes can be combined to traverse properties that are collections. For example, `Path=/Offices/ManagerName` specifies the current item of the source collection, which contains an `Offices` property that is also a collection. Its current item is an object that contains a `ManagerName` property.
- Optionally, a period (.) path can be used to bind to the current source. For example, `Text="{Binding}"` is equivalent to `Text="{Binding Path=.}"`.

Escaping mechanism

- Inside indexers (`[]`), the caret character (`^`) escapes the next character.
- If you set [Path](#) in XAML, you also need to escape (using XML entities) certain characters that are special to the XML language definition:
 - Use `&` to escape the character `"&"`.
 - Use `>` to escape the end tag `">"`.
- Additionally, if you describe the entire binding in an attribute using the markup extension syntax, you need to escape (using backslash `\`) characters that are special to the WPF markup extension parser:
 - Backslash (`\`) is the escape character itself.
 - The equal sign (`=`) separates property name from property value.
 - Comma (`,`) separates properties.
 - The right curly brace (`}`) is the end of a markup extension.

Binding direction

Use the [Binding.Mode](#) property to specify the direction of the binding. The following modes are the available options for binding updates:

BINDING MODE	DESCRIPTION
BindingMode.TwoWay	Updates the target property or the property whenever either the target property or the source property changes.
BindingMode.OneWay	Updates the target property only when the source property changes.
BindingMode.OneTime	Updates the target property only when the application starts or when the DataContext undergoes a change.
BindingMode.OneWayToSource	Updates the source property when the target property changes.
BindingMode.Default	Causes the default Mode value of target property to be used.

For more information, see the [BindingMode](#) enumeration.

The following example shows how to set the [Mode](#) property:

```
<TextBlock Name="IncomeText" Text="{Binding Path=TotalIncome, Mode=OneTime}" />
```

To detect source changes (applicable to [OneWay](#) and [TwoWay](#) bindings), the source must implement a suitable property change notification mechanism such as [INotifyPropertyChanged](#). For more information, see [Providing change notifications](#).

For [TwoWay](#) or [OneWayToSource](#) bindings, you can control the timing of the source updates by setting the [UpdateSourceTrigger](#) property. For more information, see [UpdateSourceTrigger](#).

Default behaviors

The default behavior is as follows if not specified in the declaration:

- A default converter is created that tries to do a type conversion between the binding source value and the binding target value. If a conversion cannot be made, the default converter returns `null`.
- If you don't set [ConverterCulture](#), the binding engine uses the `Language` property of the binding target object. In XAML, this defaults to `en-US` or inherits the value from the root element (or any element) of the page, if one has been explicitly set.
- As long as the binding already has a data context (for example, the inherited data context coming from a parent element), and whatever item or collection being returned by that context is appropriate for binding without requiring further path modification, a binding declaration can have no clauses at all: `{Binding}`. This is often the way a binding is specified for data styling, where the binding acts upon a collection. For more information, see [Using Entire Objects as a Binding Source](#).
- The default [Mode](#) varies between one-way and two-way depending on the dependency property that is being bound. You can always declare the binding mode explicitly to ensure that your binding has the desired behavior. In general, user-editable control properties, such as [TextBox.Text](#) and [RangeBase.Value](#), default to two-way bindings, but most other properties default to one-way bindings.
- The default [UpdateSourceTrigger](#) value varies between [PropertyChanged](#) and [LostFocus](#) depending on the bound dependency property as well. The default value for most dependency properties is [PropertyChanged](#), while the [TextBox.Text](#) property has a default value of [LostFocus](#).

See also

- [Data binding overview](#)
- [Binding sources overview](#)
- [PropertyPath XAML Syntax \(.NET Framework\)](#)

Binding sources overview (WPF .NET)

5/1/2021 • 6 minutes to read • [Edit Online](#)

In data binding, the binding source object refers to the object you obtain data from. This article discusses the types of objects you can use as the binding source, like .NET CLR objects, XML, and [DependencyObject](#) objects.

Binding source types

Windows Presentation Foundation (WPF) data binding supports the following binding source types:

- **.NET common language runtime (CLR) objects**

You can bind to public properties, sub-properties, and indexers of any common language runtime (CLR) object. The binding engine uses CLR reflection to get the values of the properties. Objects that implement [ICustomTypeDescriptor](#) or have a registered [TypeDescriptionProvider](#) also work with the binding engine.

For more information about how to implement a class that can serve as a binding source, see [Implementing a binding source on your objects](#) later in this article.

- **Dynamic objects**

You can bind to available properties and indexers of an object that implements the [IDynamicMetaObjectProvider](#) interface. If you can access the member in code, you can bind to it. For example, if a dynamic object enables you to access a member in code via `someObject.AProperty`, you can bind to it by setting the binding path to `AProperty`.

- **ADO.NET objects**

You can bind to ADO.NET objects, such as [DataTable](#). The ADO.NET [DataView](#) implements the [IBindingList](#) interface, which provides change notifications that the binding engine listens for.

- **XML objects**

You can bind to and run `xPath` queries on an [XmlNode](#), [XmlDocument](#), or [XmlElement](#). A convenient way to access XML data that is the binding source in markup is to use an [XmlDataProvider](#) object. For more information, see [Bind to XML Data Using an XMLDataProvider and XPath Queries \(.NET Framework\)](#).

You can also bind to an [XElement](#) or [XDocument](#), or bind to the results of queries run on objects of these types by using LINQ to XML. A convenient way to use LINQ to XML to access XML data that is the binding source in markup is to use an [ObjectDataProvider](#) object. For more information, see [Bind to XDocument, XElement, or LINQ for XML Query Results \(.NET Framework\)](#).

- **[DependencyObject](#) objects**

You can bind to dependency properties of any [DependencyObject](#). For an example, see [Bind the Properties of Two Controls \(.NET Framework\)](#).

Implement a binding source on your objects

Your CLR objects can become binding sources. There are a few things to be aware of when implementing a class to serve as a binding source.

Provide change notifications

If you're using either [OneWay](#) or [TwoWay](#) binding, implement a suitable "property changed" notification

mechanism. The recommended mechanism is for the CLR or dynamic class to implement the [INotifyPropertyChanged](#) interface. For more information, see [How to: Implement Property Change Notification \(.NET Framework\)](#).

There are two ways to notify a subscriber of a property change:

1. Implement the [INotifyPropertyChanged](#) interface.

This is the recommended mechanism for notifications. The [INotifyPropertyChanged](#) supplies the [PropertyChanged](#) event, which the binding system respects. By raising this event, and providing the name of the property that changed, you'll notify a binding target of the change.

2. Implement the `PropertyChanged` pattern.

Each property that needs to notify a binding target that it's changed, has a corresponding `PropertyNameChanged` event, where `PropertyName` is the name of the property. You raise the event every time the property changes.

If your binding source implements one of these notification mechanisms, target updates happen automatically. If for any reason your binding source doesn't provide the proper property changed notifications, you can use the [UpdateTarget](#) method to update the target property explicitly.

Other characteristics

The following list provides other important points to note:

- Data objects that serve as binding sources can be declared in XAML as resources, provided they have a **parameterless constructor**. Otherwise, you must create the data object in code and directly assign it to either the data context of your XAML object tree, or as the binding source of binding.
- The properties you use as binding source properties must be public properties of your class. Explicitly defined interface properties can't be accessed for binding purposes, nor can protected, private, internal, or virtual properties that have no base implementation.
- You can't bind to public fields.
- The type of the property declared in your class is the type that is passed to the binding. However, the type ultimately used by the binding depends on the type of the binding target property, not of the binding source property. If there's a difference in type, you might want to write a converter to handle how your custom property is initially passed to the binding. For more information, see [IValueConverter](#).

Entire objects as a binding source

You can use an entire object as a binding source. Specify a binding source by using the [Source](#) or the [DataContext](#) property, and then provide a blank binding declaration: `{Binding}`. Scenarios in which this is useful include binding to objects that are of type string, binding to objects with multiple properties you're interested in, or binding to collection objects. For an example of binding to an entire collection object, see [How to Use the Master-Detail Pattern with Hierarchical Data \(.NET Framework\)](#).

You may need to apply custom logic so that the data is meaningful to your bound target property. The custom logic may be in the form of a custom converter or a [DataTemplate](#). For more information about converters, see [Data conversion](#). For more information about data templates, see [Data Templating Overview \(.NET Framework\)](#).

Collection objects as a binding source

Often, the object you want to use as the binding source is a collection of custom objects. Each object serves as the source for one instance of a repeated binding. For example, you might have a `CustomerOrders` collection that consists of `CustomerOrder` objects, where your application iterates over the collection to determine how many

orders exist and the data contained in each order.

You can enumerate over any collection that implements the [IEnumerable](#) interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the [INotifyCollectionChanged](#) interface. This interface exposes an event that must be raised whenever the underlying collection changes.

The [ObservableCollection<T>](#) class is a built-in implementation of a data collection that exposes the [INotifyCollectionChanged](#) interface. The individual data objects within the collection must satisfy the requirements described in the preceding sections. For an example, see [How to Create and Bind to an ObservableCollection \(.NET Framework\)](#). Before you implement your own collection, consider using [ObservableCollection<T>](#) or one of the existing collection classes, such as [List<T>](#), [Collection<T>](#), and [BindingList<T>](#), among many others.

When you specify a collection as a binding source, WPF doesn't bind directly to the collection. Instead, WPF actually binds to the collection's default view. For information about default views, see [Using a default view](#).

If you have an advanced scenario and you want to implement your own collection, consider using the [IList](#) interface. This interface provides a non-generic collection of objects that can be individually accessed by index, which can improve performance.

Permission requirements in data binding

Unlike .NET Framework, .NET 5+ (and .NET Core 3.1) runs with full-trust security. All data binding runs with the same access as the user running the application.

See also

- [ObjectDataProvider](#)
- [XmlDataProvider](#)
- [Data binding overview](#)
- [Binding sources overview](#)
- [Overview of WPF data binding with LINQ to XML \(.NET Framework\)](#)
- [Optimizing Performance: Data Binding \(.NET Framework\)](#)

How to bind to an enumeration (WPF .NET)

5/7/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to bind to an enumeration. Unfortunately there isn't a direct way to use an enumeration as a data binding source. However, the [Enum.GetValues\(Type\)](#) method returns a collection of values. These values can be wrapped in an [ObjectDataProvider](#) and used as a data source.

The [ObjectDataProvider](#) type provides a convenient way to create an object in XAML and use it as a data source.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Reference the enumeration

Use the [ObjectDataProvider](#) type to wrap an array of enumeration values provided by the enumeration type itself.

1. Create a new `ObjectDataProvider` as a XAML resource, either in your application XAML or the XAML of the object you're working with. This example uses a window and creates the `ObjectDataProvider` with a resource key of `EnumDataSource`.

```
<Window.Resources>
  <ObjectDataProvider x:Key="EnumDataSource"
    ObjectType="{x:Type sys:Enum}"
    MethodName="GetValues">
    <ObjectDataProvider.MethodParameters>
      <x:Type TypeName="HorizontalAlignment" />
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
</Window.Resources>
```

In this example, the `ObjectDataProvider` uses three properties to retrieve the enumeration:

PROPERTY	DESCRIPTION
<code>ObjectType</code>	The type of object to be returned by the data provider. In this example, <code>System.Enum</code> . The <code>sys:</code> XAML namespace is mapped to <code>System</code> .
<code>MethodName</code>	The name of the method to run on the <code>System.Enum</code> type. In this example, <code>Enum.GetValues</code> .
<code>MethodParameters</code>	A collection of values to provide to the <code>MethodName</code> method. In this example, the method takes the <code>System.Type</code> of the enumeration.

Effectively, the XAML is breaking down a method call, method name, parameters, and the return type. The `ObjectDataProvider` configured in the previous example is the equivalent of the following code:

```
var enumDataSource = System.Enum.GetValues(typeof(System.Windows.HorizontalAlignment));
```

```
Dim enumDataSource = System.Enum.GetValues(GetType(System.Windows.HorizontalAlignment))
```

2. Reference the `ObjectDataProvider` resource. The following XAML lists the enumeration values in a `ListBox` control:

```
<ListBox Name="myComboBox" SelectedIndex="0"
         ItemsSource="{Binding Source={StaticResource EnumDataSource}}"/>
```

Full XAML

The following XAML code represents a simple window that does the following:

1. Wraps the `HorizontalAlignment` enumeration in a `ObjectDataProvider` data source as a resource.
2. Provides a `ListBox` control to list all enumeration values.
3. Binds a `Button` control's `HorizontalAlignment` property to the selected item in the `ListBox`.

```
<Window x:Class="ArticleExample.BindEnumFull"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        SizeToContent="WidthAndHeight"
        Title="Enum binding">
    <Window.Resources>
        <ObjectDataProvider x:Key="EnumDataSource"
                           ObjectType="{x:Type sys:Enum}"
                           MethodName="GetValues">
            <ObjectDataProvider.MethodParameters>
                <x:Type TypeName="HorizontalAlignment" />
            </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
    </Window.Resources>

    <StackPanel Width="300" Margin="10">
        <TextBlock>Choose the HorizontalAlignment value of the Button:</TextBlock>

        <ListBox Name="myComboBox" SelectedIndex="0"
                 ItemsSource="{Binding Source={StaticResource EnumDataSource}}"/>

        <Button Content="I'm a button"
                HorizontalAlignment="{Binding ElementName=myComboBox, Path=SelectedItem}" />
    </StackPanel>
</Window>
```

See also

- [Data binding overview](#)
- [Binding sources overview](#)
- [StaticResource Markup Extension](#)
- [An alternative way to bind to an enumeration](#)

Overview of XAML resources (WPF .NET)

4/15/2021 • 13 minutes to read • [Edit Online](#)

A resource is an object that can be reused in different places in your app. Examples of resources include brushes and styles. This overview describes how to use resources in Extensible Application Markup Language (XAML). You can also create and access resources by using code.

NOTE

XAML resources described in this article are different from *app resources*, which are generally files added to an app, such as content, data, or embedded files.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Use resources in XAML

The following example defines a [SolidColorBrush](#) as a resource on the root element of a page. The example then references the resource and uses it to set properties of several child elements, including an [Ellipse](#), a [TextBlock](#), and a [Button](#).

```

<Window x:Class="resources.ResExample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ResExample" Height="400" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
        <Style TargetType="Border">
            <Setter Property="Background" Value="#4E1A3D" />
            <Setter Property="BorderThickness" Value="5" />
            <Setter Property="BorderBrush">
                <Setter.Value>
                    <LinearGradientBrush>
                        <GradientStop Offset="0.0" Color="#4E1A3D"/>
                        <GradientStop Offset="1.0" Color="Salmon"/>
                    </LinearGradientBrush>
                </Setter.Value>
            </Setter>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
            <Setter Property="FontFamily" Value="Trebuchet MS"/>
            <Setter Property="Margin" Value="0,10,10,10"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="Label">
            <Setter Property="HorizontalAlignment" Value="Right"/>
            <Setter Property="FontSize" Value="13"/>
            <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
            <Setter Property="FontFamily" Value="Arial"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <Setter Property="Margin" Value="0,3,10,0"/>
        </Style>
    </Window.Resources>

    <Border>
        <StackPanel>
            <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
            <TextBlock Style="{StaticResource Label}">Label</TextBlock>
            <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="{StaticResource MyBrush}"
Text="Text" Margin="20" />
            <Button HorizontalAlignment="Left" Height="30" Background="{StaticResource MyBrush}"
Margin="40">Button</Button>
            <Ellipse HorizontalAlignment="Center" Width="100" Height="100" Fill="{StaticResource MyBrush}"
Margin="10" />
        </StackPanel>
    </Border>

</Window>

```

Every framework-level element ([FrameworkElement](#) or [FrameworkContentElement](#)) has a [Resources](#) property, which is a [ResourceDictionary](#) type that contains defined resources. You can define resources on any element, such as a [Button](#). However, resources are most often defined on the root element, which is [Window](#) in the example.

Each resource in a resource dictionary must have a unique key. When you define resources in markup, you assign the unique key through the [x:Key Directive](#). Typically, the key is a string; however, you can also set it to other object types by using the appropriate markup extensions. Non-string keys for resources are used by certain feature areas in WPF, notably for styles, component resources, and data styling.

You can use a defined resource with the resource markup extension syntax that specifies the key name of the resource. For example, use the resource as the value of a property on another element.

```
<Button Background="{StaticResource MyBrush}"/>
<Ellipse Fill="{StaticResource MyBrush}"/>
```

In the preceding example, when the XAML loader processes the value `{StaticResource MyBrush}` for the `Background` property on `Button`, the resource lookup logic first checks the resource dictionary for the `Button` element. If `Button` doesn't have a definition of the resource key `MyBrush` (in that example it doesn't; its resource collection is empty), the lookup next checks the parent element of `Button`. If the resource isn't defined on the parent, it continues to check the object's logical tree upward until it's found.

If you define resources on the root element, all the elements in the logical tree, such as the `Window` or `Page`, can access it. And you can reuse the same resource for setting the value of any property that accepts the same type that the resource represents. In the previous example, the same `MyBrush` resource sets two different properties: `Button.Background` and `Rectangle.Fill`.

Static and dynamic resources

A resource can be referenced as either static or dynamic. References are created by using either the [StaticResource Markup Extension](#) or the [DynamicResource Markup Extension](#). A markup extension is a XAML feature that lets you specify an object reference by having the markup extension process the attribute string and return the object to a XAML loader. For more information about markup extension behavior, see [Markup Extensions and WPF XAML](#).

When you use a markup extension, you typically provide one or more parameters in string form that are processed by that particular markup extension. The [StaticResource Markup Extension](#) processes a key by looking up the value for that key in all available resource dictionaries. Processing happens during load, which is when the loading process needs to assign the property value. The [DynamicResource Markup Extension](#) instead processes a key by creating an expression, and that expression remains unevaluated until the app runs, at which time the expression is evaluated to provide a value.

When you reference a resource, the following considerations can influence whether you use a static resource reference or a dynamic resource reference:

- When determining the overall design of how you create the resources for your app (per page, in the app, in loose XAML, or in a resource-only assembly), consider the following:
- The app's functionality. Are updating resources in real-time part of your app requirements?
- The respective lookup behavior of that resource reference type.
- The particular property or resource type, and the native behavior of those types.

Static resources

Static resource references work best for the following circumstances:

- Your app design concentrates most of its resources into page or application-level resource dictionaries.

Static resource references aren't reevaluated based on runtime behaviors, such as reloading a page. So there can be some performance benefit to avoiding large numbers of dynamic resource references when they aren't necessary based on your resource and app design.

- You're setting the value of a property that isn't on a [DependencyObject](#) or a [Freezable](#).
- You're creating a resource dictionary that's compiled into a DLL that's shared between apps.
- You're creating a theme for a custom control and are defining resources that are used within the themes.

For this case, you typically don't want the dynamic resource reference lookup behavior. Instead, use static resource reference behavior so that the lookup is predictable and self-contained to the theme. With a dynamic resource reference, even a reference within a theme is left unevaluated until run-time. And, there's a chance that when the theme is applied, some local element will redefine a key that your theme is trying to reference, and the local element will fall before the theme itself in the lookup. If that happens, your theme won't behave as expected.

- You're using resources to set large numbers of dependency properties. Dependency properties have effective value caching as enabled by the property system, so if you provide a value for a dependency property that can be evaluated at load time, the dependency property doesn't have to check for a reevaluated expression and can return the last effective value. This technique can be a performance benefit.
- You want to change the underlying resource for all consumers, or you want to maintain separate writable instances for each consumer by using the [x:Shared Attribute](#).

Static resource lookup behavior

The following describes the lookup process that automatically happens when a static resource is referenced by a property or element:

1. The lookup process checks for the requested key within the resource dictionary defined by the element that sets the property.
2. The lookup process then traverses the logical tree upward to the parent element and its resource dictionary. This process continues until the root element is reached.
3. App resources are checked. App resources are those resources within the resource dictionary that is defined by the [Application](#) object for your WPF app.

Static resource references from within a resource dictionary must reference a resource that has already been defined lexically before the resource reference. Forward references cannot be resolved by a static resource reference. For this reason, design your resource dictionary structure such that resources are defined at or near the beginning of each respective resource dictionary.

Static resource lookup can extend into themes or into system resources, but this lookup is supported only because the XAML loader defers the request. The deferral is necessary so that the runtime theme at the time the page loads applies properly to the app. However, static resource references to keys that are known to only exist in themes or as system resources aren't recommended, because such references aren't reevaluated if the theme is changed by the user in real time. A dynamic resource reference is more reliable when you request theme or system resources. The exception is when a theme element itself requests another resource. These references should be static resource references, for the reasons mentioned earlier.

The exception behavior if a static resource reference isn't found varies. If the resource was deferred, then the exception occurs at runtime. If the resource wasn't deferred, the exception occurs at load time.

Dynamic resources

Dynamic resources work best when:

- The value of the resource, including system resources, or resources that are otherwise user settable, depends on conditions that aren't known until runtime. For example, you can create setter values that refer to system properties as exposed by [SystemColors](#), [SystemFonts](#), or [SystemParameters](#). These values are truly dynamic because they ultimately come from the runtime environment of the user and operating system. You might also have application-level themes that can change, where page-level resource access must also capture the change.
- You're creating or referencing theme styles for a custom control.

- You intend to adjust the contents of a [ResourceDictionary](#) during an app lifetime.
- You have a complicated resource structure that has interdependencies, where a forward reference may be required. Static resource references don't support forward references, but dynamic resource references do support them because the resource doesn't need to be evaluated until runtime, and forward references are therefore not a relevant concept.
- You're referencing a resource that is large from the perspective of a compile or working set, and the resource might not be used immediately when the page loads. Static resource references always load from XAML when the page loads. However, a dynamic resource reference doesn't load until it's used.
- You're creating a style where setter values might come from other values that are influenced by themes or other user settings.
- You're applying resources to elements that might be reparented in the logical tree during app lifetime. Changing the parent also potentially changes the resource lookup scope, so if you want the resource for a reparented element to be reevaluated based on the new scope, always use a dynamic resource reference.

Dynamic resource lookup behavior

Resource lookup behavior for a dynamic resource reference parallels the lookup behavior in your code if you call [FindResource](#) or [SetResourceReference](#):

1. The lookup checks for the requested key within the resource dictionary defined by the element that sets the property:
 - If the element defines a [Style](#) property, the [System.Windows.FrameworkElement.Style](#) of the element has its [Resources](#) dictionary checked.
 - If the element defines a [Template](#) property, the [System.Windows.FrameworkTemplate.Resources](#) dictionary of the element is checked.
2. The lookup traverses the logical tree upward to the parent element and its resource dictionary. This process continues until the root element is reached.
3. App resources are checked. App resources are those resources within the resource dictionary that are defined by the [Application](#) object for your WPF app.
4. The theme resource dictionary is checked for the currently active theme. If the theme changes at runtime, the value is reevaluated.
5. System resources are checked.

Exception behavior (if any) varies:

- If a resource was requested by a [FindResource](#) call and wasn't found, an exception is thrown.
- If a resource was requested by a [TryFindResource](#) call and wasn't found, no exception is thrown, and the returned value is `null`. If the property being set doesn't accept `null`, then it's still possible that a deeper exception will be thrown, depending on the individual property being set.
- If a resource was requested by a dynamic resource reference in XAML and wasn't found, then the behavior depends on the general property system. The general behavior is as if no property setting operation occurred at the level where the resource exists. For instance, if you attempt to set the background on an individual button element using a resource that could not be evaluated, then no value set results, but the effective value can still come from other participants in the property system and value precedence. For instance, the background value might still come from a locally defined button style or from the theme style. For properties that aren't defined by theme styles, the effective value after a failed resource evaluation might come from the default value in the property metadata.

Restrictions

Dynamic resource references have some notable restrictions. At least one of the following conditions must be true:

- The property being set must be a property on a [FrameworkElement](#) or [FrameworkContentElement](#). That property must be backed by a [DependencyProperty](#).
- The reference is for a value within a `StyleSetter`.
- The property being set must be a property on a [Freezable](#) that is provided as a value of either a [FrameworkElement](#) or [FrameworkContentElement](#) property, or a [Setter](#) value.

Because the property being set must be a [DependencyProperty](#) or [Freezable](#) property, most property changes can propagate to the UI because a property change (the changed dynamic resource value) is acknowledged by the property system. Most controls include logic that will force another layout of a control if a [DependencyProperty](#) changes and that property might affect layout. However, not all properties that have a [DynamicResource Markup Extension](#) as their value are guaranteed to provide real-time updates in the UI. That functionality still might vary depending on the property, and depending on the type that owns the property, or even the logical structure of your app.

Styles, DataTemplates, and implicit keys

Although all items in a [ResourceDictionary](#) must have a key, that doesn't mean that all resources must have an explicit `x:Key`. Several object types support an implicit key when defined as a resource, where the key value is tied to the value of another property. This type of key is known as an implicit key, and an `x:Key` attribute is an explicit key. You can overwrite any implicit key by specifying an explicit key.

One important scenario for resources is when you define a [Style](#). In fact, a [Style](#) is almost always defined as an entry in a resource dictionary, because styles are inherently intended for reuse. For more information about styles, see [Styles and templates \(WPF .NET\)](#).

Styles for controls can be both created with and referenced with an implicit key. The theme styles that define the default appearance of a control rely on this implicit key. From the standpoint of requesting it, the implicit key is the [Type](#) of the control itself. From the standpoint of defining the resources, the implicit key is the [TargetType](#) of the style. As such, if you're creating themes for custom controls or creating styles that interact with existing theme styles, you don't need to specify an [x:Key Directive](#) for that [Style](#). And if you want to use the themed styles, you don't need to specify any style at all. For instance, the following style definition works, even though the [Style](#) resource doesn't appear to have a key:

```
<Style TargetType="Button">
  <Setter Property="Background" Value="#4E1A3D" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="BorderThickness" Value="5" />
  <Setter Property="BorderBrush">
    <Setter.Value>
      <LinearGradientBrush>
        <GradientStop Offset="0.0" Color="#4E1A3D"/>
        <GradientStop Offset="1.0" Color="Salmon"/>
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

That style really does have a key: the implicit key: the `System.Windows.Controls.Button` type. In markup, you can specify a [TargetType](#) directly as the type name (or you can optionally use `{x:Type...}` to return a [Type](#).

Through the default theme style mechanisms used by WPF, that style is applied as the runtime style of a [Button](#)

on the page, even though the [Button](#) itself doesn't attempt to specify its [Style](#) property or a specific resource reference to the style. Your style defined in the page is found earlier in the lookup sequence than the theme dictionary style, using the same key that the theme dictionary style has. You could just specify `<Button>Hello</Button>` anywhere in the page, and the style you defined with [TargetType](#) of `Button` would apply to that button. If you want, you can still explicitly key the style with the same type value as [TargetType](#) for clarity in your markup, but that is optional.

Implicit keys for styles don't apply on a control if [OverridesDefaultStyle](#) is `true`. (Also note that [OverridesDefaultStyle](#) might be set as part of native behavior for the control class, rather than explicitly on an instance of the control.) Also, to support implicit keys for derived class scenarios, the control must override [DefaultStyleKey](#) (all existing controls provided as part of WPF include this override). For more information about styles, themes, and control design, see [Guidelines for Designing Stylable Controls](#).

[DataTemplate](#) also has an implicit key. The implicit key for a [DataTemplate](#) is the [DataType](#) property value. [DataType](#) can also be specified as the name of the type rather than explicitly using `{x:Type...}`. For details, see [Data Templating Overview](#).

See also

- [Resources in code](#)
- [Merged resource dictionaries](#)
- [How to define and reference a WPF resource](#)
- [How to use system resources](#)
- [How to use application resources](#)
- [x:Type markup extension](#)
- [ResourceDictionary](#)
- [Application resources](#)
- [Define and reference a resource](#)
- [Application management overview](#)
- [StaticResource markup extension](#)
- [DynamicResource markup extension](#)

Merged resource dictionaries (WPF .NET)

4/15/2021 • 5 minutes to read • [Edit Online](#)

Windows Presentation Foundation (WPF) resources support a merged resource dictionary feature. This feature provides a way to define the resources portion of a WPF application outside of the compiled XAML application. Resources can then be shared across applications and are also more conveniently isolated for localization.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Create a merged dictionary

In markup, you use the following syntax to introduce a merged resource dictionary into a page:

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="myresourcedictionary.xaml"/>
      <ResourceDictionary Source="myresourcedictionary2.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Page.Resources>
```

The [ResourceDictionary](#) element doesn't have an [x:Key Directive](#), which is generally required for all items in a resource collection. But another [ResourceDictionary](#) reference within the [MergedDictionaries](#) collection is a special case, reserved for this merged resource dictionary scenario. Further, the [ResourceDictionary](#) that introduces a merged resource dictionary can't have an [x:Key Directive](#).

Typically, each [ResourceDictionary](#) within the [MergedDictionaries](#) collection specifies a [Source](#) attribute. The value of [Source](#) should be a uniform resource identifier (URI) that resolves to the location of the resources file to be merged. The destination of that URI must be another XAML file, with [ResourceDictionary](#) as its root element.

NOTE

It's legal to define resources within a [ResourceDictionary](#) that's specified as a merged dictionary, either as an alternative to specifying [Source](#), or in addition to whatever resources are included from the specified source. However, this isn't a common scenario. The main scenario for merged dictionaries is to merge resources from external file locations. If you want to specify resources within the markup for a page, define these in the main [ResourceDictionary](#) and not in the merged dictionaries.

Merged dictionary behavior

Resources in a merged dictionary occupy a location in the resource lookup scope that's just after the scope of the main resource dictionary they are merged into. Although a resource key must be unique within any individual dictionary, a key can exist multiple times in a set of merged dictionaries. In this case, the resource that's returned will come from the last dictionary found sequentially in the [MergedDictionaries](#) collection. If the [MergedDictionaries](#) collection was defined in XAML, then the order of the merged dictionaries in the collection is the order of the elements as provided in the markup. If a key is defined in the primary dictionary and also in a

dictionary that was merged, then the resource that's returned will come from the primary dictionary. These scoping rules apply equally for both static resource references and dynamic resource references.

Merged dictionaries and code

Merged dictionaries can be added to a `Resources` dictionary through code. The default, initially empty `ResourceDictionary` that exists for any `Resources` property also has a default, initially empty `MergedDictionaries` collection property. To add a merged dictionary through code, you obtain a reference to the desired primary `ResourceDictionary`, get its `MergedDictionaries` property value, and call `Add` on the generic `Collection` that's contained in `MergedDictionaries`. The object you add must be a new `ResourceDictionary`.

In code, you don't set the `Source` property. Instead, you must obtain a `ResourceDictionary` object by either creating one or loading one. One way to load an existing `ResourceDictionary` is to call `XamlReader.Load` on an existing XAML file stream that has a `ResourceDictionary` root, then casting the return value to `ResourceDictionary`.

Merged dictionary URIs

There are several techniques for how to include a merged resource dictionary, which are indicated by the uniform resource identifier (URI) format that you use. Broadly speaking, these techniques can be divided into two categories: resources that are compiled as part of the project, and resources that aren't compiled as part of the project.

For resources that are compiled as part of the project, you can use a relative path that refers to the resource location. The relative path is evaluated during compilation. Your resource must be defined as part of the project as a **Resource** build action. If you include a resource `.xaml` file in the project as **Resource**, you don't need to copy the resource file to the output directory, the resource is already included within the compiled application. You can also use **Content** build action, but you must then copy the files to the output directory and also deploy the resource files in the same path relationship to the executable.

NOTE

Don't use the **Embedded Resource** build action. The build action itself is supported for WPF applications, but the resolution of `Source` doesn't incorporate `ResourceManager`, and thus cannot separate the individual resource out of the stream. You could still use **Embedded Resource** for other purposes so long as you also used `ResourceManager` to access the resources.

A related technique is to use a **Pack URI** to a XAML file, and refer to it as **Source**. **Pack URI** enables references to components of referenced assemblies and other techniques. For more information on **Pack URIs**, see [WPF Application Resource, Content, and Data Files](#).

For resources that aren't compiled as part of the project, the URI is evaluated at run time. You can use a common URI transport such as `file:` or `http:` to refer to the resource file. The disadvantage of using the non-compiled resource approach is that `file:` access requires additional deployment steps, and `http:` access implies the Internet security zone.

Reusing merged dictionaries

You can reuse or share merged resource dictionaries between applications, because the resource dictionary to merge can be referenced through any valid uniform resource identifier (URI). Exactly how you do this depends on your application deployment strategy and which application model you follow. The [previously mentioned Pack URI strategy](#) provides a way to commonly source a merged resource across multiple projects during development by sharing an assembly reference. In this scenario the resources are still distributed by the client,

and at least one of the applications must deploy the referenced assembly. It's also possible to reference merged resources through a distributed URI that uses the *http:* protocol.

Writing merged dictionaries as local application files or to local shared storage is another possible merged dictionary and application deployment scenario.

Localization

If resources that need to be localized are isolated to dictionaries that are merged into primary dictionaries, and kept as loose XAML, these files can be localized separately. This technique is a lightweight alternative to localizing the satellite resource assemblies. For details, see [WPF Globalization and Localization Overview](#).

See also

- [ResourceDictionary](#)
- [Overview of XAML resources](#)
- [Resources in code](#)
- [WPF Application Resource, Content, and Data Files](#)
- [WPF Globalization and Localization Overview](#)

Resources in code (WPF .NET)

4/15/2021 • 4 minutes to read • [Edit Online](#)

This overview concentrates on how Windows Presentation Foundation (WPF) resources can be accessed or created using code rather than XAML syntax. For more information on general resource usage and resources from a XAML syntax perspective, see [Overview of XAML resources](#).

Accessing resources from code

The keys that identify XAML defined resources are also used to retrieve specific resources if you request the resource in code. The simplest way to retrieve a resource from code is to call either the [FindResource](#) or the [TryFindResource](#) method from framework-level objects in your application. The behavioral difference between these methods is what happens if the requested key isn't found. [FindResource](#) raises an exception.

[TryFindResource](#) won't raise an exception but returns `null`. Each method takes the resource key as an input parameter, and returns a loosely typed object.

Typically, a resource key is a string, but there are [occasional nonstring usages](#). The lookup logic for code resource resolution is the same as the dynamic resource reference XAML case. The search for resources starts from the calling element, then continues through parent elements in the logical tree. The lookup continues onwards into application resources, themes, and system resources if necessary. A code request for a resource will properly account for changes to those resources that happened during runtime.

The following code example demonstrates a [Click](#) event handler that finds a resource by key, and uses the returned value to set a property.

```
private void myButton_Click(object sender, RoutedEventArgs e)
{
    Button button = (Button)sender;
    button.Background = (Brush)this.FindResource("RainbowBrush");
}
```

```
Private Sub myButton_Click(sender As Object, e As RoutedEventArgs)
    Dim buttonControl = DirectCast(sender, Button)
    buttonControl.Background = DirectCast(Me.FindResource("RainbowBrush"), Brush)
End Sub
```

An alternative method for assigning a resource reference is [SetResourceReference](#). This method takes two parameters: the key of the resource, and the identifier for a particular dependency property that's present on the element instance to which the resource value should be assigned. Functionally, this method is the same and has the advantage of not requiring any casting of return values.

Still another way to access resources programmatically is to access the contents of the [Resources](#) property as a dictionary. Resource dictionaries are used to add new resources to existing collections, check to see if a given key name is already used by the collection, and other operations. If you're writing a WPF application entirely in code, you can also create the entire collection in code, assign resources to it. The collection can then be assigned to the [Resources](#) property of an element. This is described in the next section.

You can index within any given [Resources](#) collection, using a specific key as the index. Resources accessed in this way don't follow the normal runtime rules of resource resolution. You're only accessing that particular collection. Resource lookup doesn't traverse the resource scope to the root or the application if no valid object was found at the requested key. However, this approach may have performance advantages in some cases precisely

because the scope of the search for the key is more constrained. For more information about how to work with a resource dictionary directly, see the [ResourceDictionary](#) class.

Creating resources with code

If you want to create an entire WPF application in code, you might also want to create any resources in that application in code. To achieve this, create a new [ResourceDictionary](#) instance, and then add all the resources to the dictionary using successive calls to [ResourceDictionary.Add](#). Then, assign the created [ResourceDictionary](#) to set the [Resources](#) property on an element that's present in a page scope, or the [Application.Resources](#). You could also maintain the [ResourceDictionary](#) as a standalone object without adding it to an element. However, if you do this, you must access the resources within it by item key, as if it were a generic dictionary. A [ResourceDictionary](#) that's not attached to an element `Resources` property wouldn't exist as part of the element tree and has no scope in a lookup sequence that can be used by [FindResource](#) and related methods.

Using objects as keys

Most resource usages will set the key of the resource to be a string. However, various WPF features deliberately use the object type as a key instead of a string. The capability of having the resource be keyed by an object type is used by the WPF style and theming support. The styles and themes that become the default for an otherwise non-styled control are each keyed by the [Type](#) of the control that they should apply to.

Being keyed by type provides a reliable lookup mechanism that works on default instances of each control type. The type can be detected by reflection and used for styling derived classes even though the derived type otherwise has no default style. You can specify a [Type](#) key for a resource defined in XAML by using the [x:Type Markup Extension](#). Similar extensions exist for other nonstring key usages that support WPF features, such as [ComponentResourceKey Markup Extension](#).

For more information, see [Styles](#), [DataTemplates](#), and [implicit keys](#).

See also

- [Overview of XAML resources](#)
- [How to define and reference a WPF resource](#)
- [How to use system resources](#)
- [How to use application resources](#)

How to define and reference a WPF resource (WPF .NET)

4/15/2021 • 5 minutes to read • [Edit Online](#)

This example shows how to define a resource and reference it. A resource can be referenced through XAML or through code.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

XAML example

The following example defines two types of resources: a [SolidColorBrush](#) resource, and several [Style](#) resources.

```
<Window.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
  <Style TargetType="Border">
    <Setter Property="Background" Value="#4E1A3D" />
    <Setter Property="BorderThickness" Value="5" />
    <Setter Property="BorderBrush">
      <Setter.Value>
        <LinearGradientBrush>
          <GradientStop Offset="0.0" Color="#4E1A3D"/>
          <GradientStop Offset="1.0" Color="Salmon"/>
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
  <Style TargetType="TextBlock" x:Key="TitleText">
    <Setter Property="FontSize" Value="18"/>
    <Setter Property="Foreground" Value="#4E87D4"/>
    <Setter Property="FontFamily" Value="Trebuchet MS"/>
    <Setter Property="Margin" Value="0,10,10,10"/>
  </Style>
  <Style TargetType="TextBlock" x:Key="Label">
    <Setter Property="HorizontalAlignment" Value="Right"/>
    <Setter Property="FontSize" Value="13"/>
    <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
    <Setter Property="FontFamily" Value="Arial"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="Margin" Value="0,3,10,0"/>
  </Style>
</Window.Resources>
```

Resources

The [SolidColorBrush](#) resource `MyBrush` is used to provide the value of several properties that each take a [Brush](#) type value. This resource is referenced through the `x:Key` value.

```

<Border>
  <StackPanel>
    <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
    <TextBlock Style="{StaticResource Label}">Label</TextBlock>
    <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="{StaticResource MyBrush}"
Text="Text" Margin="20" />
    <Button HorizontalAlignment="Left" Height="30" Background="{StaticResource MyBrush}"
Margin="40">Button</Button>
    <Ellipse HorizontalAlignment="Center" Width="100" Height="100" Fill="{StaticResource MyBrush}"
Margin="10" />
  </StackPanel>
</Border>

```

In the previous example, the `MyBrush` resource is accessed with the [StaticResource Markup Extension](#). The resource is assigned to a property that can accept the type of resource being defined. In this case the [Background](#), [Foreground](#), and [Fill](#) properties.

All resources in a resource dictionary must provide a key. When styles are defined though, they can omit the key, as explained in the [next section](#).

Resources are also requested by the order found within the dictionary if you use the [StaticResource Markup Extension](#) to reference them from within another resource. Make sure that any resource that you reference is defined in the collection earlier than where that resource is requested. For more information, see [Static resources](#).

If necessary, you can work around the strict creation order of resource references by using a [DynamicResource Markup Extension](#) to reference the resource at runtime instead, but you should be aware that this `DynamicResource` technique has performance consequences. For more information, see [Dynamic resources](#).

Style resources

The following example references styles implicitly and explicitly:

```

<Border>
  <StackPanel>
    <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
    <TextBlock Style="{StaticResource Label}">Label</TextBlock>
    <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="{StaticResource MyBrush}"
Text="Text" Margin="20" />
    <Button HorizontalAlignment="Left" Height="30" Background="{StaticResource MyBrush}"
Margin="40">Button</Button>
    <Ellipse HorizontalAlignment="Center" Width="100" Height="100" Fill="{StaticResource MyBrush}"
Margin="10" />
  </StackPanel>
</Border>

```

In the previous code example, the [Style](#) resources `TitleText` and `Label`, each target a particular control type. In this case, they both target a [TextBlock](#). The styles set a variety of different properties on the targeted controls when that style resource is referenced by its resource key for the [Style](#) property.

The style though that targets a [Border](#) control doesn't define a key. When a key is omitted, the type of object being targeted by the [TargetType](#) property is implicitly used as the key for the style. When a style is keyed to a type, it becomes the default style for all controls of that type, as long as these controls are within scope of the style. For more information, see [Styles, DataTemplates, and implicit keys](#).

Code examples

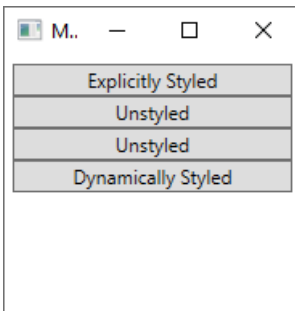
The following code snippets demonstrate creating and setting resources through code

Create a style resource

Creating a resource and assigning it to a resource dictionary can happen at any time. However, only XAML elements that use the `DynamicResource` syntax will be automatically updated with the resource after it's created.

Take for example the following Window. It has four buttons. The forth button is using a [DynamicResource](#) to style itself. However, this resource doesn't yet exist, so it just looks like a normal button:

```
<StackPanel Margin="5">
    <Button Click="Button_Click">Explicitly Styled</Button>
    <Button>Unstyled</Button>
    <Button>Unstyled</Button>
    <Button Style="{DynamicResource ResourceKey=buttonStyle1}">Dynamically Styled</Button>
</StackPanel>
```



The following code is invoked when the first button is clicked and performs the following tasks:

- Creates some colors for easy reference.
- Creates a new style.
- Assigns setters to the style.
- Adds the style as a resource named `buttonStyle1` to the window's resource dictionary.
- Assigns the style directly to the button raising the `Click` event.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create colors
    Color purple = (Color)ColorConverter.ConvertFromString("#4E1A3D");
    Color white = Colors.White;
    Color salmon = Colors.Salmon;

    // Create a new style for a button
    var buttonStyle = new Style(typeof(Button));

    // Set the properties of the style
    buttonStyle.Setters.Add(new Setter(Control.BackgroundProperty, new SolidColorBrush(purple)));
    buttonStyle.Setters.Add(new Setter(Control.ForegroundProperty, new SolidColorBrush(white)));
    buttonStyle.Setters.Add(new Setter(Control.BorderBrushProperty, new LinearGradientBrush(purple, salmon,
45d)));
    buttonStyle.Setters.Add(new Setter(Control.BorderThicknessProperty, new Thickness(5)));

    // Set this style as a resource. Any DynamicResource tied to this key will be updated.
    this.Resources["buttonStyle1"] = buttonStyle;

    // Set this style directly to a button
    ((Button)sender).Style = buttonStyle;
}
```

```

Private Sub Button_Click(sender As Object, e As RoutedEventArgs)

    'Create colors
    Dim purple = DirectCast(ColorConverter.ConvertFromString("#4E1A3D"), Color)
    Dim white = Colors.White
    Dim salmon = Colors.Salmon

    'Create a new style for a button
    Dim buttonStyle As New Style()

    'Set the properties of the style
    buttonStyle.Setters.Add(New Setter(Control.BackgroundProperty, New SolidColorBrush(purple)))
    buttonStyle.Setters.Add(New Setter(Control.ForegroundProperty, New SolidColorBrush(white)))
    buttonStyle.Setters.Add(New Setter(Control.BorderBrushProperty, New LinearGradientBrush(purple, salmon,
45D)))
    buttonStyle.Setters.Add(New Setter(Control.BorderThicknessProperty, New Thickness(5)))

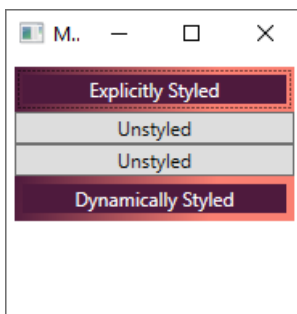
    'Set this style as a resource. Any DynamicResource looking for this key will be updated.
    Me.Resources("buttonStyle1") = buttonStyle

    'Set this style directly to a button
    DirectCast(sender, Button).Style = buttonStyle

End Sub

```

After the code runs, the window is updated:



Notice that the forth button's style was updated. The style was automatically applied because the button used the [DynamicResource Markup Extension](#) to reference a style that didn't yet exist. Once the style was created and added to the resources of the window, it was applied to the button. For more information, see [Dynamic resources](#).

Find a resource

The following code traverses the logical tree of the XAML object in which is run, to find the specified resource. The resource might be defined on the object itself, it's parent, all the way to the root, the application itself. The following code searches for a resource, starting with the button itself:

```
myButton.Style = myButton.TryFindResource("buttonStyle1") as Style;
```

```
myButton.Style = myButton.TryFindResource("buttonStyle1")
```

Explicitly reference a resource

When you have reference to a resource, either by searching for it or by creating it, it can be assigned to a property directly:

```
// Set this style as a resource. Any DynamicResource tied to this key will be updated.  
this.Resources["buttonStyle1"] = buttonStyle;
```

```
'Set this style as a resource. Any DynamicResource looking for this key will be updated.  
Me.Resources("buttonStyle1") = buttonStyle
```

See also

- [Overview of XAML resources](#)
- [Styles and templates](#)
- [How to use system resources](#)
- [How to use application resources](#)

How to use application resources (WPF .NET)

4/15/2021 • 2 minutes to read • [Edit Online](#)

This example demonstrates how to use application-defined resources. Resources can be defined at the application level, generally through the *App.xaml* or *Application.xaml* file, whichever one your project uses. Resources that are defined by the application are globally scoped and accessible by all parts of the application.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Example

The following example shows an application definition file. The application definition file defines a resource section (a value for the [Resources](#) property). Resources defined at the application level can be accessed by all other pages that are part of the application. In this case, the resource is a declared style. Because a complete style that includes a control template can be lengthy, this example omits the control template that is defined within the [ContentTemplate](#) property setter of the style.

```
<Application.Resources>
  <Style TargetType="Border" x:Key="FancyBorder">
    <Setter Property="Background" Value="#4E1A3D" />
    <Setter Property="BorderThickness" Value="5" />
    <Setter Property="BorderBrush">
      <Setter.Value>
        <LinearGradientBrush>
          <GradientStop Offset="0.0" Color="#4E1A3D"/>
          <GradientStop Offset="1.0" Color="Salmon"/>
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
</Application.Resources>
```

The following example shows a XAML page that references an application-level resource from the previous example. The resource is referenced with a [StaticResource Markup Extension](#) that specifies the unique resource key for the resource. The resource "FancyBorder" isn't found in the scope of the current object and window, so the resource lookup continues beyond the current page and into application-level resources.

```
<Border Style="{StaticResource FancyBorder}">
  <StackPanel Margin="5">
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </StackPanel>
</Border>
```

See also

- [Overview of XAML resources](#)
- [Resources in code](#)

- [How to define and reference a WPF resource](#)
- [How to use system resources](#)

How to use system resources (WPF .NET)

4/15/2021 • 3 minutes to read • [Edit Online](#)

This example demonstrates how to use system-defined resources. System resources are provided by WPF and allow access to operating system resources, such as fonts, colors, and icons. System resources expose several system-defined values as both resources and properties to help you create visuals that are consistent with Windows.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Fonts

Use the [SystemFonts](#) class to reference the fonts used by the operating system. This class contains system font values as static properties, and properties that reference resource keys that can be used to access those values dynamically at run time. For example, [CaptionFontFamily](#) is a [SystemFonts](#) value, and [CaptionFontFamilyKey](#) is a corresponding resource key.

The following example shows how to access and use the properties of [SystemFonts](#) as static values to style or customize a text block:

```
<TextBlock FontSize="{x:Static SystemFonts.SmallCaptionFontSize}"
           FontWeight="{x:Static SystemFonts.SmallCaptionFontWeight}"
           FontFamily="{x:Static SystemFonts.SmallCaptionFontFamily}"
           Text="Small Caption Font">
</TextBlock>
```

To use the values of [SystemFonts](#) in code, you don't have to use either a static value or a dynamic resource reference. Instead, use the non-key properties of the [SystemFonts](#) class. Although the non-key properties are apparently defined as static properties, the run-time behavior of WPF as hosted by the system will reevaluate the properties in real time and will properly account for user-driven changes to system values. The following example shows how to specify the font settings of a button:

```
var myButton = new Button()
{
    Content = "SystemFonts",
    Background = SystemColors.ControlDarkDarkBrush,
    FontSize = SystemFonts.IconFontSize,
    FontWeight = SystemFonts.MessageFontWeight,
    FontFamily = SystemFonts.CaptionFontFamily
};

mainStackPanel.Children.Add(myButton);
```

```
Dim myButton = New Button() With
{
    .Content = "SystemFonts",
    .Background = SystemColors.ControlDarkDarkBrush,
    .FontSize = SystemFonts.IconFontSize,
    .FontWeight = SystemFonts.MessageFontWeight,
    .FontFamily = SystemFonts.CaptionFontFamily
}

mainStackPanel.Children.Add(myButton)
```

Dynamic fonts in XAML

System font metrics can be used as either static or dynamic resources. Use a dynamic resource if you want the font metric to update automatically while the application runs; otherwise, use a static resource.

NOTE

Dynamic resources have the keyword `key` appended to the property name.

The following example shows how to access and use system font dynamic resources to style or customize a text block:

```
<TextBlock FontSize="{DynamicResource {x:Static SystemFonts.SmallCaptionFontSize}}"
           FontWeight="{DynamicResource {x:Static SystemFonts.SmallCaptionFontWeight}}"
           FontFamily="{DynamicResource {x:Static SystemFonts.SmallCaptionFontFamily}}"
           Text="Small Caption Font">
</TextBlock>
```

Parameters

Use the [SystemParameters](#) class to reference system-level properties, such as the size of the primary display. This class contains both system parameter value properties, and resource keys that bind to the values. For example, [FullPrimaryScreenHeight](#) is a [SystemParameters](#) property value and [FullPrimaryScreenHeightKey](#) is the corresponding resource key.

The following example shows how to access and use the static values of [SystemParameters](#) to style or customize a button. This markup example sizes a button by applying [SystemParameters](#) values to a button:

```
<Button FontSize="8"
        Height="{x:Static SystemParameters.CaptionHeight}"
        Width="{x:Static SystemParameters.IconGridWidth}"
        Content="System Parameters">
</Button>
```

To use the values of [SystemParameters](#) in code, you don't have to use either static references or dynamic resource references. Instead, use the values of the [SystemParameters](#) class. Although the non-key properties are apparently defined as static properties, the run-time behavior of WPF as hosted by the system will reevaluate the properties in real time, and will properly account for user-driven changes to system values. The following example shows how to set the width and height of a button by using [SystemParameters](#) values:

```
var myButton = new Button()
{
    Content = "SystemParameters",
    FontSize = 8,
    Background = SystemColors.ControlDarkDarkBrush,
    Height = SystemParameters.CaptionHeight,
    Width = SystemParameters.CaptionWidth,
};

mainStackPanel.Children.Add(myButton);
```

```
Dim myButton = New Button() With
{
    .Content = "SystemParameters",
    .FontSize = 8,
    .Background = SystemColors.ControlDarkDarkBrush,
    .Height = SystemParameters.CaptionHeight,
    .Width = SystemParameters.CaptionWidth
}

mainStackPanel.Children.Add(myButton)
```

Dynamic parameters in XAML

System parameter metrics can be used as either static or dynamic resources. Use a dynamic resource if you want the parameter metric to update automatically while the application runs; otherwise, use a static resource.

NOTE

Dynamic resources have the keyword `key` appended to the property name.

The following example shows how to access and use system parameter dynamic resources to style or customize a button. This XAML example sizes a button by assigning [SystemParameters](#) values to the button's width and height.

```
<Button FontSize="8"
        Height="{DynamicResource {x:Static SystemParameters.CaptionHeightKey}}"
        Width="{DynamicResource {x:Static SystemParameters.IconGridWidthKey}}"
        Content="System Parameters">
</Button>
```

See also

- [Overview of XAML resources](#)
- [Resources in code](#)
- [How to define and reference a WPF resource](#)
- [How to use application resources](#)

XAML overview (WPF .NET)

4/15/2021 • 25 minutes to read • [Edit Online](#)

This article describes the features of the XAML language and demonstrates how you can use XAML to write Windows Presentation Foundation (WPF) apps. This article specifically describes XAML as implemented by WPF. XAML itself is a larger language concept than WPF.

IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

What is 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. You 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. This is unlike most other markup languages, which are typically an interpreted language without such a direct tie to a backing type system. 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 following example shows how you might create a button as part of a UI. This example is intended to give you a flavor of how XAML represents common UI programming metaphors (it's not a complete sample).

```
<StackPanel>
  <Button Content="Click Me"/>
</StackPanel>
```

XAML syntax in brief

The following sections explain the basic forms of XAML syntax, and give a short markup example. These sections aren't intended to provide complete information about each syntax form, such as how these are represented in the backing type system. For more information about the specifics of XAML syntax, see [XAML Syntax In Detail](#).

Much of the material in the next few sections will be elementary to you if you have previous familiarity with the XML language. This is a consequence of one of the basic design principles of XAML. The XAML language defines concepts of its own, but these concepts work within the XML language and markup form.

XAML object elements

An object element typically declares an instance of a type. That type is defined in the assemblies referenced by the technology that uses XAML as a language.

Object element syntax always starts with an opening angle bracket (`<`). This is followed by the name of the type where you want to create an instance. (The name can include a prefix, a concept that will be explained later.) After this, you can optionally declare attributes on the object element. To complete the object element tag, end with a closing angle bracket (`>`). You can instead use a self-closing form that doesn't have any content, by completing the tag with a forward slash and closing angle bracket in succession (`/>`). For example, look at the

previously shown markup snippet again.

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

This specifies two object elements: `<StackPanel>` (with content, and a closing tag later), and `<Button .../>` (the self-closing form, with several attributes). The object elements `StackPanel` and `Button` each map to the name of a class that is defined by WPF and is part of the WPF assemblies. When you specify an object element tag, you create an instruction for XAML processing to create a new instance of the underlying type. Each instance is created by calling the parameterless constructor of the underlying type when parsing and loading the XAML.

Attribute syntax (properties)

Properties of an object can often be expressed as attributes of the object element. The attribute syntax names the object property that is being set, followed by the assignment operator (=). The value of an attribute is always specified as a string that is contained within quotation marks.

Attribute syntax is the most streamlined property setting syntax and is the most intuitive syntax to use for developers who have used markup languages in the past. For example, the following markup creates a button that has red text and a blue background with a display text of `Content`.

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

Property element syntax

For some properties of an object element, attribute syntax isn't possible, because the object or information necessary to provide the property value can't be adequately expressed within the quotation mark and string restrictions of attribute syntax. For these cases, a different syntax known as property element syntax can be used.

The syntax for the property element start tag is `<TypeName.PropertyName>`. Generally, the content of that tag is an object element of the type that the property takes as its value. After specifying the content, you must close the property element with an end tag. The syntax for the end tag is `</TypeName.PropertyName>`.

If an attribute syntax is possible, using the attribute syntax is typically more convenient and enables a more compact markup, but that is often just a matter of style, not a technical limitation. The following example shows the same properties being set as in the previous attribute syntax example, but this time by using property element syntax for all properties of the `Button`.

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

Collection syntax

The XAML language includes some optimizations that produce more human-readable markup. One such optimization is that if a particular property takes a collection type, then items that you declare in markup as child elements within that property's value become part of the collection. In this case, a collection of child object

elements is the value being set to the collection property.

The following example shows collection syntax for setting values of the [GradientStops](#) property.

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <!-- no explicit new GradientStopCollection, parser knows how to find or create -->
    <GradientStop Offset="0.0" Color="Red" />
    <GradientStop Offset="1.0" Color="Blue" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

XAML content properties

XAML specifies a language feature whereby a class can designate exactly one of its properties to be the XAML *content* property. Child elements of that object element are used to set the value of that content property. In other words, for the content property uniquely, you can omit a property element when setting that property in XAML markup and produce a more visible parent/child metaphor in the markup.

For example, [Border](#) specifies a *content* property of [Child](#). The following two [Border](#) elements are treated identically. The first one takes advantage of the content property syntax and omits the `Border.Child` property element. The second one shows `Border.Child` explicitly.

```
<Border>
  <TextBox Width="300"/>
</Border>
<!--explicit equivalent-->
<Border>
  <Border.Child>
    <TextBox Width="300"/>
  </Border.Child>
</Border>
```

As a rule of the XAML language, the value of a XAML content property must be given either entirely before or entirely after any other property elements on that object element. For instance, the following markup doesn't compile.

```
<Button>I am a
  <Button.Background>Blue</Button.Background>
blue button</Button>
```

For more information about the specifics of XAML syntax, see [XAML Syntax In Detail](#).

Text content

A small number of XAML elements can directly process text as their content. To enable this, one of the following cases must be true:

- The class must declare a content property, and that content property must be of a type assignable to a string (the type could be [Object](#)). For instance, any [ContentControl](#) uses [Content](#) as its content property and it's type [Object](#), and this supports the following usage on a [ContentControl](#) such as a [Button](#):
`<Button>Hello</Button>`.
- The type must declare a type converter, in which case the text content is used as initialization text for that type converter. For example, `<Brush>Blue</Brush>` converts the content value of `Blue` into a brush. This case is less common in practice.
- The type must be a known XAML language primitive.

Content properties and collection syntax combined

Consider this example.

```
<StackPanel>
    <Button>First Button</Button>
    <Button>Second Button</Button>
</StackPanel>
```

Here, each [Button](#) is a child element of [StackPanel](#). This is a streamlined and intuitive markup that omits two tags for two different reasons.

- **Omitted `StackPanel.Children` property element:** [StackPanel](#) derives from [Panel](#). [Panel](#) defines [Panel.Children](#) as its XAML content property.
- **Omitted `UIElementCollection` object element:** The [Panel.Children](#) property takes the type [UIElementCollection](#), which implements [IList](#). The collection's element tag can be omitted, based on the XAML rules for processing collections such as [IList](#). (In this case, [UIElementCollection](#) actually can't be instantiated because it doesn't expose a parameterless constructor, and that is why the [UIElementCollection](#) object element is shown commented out).

```
<StackPanel>
    <StackPanel.Children>
        <Button>First Button</Button>
        <Button>Second Button</Button>
    </StackPanel.Children>
</StackPanel>
```

Attribute syntax (events)

Attribute syntax can also be used for members that are events rather than properties. In this case, the attribute's name is the name of the event. In the WPF implementation of events for XAML, the attribute's value is the name of a handler that implements that event's delegate. For example, the following markup assigns a handler for the [Click](#) event to a [Button](#) created in markup:

```
<Button Click="Button_Click" >Click Me!</Button>
```

There's more to events and XAML in WPF than just this example of the attribute syntax. For example, you might wonder what the `ClickHandler` referenced here represents and how it's defined. This will be explained in the upcoming [Events and XAML code-behind](#) section of this article.

Case and white space in XAML

In general, XAML is case-sensitive. For purposes of resolving backing types, WPF XAML is case-sensitive by the same rules that the CLR is case-sensitive. Object elements, property elements, and attribute names must all be specified by using the sensitive casing when compared by name to the underlying type in the assembly, or to a member of a type. XAML language keywords and primitives are also case-sensitive. Values aren't always case-sensitive. Case sensitivity for values will depend on the type converter behavior associated with the property that takes the value, or the property value type. For example, properties that take the [Boolean](#) type can take either `true` or `True` as equivalent values, but only because the native WPF XAML parser type conversion for string to [Boolean](#) already permits these as equivalents.

WPF XAML processors and serializers will ignore or drop all nonsignificant white space, and will normalize any significant white space. This is consistent with the default white-space behavior recommendations of the XAML specification. This behavior is only of consequence when you specify strings within XAML content properties. In simplest terms, XAML converts space, linefeed, and tab characters into spaces, and then preserves one space if

found at either end of a contiguous string. The full explanation of XAML white-space handling isn't covered in this article. For more information, see [White space processing in XAML](#).

Markup extensions

Markup extensions are a XAML language concept. When used to provide the value of an attribute syntax, curly braces (`{` and `}`) indicate a markup extension usage. This usage directs the XAML processing to escape from the general treatment of attribute values as either a literal string or a string-convertible value.

The most common markup extensions used in WPF app programming are [Binding](#), used for data binding expressions, and the resource references [StaticResource](#) and [DynamicResource](#). By using markup extensions, you can use attribute syntax to provide values for properties even if that property doesn't support an attribute syntax in general. Markup extensions often use intermediate expression types to enable features such as deferring values or referencing other objects that are only present at run-time.

For example, the following markup sets the value of the [Style](#) property using attribute syntax. The [Style](#) property takes an instance of the [Style](#) class, which by default could not be instantiated by an attribute syntax string. But in this case, the attribute references a particular markup extension, [StaticResource](#). When that markup extension is processed, it returns a reference to a style that was previously instantiated as a keyed resource in a resource dictionary.

```
<Window x:Class="index.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="100" Width="300">
  <Window.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
    <Style TargetType="Border" x:Key="PageBackground">
      <Setter Property="BorderBrush" Value="Blue"/>
      <Setter Property="BorderThickness" Value="5" />
    </Style>
  </Window.Resources>
  <Border Style="{StaticResource PageBackground}">
    <StackPanel>
      <TextBlock Text="Hello" />
    </StackPanel>
  </Border>
</Window>
```

For a reference listing of all markup extensions for XAML implemented specifically in WPF, see [WPF XAML Extensions](#). For a reference listing of the markup extensions that are defined by System.Xaml and are more widely available for .NET Core XAML implementations, see [XAML Namespace \(x:\) Language Features](#). For more information about markup extension concepts, see [Markup Extensions and WPF XAML](#).

Type converters

In the [XAML Syntax in Brief](#) section, it was stated that the attribute value must be able to be set by a string. The basic, native handling of how strings are converted into other object types or primitive values is based on the [String](#) type itself, along with native processing for certain types such as [DateTime](#) or [Uri](#). But many WPF types or members of those types extend the basic string attribute processing behavior in such a way that instances of more complex object types can be specified as strings and attributes.

The [Thickness](#) structure is an example of a type that has a type conversion enabled for XAML usages. [Thickness](#) indicates measurements within a nested rectangle and is used as the value for properties such as [Margin](#). By placing a type converter on [Thickness](#), all properties that use a [Thickness](#) are easier to specify in XAML because they can be specified as attributes. The following example uses a type conversion and attribute syntax to provide a value for a [Margin](#):


```
<Button Margin="10,20,10,30" Content="Click me"/>
```

The previous attribute syntax example is equivalent to the following more verbose syntax example, where the [Margin](#) is instead set through property element syntax containing a [Thickness](#) object element. The four key properties of [Thickness](#) are set as attributes on the new instance:

```
<Button Content="Click me">
  <Button.Margin>
    <Thickness Left="10" Top="20" Right="10" Bottom="30"/>
  </Button.Margin>
</Button>
```

NOTE

There are also a limited number of objects where the type conversion is the only public way to set a property to that type without involving a subclass, because the type itself doesn't have a parameterless constructor. An example is [Cursor](#).

For more information on type conversion, see [TypeConverters and XAML](#).

Root elements and namespaces

A XAML file must have only one root element, to be both a well-formed XML file and a valid XAML file. For typical WPF scenarios, you use a root element that has a prominent meaning in the WPF app model (for example, [Window](#) or [Page](#) for a page, [ResourceDictionary](#) for an external dictionary, or [Application](#) for the app definition). The following example shows the root element of a typical XAML file for a WPF page, with the root element of [Page](#).

```
<Page x:Class="index.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1">

</Page>
```

The root element also contains the attributes `xmlns` and `xmlns:x`. These attributes indicate to a XAML processor which XAML namespaces contain the type definitions for backing types that the markup will reference as elements. The `xmlns` attribute specifically indicates the default XAML namespace. Within the default XAML namespace, object elements in the markup can be specified without a prefix. For most WPF app scenarios, and for almost all of the examples given in the WPF sections of the SDK, the default XAML namespace is mapped to the WPF namespace `http://schemas.microsoft.com/winfx/2006/xaml/presentation`. The `xmlns:x` attribute indicates an additional XAML namespace, which maps the XAML language namespace `http://schemas.microsoft.com/winfx/2006/xaml`.

This usage of `xmlns` to define a scope for usage and mapping of a namespace is consistent with the XML 1.0 specification. XAML namespaces are different from XML namespaces only in that a XAML namespace also implies something about how the namespace's elements are backed by types when it comes to type resolution and parsing the XAML.

The `xmlns` attributes are only strictly necessary on the root element of each XAML file. `xmlns` definitions will apply to all descendant elements of the root element (this behavior is again consistent with the XML 1.0 specification for `xmlns`.) `xmlns` attributes are also permitted on other elements underneath the root, and would apply to any descendant elements of the defining element. However, frequent definition or redefinition of XAML namespaces can result in a XAML markup style that is difficult to read.

The WPF implementation of its XAML processor includes an infrastructure that has awareness of the WPF core assemblies. The WPF core assemblies are known to contain the types that support the WPF mappings to the default XAML namespace. This is enabled through configuration that is part of your project build file and the WPF build and project systems. Therefore, declaring the default XAML namespace as the default `xmlns` is all that is necessary to reference XAML elements that come from WPF assemblies.

The x: prefix

In the previous root element example, the prefix `x:` was used to map the XAML namespace `http://schemas.microsoft.com/winfx/2006/xaml`, which is the dedicated XAML namespace that supports XAML language constructs. This `x:` prefix is used for mapping this XAML namespace in the templates for projects, in examples, and in documentation throughout this SDK. The XAML namespace for the XAML language contains several programming constructs that you will use frequently in your XAML. The following is a listing of the most common `x:` prefix programming constructs you will use:

- **x:Key**: Sets a unique key for each resource in a [ResourceDictionary](#) (or similar dictionary concepts in other frameworks). `x:Key` will probably account for 90 percent of the `x:` usages you will see in a typical WPF app's markup.
- **x:Class**: Specifies the CLR namespace and class name for the class that provides code-behind for a XAML page. You must have such a class to support code-behind per the WPF programming model, and therefore you almost always see `x:` mapped, even if there are no resources.
- **x>Name**: Specifies a run-time object name for the instance that exists in run-time code after an object element is processed. In general, you will frequently use a WPF-defined equivalent property for **x>Name**. Such properties map specifically to a CLR backing property and are thus more convenient for app programming, where you frequently use run-time code to find the named elements from initialized XAML. The most common such property is [FrameworkElement.Name](#). You might still use **x>Name** when the equivalent WPF framework-level [Name](#) property isn't supported in a particular type. This occurs in certain animation scenarios.
- **x:Static**: Enables a reference that returns a static value that isn't otherwise a XAML-compatible property.
- **x:Type**: Constructs a [Type](#) reference based on a type name. This is used to specify attributes that take [Type](#), such as [Style.TargetType](#), although frequently the property has native string-to-[Type](#) conversion in such a way that the **x:Type** markup extension usage is optional.

There are additional programming constructs in the `x:` prefix/XAML namespace, which aren't as common. For details, see [XAML Namespace \(x\) Language Features](#).

Custom prefixes and custom types

For your own custom assemblies, or for assemblies outside the WPF core of **PresentationCore**, **PresentationFramework** and **WindowsBase**, you can specify the assembly as part of a custom `xmlns` mapping. You can then reference types from that assembly in your XAML, so long as that type is correctly implemented to support the XAML usages you are attempting.

The following is a basic example of how custom prefixes work in XAML markup. The prefix `custom` is defined in the root element tag, and mapped to a specific assembly that is packaged and available with the app. This assembly contains a type `NumericUpDown`, which is implemented to support general XAML usage as well as using a class inheritance that permits its insertion at this particular point in a WPF XAML content model. An instance of this `NumericUpDown` control is declared as an object element, using the prefix so that a XAML parser knows which XAML namespace contains the type, and therefore where the backing assembly is that contains the type definition.

```

<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:custom="clr-namespace:NumericUpDownCustomControl;assembly=CustomLibrary"
>
  <StackPanel Name="LayoutRoot">
    <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
  ...
  </StackPanel>
</Page>

```

For more information about custom types in XAML, see [XAML and Custom Classes for WPF](#).

For more information about how XML namespaces and code namespaces in assemblies are related, see [XAML Namespaces and Namespace Mapping for WPF XAML](#).

Events and XAML code-behind

Most WPF apps consist of both XAML markup and code-behind. Within a project, the XAML is written as a `.xaml` file, and a CLR language such as Microsoft Visual Basic or C# is used to write a code-behind file. When a XAML file is markup compiled as part of the WPF programming and application models, the location of the XAML code-behind file for a XAML file is identified by specifying a namespace and class as the `x:Class` attribute of the root element of the XAML.

In the examples so far, you have seen several buttons, but none of these buttons had any logical behavior associated with them yet. The primary application-level mechanism for adding a behavior for an object element is to use an existing event of the element class, and to write a specific handler for that event that is invoked when that event is raised at run-time. The event name and the name of the handler to use are specified in the markup, whereas the code that implements your handler is defined in the code-behind.

```

<Window x:Class="index.Window2"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window2" Height="450" Width="800">
  <StackPanel>
    <Button Click="Button_Click">Click me</Button>
  </StackPanel>
</Window>

```

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    var buttonControl = (Button)e.Source;
    buttonControl.Foreground = Brushes.Red;
}

```

```

Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
    Dim buttonControl = DirectCast(e.Source, Button)
    buttonControl.Foreground = Brushes.Red
End Sub

```

Notice that the code-behind file uses the CLR namespace `ExampleNamespace` and declares `ExamplePage` as a partial class within that namespace. This parallels the `x:Class` attribute value of `ExampleNamespace.ExamplePage` that was provided in the markup root. The WPF markup compiler will create a partial class for any compiled XAML file, by deriving a class from the root element type. When you provide code-behind that also defines the same partial class, the resulting code is combined within the same namespace and class of the compiled app.

For more information about requirements for code-behind programming in WPF, see [Code-behind, Event Handler, and Partial Class Requirements in WPF](#).

If you don't want to create a separate code-behind file, you can also inline your code in a XAML file. However, inline code is a less versatile technique that has substantial limitations. For more information, see [Code-Behind and XAML in WPF](#).

Routed events

A particular event feature that is fundamental to WPF is a routed event. Routed events enable an element to handle an event that was raised by a different element, as long as the elements are connected through a tree relationship. When specifying event handling with a XAML attribute, the routed event can be listened for and handled on any element, including elements that don't list that particular event in the class members table. This is accomplished by qualifying the event name attribute with the owning class name. For instance, the parent `StackPanel` in the ongoing `StackPanel` / `Button` example could register a handler for the child element button's `Click` event by specifying the attribute `Button.Click` on the `StackPanel` object element, with your handler name as the attribute value. For more information, see [Routed Events Overview](#).

Named elements

By default, the object instance that is created in an object graph by processing a XAML object element doesn't have a unique identifier or object reference. In contrast, if you call a constructor in code, you almost always use the constructor result to set a variable to the constructed instance, so that you can reference the instance later in your code. To provide standardized access to objects that were created through a markup definition, XAML defines the `x:Name` attribute. You can set the value of the `x:Name` attribute on any object element. In your code-behind, the identifier you choose is equivalent to an instance variable that refers to the constructed instance. In all respects, named elements function as if they were object instances (the name references that instance), and your code-behind can reference the named elements to handle run-time interactions within the app. This connection between instances and variables is accomplished by the WPF XAML markup compiler, and more specifically involve features and patterns such as [InitializeComponent](#) that won't be discussed in detail in this article.

WPF framework-level XAML elements inherit a `Name` property, which is equivalent to the XAML defined `x:Name` attribute. Certain other classes also provide property-level equivalents for `x:Name`, which is also usually defined as a `Name` property. Generally speaking, if you can't find a `Name` property in the members table for your chosen element/type, use `x:Name` instead. The `x:Name` values will provide an identifier to a XAML element that can be used at run-time, either by specific subsystems or by utility methods such as [FindName](#).

The following example sets `Name` on a `StackPanel` element. Then, a handler on a `Button` within that `StackPanel` references the `StackPanel` through its instance reference `buttonContainer` as set by `Name`.

```
<StackPanel Name="buttonContainer">
    <Button Click="RemoveThis_Click">Click to remove this button</Button>
</StackPanel>
```

```
private void RemoveThis_Click(object sender, RoutedEventArgs e)
{
    var element = (FrameworkElement)e.Source;

    if (buttonContainer.Children.Contains(element))
        buttonContainer.Children.Remove(element);
}
```

```

Private Sub RemoveThis_Click(sender As Object, e As RoutedEventArgs)
    Dim element = DirectCast(e.Source, FrameworkElement)

    If buttonContainer.Children.Contains(element) Then
        buttonContainer.Children.Remove(element)
    End If
End Sub

```

Just like a variable, the XAML name for an instance is governed by a concept of scope, so that names can be enforced to be unique within a certain scope that is predictable. The primary markup that defines a page denotes one unique XAML namespace, with the XAML namespace boundary being the root element of that page. However, other markup sources can interact with a page at run-time, such as styles or templates within styles, and such markup sources often have their own XAML namespaces that don't necessarily connect with the XAML namespace of the page. For more information on `x:Name` and XAML namespaces, see [Name, x:Name Directive](#), or [WPF XAML Namespaces](#).

Attached properties and attached events

XAML specifies a language feature that enables certain properties or events to be specified on any element, even if the property or event doesn't exist in the type's definitions for the element it's being set on. The properties version of this feature is called an attached property, the events version is called an attached event. Conceptually, you can think of attached properties and attached events as global members that can be set on any XAML element/object instance. However, that element/class or a larger infrastructure must support a backing property store for the attached values.

Attached properties in XAML are typically used through attribute syntax. In attribute syntax, you specify an attached property in the form `ownerType.propertyName`.

Superficially, this resembles a property element usage, but in this case the `ownerType` you specify is always a different type than the object element where the attached property is being set. `ownerType` is the type that provides the accessor methods that are required by a XAML processor to get or set the attached property value.

The most common scenario for attached properties is to enable child elements to report a property value to their parent element.

The following example illustrates the [DockPanel.Dock](#) attached property. The [DockPanel](#) class defines the accessors for [DockPanel.Dock](#) and owns the attached property. The [DockPanel](#) class also includes logic that iterates its child elements and specifically checks each element for a set value of [DockPanel.Dock](#). If a value is found, that value is used during layout to position the child elements. Use of the [DockPanel.Dock](#) attached property and this positioning capability is in fact the motivating scenario for the [DockPanel](#) class.

```

<DockPanel>
    <Button DockPanel.Dock="Left" Width="100" Height="20">I am on the left</Button>
    <Button DockPanel.Dock="Right" Width="100" Height="20">I am on the right</Button>
</DockPanel>

```

In WPF, most or all the attached properties are also implemented as dependency properties. For more information, see [Attached Properties Overview](#).

Attached events use a similar `ownerType.eventName` form of attribute syntax. Just like the non-attached events, the attribute value for an attached event in XAML specifies the name of the handler method that is invoked when the event is handled on the element. Attached event usages in WPF XAML are less common. For more information, see [Attached Events Overview](#).

Base types

Underlying WPF XAML and its XAML namespace is a collection of types that correspond to CLR objects and markup elements for XAML. However, not all classes can be mapped to elements. Abstract classes, such as [ButtonBase](#), and certain non-abstract base classes, are used for inheritance in the CLR objects model. Base classes, including abstract ones, are still important to XAML development because each of the concrete XAML elements inherits members from some base class in its hierarchy. Often these members include properties that can be set as attributes on the element, or events that can be handled. [FrameworkElement](#) is the concrete base UI class of WPF at the WPF framework level. When designing UI, you will use various shape, panel, decorator, or control classes, which all derive from [FrameworkElement](#). A related base class, [FrameworkContentElement](#), supports document-oriented elements that work well for a flow layout presentation, using APIs that deliberately mirror the APIs in [FrameworkElement](#). The combination of attributes at the element level and a CLR object model provides you with a set of common properties that are settable on most concrete XAML elements, regardless of the specific XAML element and its underlying type.

Security

XAML is a markup language that directly represents object instantiation and execution. That's why elements created in XAML have the same ability to interact with system resources (network access, file system IO, for example) as your app code does. XAML also has the same access to the system resources as the hosting app does.

Code Access Security (CAS) in WPF

Unlike .NET Framework, WPF for .NET doesn't support CAS. For more information, see [Code Access Security differences](#).

Load XAML from code

XAML can be used to define all of the UI, but it's sometimes also appropriate to define just a piece of the UI in XAML. This capability could be used to:

- Enable partial customization.
- Local storage of UI information.
- Model a business object.

The key to these scenarios is the [XamlReader](#) class and its [Load](#) method. The input is a XAML file, and the output is an object that represents all of the run-time tree of objects that was created from that markup. You then can insert the object to be a property of another object that already exists in the app. As long as the property is in the content model and has display capabilities that will notify the execution engine that new content has been added into the app, you can modify a running app's contents easily by dynamically loading in XAML.

See also

- [XAML Syntax In Detail](#)
- [XAML and Custom Classes for WPF](#)
- [XAML Namespace \(x:\) Language Features](#)
- [WPF XAML Extensions](#)
- [Base Elements Overview](#)
- [Trees in WPF](#)

XAML Services

4/15/2021 • 7 minutes to read • [Edit Online](#)

This topic describes the capabilities of a technology set known as .NET XAML Services. The majority of the services and APIs described are located in the assembly `System.Xaml`. Services include readers and writers, schema classes and schema support, factories, attributing of classes, XAML language intrinsic support, and other XAML language features.

About This Documentation

Conceptual documentation for .NET XAML Services assumes that you have previous experience with the XAML language and how it might apply to a specific framework, for example Windows Presentation Foundation (WPF) or Windows Workflow Foundation, or a specific technology feature area, for example the build customization features in [Microsoft.Build.Framework.XamlTypes](#). This documentation does not attempt to explain the basics of XAML as a markup language, XAML syntax terminology, or other introductory material. Instead, this documentation focuses on specifically using .NET XAML Services that are enabled in the `System.Xaml` assembly library. Most of these APIs are for scenarios of XAML language integration and extensibility. This might include any of the following scenarios:

- Extending the capabilities of the base XAML readers or XAML writers (processing the XAML node stream directly; deriving your own XAML reader or XAML writer).
- Defining XAML-usable custom types that do not have specific framework dependencies, and attributing the types to convey their XAML type system characteristics to .NET XAML Services.
- Hosting XAML readers or XAML writers as a component of an application, such as a visual designer or interactive editor for XAML markup sources.
- Writing XAML value converters (markup extensions; type converters for custom types).
- Defining a custom XAML schema context (using alternate assembly-loading techniques for backing type sources; using known-types lookup techniques instead of always reflecting assemblies; using loaded assembly concepts that do not use the common language runtime (CLR) `AppDomain` and its associated security model).
- Extending the base XAML type system.
- Using the `Lookup` or `Invoker` techniques to influence the XAML type system and how type backings are evaluated.

If you are looking for introductory material on XAML as a language, you might try [XAML overview \(WPF .NET\)](#). That topic discusses XAML for an audience that is new both to Windows Presentation Foundation (WPF) and also to using XAML markup and XAML language features. Another useful document is the introductory material in the [XAML language specification](#).

.NET XAML Services and `System.Xaml` in the .NET Architecture

.NET XAML Services and the `System.Xaml` assembly define much of what is needed for supporting XAML language features. This includes base classes for XAML readers and XAML writers. The most important feature added to .NET XAML Services that was not present in any of the framework-specific XAML implementations is a type system representation for XAML. The type system representation presents XAML in an object-oriented way that centers on XAML capabilities without taking dependencies on specific capabilities of frameworks.

The XAML type system is not limited by the markup form or run-time specifics of the XAML origin; nor is it limited by any specific backing type system. The XAML type system includes object representations for types, members, XAML schema contexts, XML-level concepts, and other XAML language concepts or XAML intrinsics. Using or extending the XAML type system makes it possible to derive from classes like XAML readers and XAML writers, and extend the functionality of XAML representations into specific features enabled by a framework, a technology, or an application that consumes or emits XAML. The concept of a XAML schema context enables practical object graph write operations from the combination of a XAML object writer implementation, a technology's backing type system as communicated through assembly information in the context, and the XAML node source. For more information on the XAML schema concept. see [Default XAML Schema Context and WPF XAML Schema Context](#).

XAML Node Streams, XAML Readers, and XAML Writers

To understand the role that .NET XAML Services plays in the relationship between the XAML language and specific technologies that use XAML as a language, it is helpful to understand the concept of a XAML node stream and how that concept shapes the API and terminology. The XAML node stream is a conceptual intermediate between a XAML language representation and the object graph that the XAML represents or defines.

- A XAML reader is an entity that processes XAML in some form, and produces a XAML node stream. In the API, a XAML reader is represented by the base class [XamlReader](#).
- A XAML writer is an entity that processes a XAML node stream and produces something else. In the API, a XAML writer is represented by the base class [XamlWriter](#).

The two most common scenarios involving XAML are loading XAML to instantiate an object graph, and saving an object graph from an application or tool and producing a XAML representation (typically in markup form saved as text file). Loading XAML and creating an object graph is often referred to in this documentation as the load path. Saving or serializing an existing object graph to XAML is often referred to in this documentation as the save path.

The most common type of load path can be described as follows:

- Start with a XAML representation, in UTF-encoded XML format and saved as a text file.
- Load that XAML into [XamlXmlReader](#). [XamlXmlReader](#) is a [XamlReader](#) subclass.
- The result is a XAML node stream. You can access individual nodes of the XAML node stream using [XamlXmlReader](#) / [XamlReader](#) API. The most typical operation here is to advance through the XAML node stream, processing each node using a "current record" metaphor.
- Pass the resulting nodes from the XAML node stream to a [XamlObjectWriter](#) API. [XamlObjectWriter](#) is a [XamlWriter](#) subclass.
- The [XamlObjectWriter](#) writes an object graph, one object at a time, in accordance to progress through the source XAML node stream. Object writing is done with the assistance of a XAML schema context and an implementation that can access the assemblies and types of a backing type system and framework.
- Call [Result](#) at the end of the XAML node stream to obtain the root object of the object graph.

The most common type of save path can be described as follows:

- Start with the object graph of an entire application run time, the UI content and state of a run time, or a smaller segment of an overall application's object representation at run time.
- From a logical start object, such as an application root or document root, load the objects into [XamlObjectReader](#). [XamlObjectReader](#) is a [XamlReader](#) subclass.

- The result is a XAML node stream. You can access individual nodes of the XAML node stream using [XamlObjectReader](#) and [XamlReader](#) API. The most typical operation here is to advance through the XAML node stream, processing each node using a "current record" metaphor.
- Pass the resulting nodes from the XAML node stream to a [XamlXmlWriter](#) API. [XamlXmlWriter](#) is a [XamlWriter](#) subclass.
- The [XamlXmlWriter](#) writes XAML in an XML UTF encoding. You can save this as a text file, as a stream, or in other forms.
- Call [Flush](#) to obtain the final output.

For more information about XAML node stream concepts, see [Understanding XAML Node Stream Structures and Concepts](#).

The XamlServices Class

It is not always necessary to deal with a XAML node stream. If you want a basic load path or a basic save path, you can use APIs in the [XamlServices](#) class.

- Various signatures of [Load](#) implement a load path. You can either load a file or stream, or can load an [XmlReader](#), [TextReader](#) or [XamlReader](#) that wrap your XAML input by loading with that reader's APIs.
- Various signatures of [Save](#) save an object graph and produce output as a stream, file, or [XmlWriter](#)/[TextWriter](#) instance.
- [Transform](#) converts XAML by linking a load path and a save path as a single operation. A different schema context or different backing type system could be used for [XamlReader](#) and [XamlWriter](#), which is what influences how the resulting XAML is transformed.

For more information about how to use [XamlServices](#), see [XamlServices Class and Basic XAML Reading or Writing](#).

XAML Type System

The XAML type system provides the APIs that are required to work with a given individual node of a XAML node stream.

[XamlType](#) is the representation for an object - what you are processing between a start object node and end object node.

[XamlMember](#) is the representation for a member of an object - what you are processing between a start member node and end member node.

APIs such as [GetAllMembers](#) and [GetMember](#) and [DeclaringType](#) report the relationships between a [XamlType](#) and [XamlMember](#).

The default behavior of the XAML type system as implemented by .NET XAML Services is based on the common language runtime (CLR), and static analysis of CLR types in assemblies by using reflection. Therefore, for a specific CLR type, the default implementation of the XAML type system can expose the XAML schema of that type and its members and report it in terms of the XAML type system. In the default XAML type system, the concept of assignability of types is mapped onto CLR inheritance, and the concepts of instances, value types, and so on, are also mapped to the supporting behaviors and features of the CLR.

Reference for XAML Language Features

To support XAML, .NET XAML Services provides specific implementation of XAML language concepts as defined for the XAML language XAML namespace. These are documented as specific reference pages. The language features are documented from the perspective of how these language features behave when they are processed

by a XAML reader or XAML writer that is defined by .NET XAML Services. For more information, see [XAML Namespace \(x:\) Language Features](#).